FLEXIBLE QOS-MANAGED STATUS DISSEMINATION MIDDLEWARE FRAMEWORK FOR THE ELECTRIC POWER GRID

By

KJELL HARALD GJERMUNDRØD

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2006

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of KJELL HARALD GJERMUNDRØD find it satisfactory and recommend that it be accepted.

Chair

ACKNOWLEDGEMENT

First of all I would like to thank my advisor Dave Bakken for his advice and guidance throughout my studies. I would also like to thank my other committee members Carl Hauser, Anjan Bose, John Shovic, and Jack Hagemenister. Especially, I would like to thank Carl Hauser for his active role in my research and the valuable feedback and insight.

Special thanks also go to my fantastic wife Ioanna for her support and help during the years that we have been studying together. This accomplishment would not have been possible without her. Also thanks go out to all the acquaintances and friends that I have had during my time in Pullman. They all made the stay so much more enjoyable.

Finally, I would like to thank the different organizations that have funded me throughout my studies. I have received teaching and research assistantship as well as a teaching fellowship from Washington State University, stipend from the Norwegian State Educational Loan Fund, and stipend from Trigeo Network Security. In addition, my research has been supported in part by Grant NSF CNS 05-24695 (CT-CS: Trustworthy Cyber Infrastructure for the Power Grid) and Grant CCR-0326006 from the National Science Foundation and by Grant #60NANB1D0016 (Critical Infrastructure Protection Program), in a subcontract to Schweitzer Engineering Labs Inc from the National Institute of Standards and Technology. In addition, I would like to thank Eastern Washington University for letting me use their computational resources to run the experiments that were conducted for this research.

ATTRIBUTION

Portions of this dissertation were published in the following conferences or workshops.

- Carl H. Hauser, David E. Bakken, Ioanna Dionysiou, K. Harald Gjermundrød, Venkata S. Irava, Joel Helkey, and Anjan Bose, Security, trust and QoS in next-generation control and communication for large power systems, *International Journal of Critical Infrastructures*, Inderscience, to appear, 2007.

- Carl H. Hauser, David E. Bakken, Ioanna Dionysiou, K. Harald Gjermundrød, Venkata S. Irava and Anjan Bose, Security, trust, and QoS in next-generation control and communication for large power systems, in *Proceedings of the Workshop on Complex Network and Infrastructure Protection (CNIP06)*, Rome, Italy, March 28-29, 2006.

- Ryan A. Johnston, Carl H. Hauser, K. Harald Gjermundrød, and David E. Bakken. Distributing Time-synchronous Phasor Measurement Data Using the GridStat Communication Infrastructure. In *Proceedings of 39th Annual Hawaii International Conference on System Sciences (CD/ROM)*, Computer Society Press, Kauai, HI, January 4–7, 2006.

- Harald Gjermundrød, Ioanna Dionysiou, David Bakken, and Carl Hauser, Fault Tolerance Mechanisms in Status Dissemination Middleware, in *Supplement of the International Conference on Dependable Systems and Networks (DSN-2002)*, IEEE/IFIP, San Francisco, CA, June 2003, B-56–57.

- D. Bakken, A. Bose, C. Hauser, I. Dionysiou, H. Gjermundrød, L. Xu, and S. Bhowmik, Towards More Extensible and Resilient Real-Time Information Dissemination for the Electric Power Grid, in *Proceedings of Power Systems and Communications Systems for the Future, International Institute for Critical Infrastructures*, Beijing, China, September 2002.

- I. Dionysiou, H. Gjermundrød, and D. Bakken, Fault Tolerance Issues in Publish-Subscribe Status Dissemination Middleware for the Electric Power Grid, in *Supplement of the International Conference on Dependable Systems and Networks (DSN-2002), IEEE/IFIP*, Washington, DC, June 23–26, 2002, B-62–63.

- D. Bakken, A. Bose, C. Dyreson, S. Bhowmik, I. Dionysiou, H. Gjermundrød, and Lin Xu, Impediments to Survivability of the Electric Power Grid and Some Collaborative EE-CS Research Issues to Solve Them, in *Proceedings of the fourth Information Survivability Workshop (ISW-2001/2002), Impediments to Achieving Survivable Systems*, Vancouver, BC, Canada, March 2002.

K. Harald Gjermundrød was the chief architect, designer, and implementer of the GridStat middleware framework to date. This framework has been used as a baseline for the following masters theses [43, 44, 39], as well as 3 ongoing ones. It also defines the routing hooks and topologies for the following PhD. Dissertation [41], which will be integrated into the framework in the next year or so. This framework has been deployed for 3 years in a technology evaluation project using live power grid data from regional gas and electric utility Avista, who also sponsored 3 senior projects using this framework. This framework is being used to integrate research from multiple researchers in the NSF CyberTrust Center called Trustworthy Cyber Infrastructure for the Power Grid (TCIP) [29]. Finally, plans are underway to integrate GridStat into the new Integrated Energy Operations Center at Pacific Northwest National Laboratory.

FLEXIBLE QOS-MANAGED STATUS DISSEMINATION MIDDLEWARE FRAMEWORK FOR THE ELECTRIC POWER GRID

Abstract

by Kjell Harald Gjermundrød, Ph.D.
Washington State University
August 2006

Chair: David E. Bakken

Electric power grids are complex interconnected systems that in North America can span more than a thousand miles. They must be operated such that the dynamic balance between supply and demand is maintained. Grids exhibit many power dynamics that are global over the entire grid, yet their control mechanisms and almost all of their operational data on the current dynamics and configuration of the grid status data are local in nature, typically limited to a single substation. The reason for this is that the communication system that is used to coordinate and monitor utility operations was designed largely in response to the 1965 blackout in the US Northeast. However, since then network and related technologies have improved dramatically.

Status dissemination middleware is a new specialization of the publish-subscribe model that takes advantage of the semantics of status data. This dissertation presents a status dissemination middleware framework named GridStat for the electric power grid and other critical infrastructures. It takes advantage of the semantics of status data to optimize delivery and to manage its subscriptions for quality of service. In doing so, GridStat allows status information to be disseminated over a wide-area. The architecture, design, and implementation of this framework are described, along with its baseline mechanisms to improve efficiency and resilience. In addition, the rationale, design, and implementation of four mechanisms are presented. These mechanisms are limited flooding mechanisms, operating modes mechanism, filtering with multicast mechanism, and condensation function mechanism. This dissertation also presents an experimental evaluation of the GridStat framework.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

The continuous and correct functioning of the critical infrastructures is of paramount importance to any nation. These infrastructures usually cover a wide geographic area that consists of a large number of participants residing in different administration domains. These participants are simultaneously collaborating on reliability and safety issues while competing in the business marketplace. Each of these infrastructures must have a control system in place so that the participants are able to operate the infrastructure within its safety margins.

An example of such a critical infrastructure is the North America electric power grid. It covers most of the USA, part of Canada, and part of Mexico, and consists of approximately 3,500 organizations. The current communications infrastructure for the electric power grid was designed in the 1960s and consists of a fixed, hardwired and sequential data acquisition infrastructure, which is not able to meet the current needs and trends of the power industry [10, 37]. The power grid engineering community is beginning to recognize a need for a fundamental transformation in the capabilities of the communication infrastructure that supports power grid operations, [23, 27, 75, 53, 3, 40]. What is needed is a communications system responsible for distributing status information to legitimate parties in a timely, secure and accurate manner. This need has not been met to date for at least two reasons: very low levels of R&D investment by electric utilities, and the fact that the R&D needed is in the domain of computer science, not electric power.

Recent advances and trends in network communications make such an information infrastructure feasible. Bandwidth availability, lower hardware costs, increased computer speed, lower latencies, middleware technologies, as well as the ability to provide QoS at multiple levels of a wide-area distributed computing system can be appropriately utilized in developing a new communication paradigm.

This dissertation presents a novel middleware framework for wide-area status dissemination systems. This chapter will overview relevant computer science areas to provide a context to enable the reader to better understand the research contributions of this dissertation. First distributed computing systems are presented, followed by middleware. Following this is a discussion of the publish-subscribe paradigm, which this research extends, followed by the thesis statement which this research validates. Finally, this chapter ends with an outline of the organization of the remainder of this dissertation.

1.1 Distributed Computing Systems

Distributed computing systems are information systems that are constructed by integrating separate components (personal computer, server, embedded device, etc), which are connected by various forms of communication services. Distributed system components cooperate to achieve some common goal. The reasons to use distributed systems are manyfold. First, systems are distributed due to the nature of the application that uses them. Second, resources, both with respect to processing and data, can be shared among the components. Therefore, components can be specialized/dedicated to provide resources that all share. The third reason is that components can be replicated, which means that multiple components provide the same resources. If one of the replicated components fails, then the failed resource can be acquired from one of the replicas.

A distributed system can be architected to take advantage of the strength of the individual components and provide a system that is well beyond the capabilities of any of the individual components. The size and the degree of how tightly coupled such systems are depend on their usage area. For instance, avionics mission computing systems are small and tightly coupled, while global telecommunication systems and the Internet are very large and loosely coupled systems. Another example is Skype which is both a tightly and a loosely coupled system. The users can register themselves and send messages to each other (loosely coupled), while if two or more users want to talk to each other using voice over IP this is done in real time (tightly coupled.)

Interconnecting components is challenging due to the components' heterogeneity. These components may be of different hardware platforms, using different operating systems and network technologies to communicate. Additionally, the application may be written in different programming languages. In order to architect and develop a distributed system, such heterogeneity must be supported. It is tedious and error-prone to build such systems using only the API that the involved operating systems and programming languages provide. This is the main reason for the existence of middleware, which is described further in Section 1.2.

The term Quality of Service (QoS) is used in distributed computing system to characterize/specify the management of the "non-functional" properties of a distributed application [5]. These properties include, among others, reliability, performance, and security. In order for a distributed system to be able to provide these properties, the resources available need to be managed. In a LAN setting, it is reasonable to assume that resources are plentiful, belong to the same domain, and their availability is predictable. When a distributed application from the LAN setting is moved into the WAN setting, these assumptions do not hold true [81]. The reason lies on the dynamic nature of the WAN and the difficulty to manage its resources at the end-points.

1.2 Middleware

Just like an operating system provides an abstraction of the underlying hardware that provides services to the user applications through mechanisms, so can middleware provide abstractions and services to distributed applications. Middleware lies beneath the application layer and above the operating system. One of the design goals of middleware is to solve the problems caused by heterogeneity in platforms, networks, programming languages, operating systems, and vendor implementations. It also provides a set of services that the application layer programmer can use and reuse to build distributed applications [5]. In essence, middleware masks system heterogeneity and provides transparency to the application layer to ease in the development of distributed applications.

Different middleware styles provide different kinds of abstractions to the programmer:

Distributed Tuples Middleware frameworks of this type provide the abstraction that each entity has access to one global bag of values (tuples) that can be added, removed, and updated. In reality, all the entities share a distributed bag (dataspace) where they all add, remove, and update the shared tuples. The Linda programming language [30] and Sun's Jini [51] are examples of this kind of middleware.

Remote Procedure Call (RPC) This type of middleware framework provides the abstraction that a call to a remote procedure/function looks like a call to a local procedure/function [9]. Therefore, a remote procedure call is synchronous and blocking, where a client process making a request to a server is blocked until the server returns the result. The Java and Ada programming languages provide mechanisms for RPC interactions.

Message-oriented The key feature of this middleware framework is the use of message exchange for distributing data. Message queuing is an indirect communication model that allows applications to communicate via message queues, rather than by calling each other directly. Message queuing always implies a connectionless model. IBM's MQSeries and Oracle's Advanced Queuing are examples of message-oriented middleware.

Distributed Object Computing (DOC) In order to match object oriented paradigms and programming languages, distributed object middleware frameworks provide the equivalent abstraction that RPC does to non-object oriented programming languages. DOC provides the abstraction at the object level, where a remote object behaves to the client just like a local object. CORBA, .NET, and Java's RMI are the major representatives of this category, published by OMG, Microsoft, and Sun Microsystems, respectively.

1.3 Publish-Subscribe

One specialization of the message-oriented middleware framework is the publish-subscribe (pub-sub) paradigm, where entities are either *publishers* (information producers) or *subscribers* (information consumers). The publishers and subscribers interact with each other through a network of event notification servers. A powerful feature of the pub-sub paradigm is that it is decoupled in time, space and flow with respect to the publishers and subscribers [24]. The interacting entities, publishers and subscribers, do not need to know each other (space decoupling), or actively participate in the interaction at the same time (time decoupling). Moreover, neither entity is blocked while producing or consuming events (flow decoupling).

Pub-sub middleware (PSM) products are usually divided into three categories depending on how the subscriber subscribes to the information. These categories are topic based, content based, and type based. Each of the categories is explained below:

Topic-based In this class of PSM, the publisher produces information that belongs to a specific topic or subject and the subscriber subscribes to that specific topic or subject. This communication paradigm resembles the group communication paradigms where participants belong to groups and receive all the information that is posted to that group. For the topic-based PSM, the subscribers become members of the topic and receive all the information that is published for that topic.

Topics can be organized either as flat addressing or hierarchical addressing. Flat addressing has the obvious problem of scaling and lacks the flexibility of subgrouping the topics. Hierarchical addressing provides flexibility by allowing the subscribers to use wildcards in their topic expressions. Furthermore, subscribers are able to consume all topics of a subgroup.

Content-based The content-based PSM adds more flexibility than the topic-based, with a greater expressiveness in information that the subscribers want to receive. The subscriber subscribes to information using name-value pairs of properties that it wants to receive. The property pairs can usually be combined with logical operators (*and*, *or*, etc.) to form subscription patterns. These patterns can be translated into filters which filter out the information that a specific subscriber wants to receive. There are different methods of representing the subscription patterns, such as:

String: It represents the filtering by a subscription grammar, for instance SQL or OMG's Default Constraint Language.

Template object: An object represents the filtering by its type and its attributes, so the filtering is done by first matching the object type and then testing if all the attributes can be satisfied.

Executable code: A predicate object is provided as the filtering mechanism. At runtime the predicate object is executed and it filters out the unwanted messages.

Type-based Type-based pub-sub can be seen as a hybrid between DOC and PSM. In DOC the system consists of objects that can be accessed remotely either synchronously or asynchronously, while PSM asynchronously sends and receives messages. By making the messages first-class objects, type checking of these systems can be done at compile time (more robust) and explicit casting is no longer necessary (resulting in speed up). Type hierarchies can be created by the use of inheritance for objects or sub-typing. In content-based systems, objects may be marshalled into messages exposing their private data, or the private data members of the objects can be copied into the message header as properties. Filtering is applied to the exposed data members of the message. On the other hand, in type-based systems the filtering can be done on the object itself which may provide filtering methods. In this way, the object's private data members are not exposed [25].

A new category is, *status dissemination middleware* with multiple QoS and this is the category that this dissertation work falls into. The difference between a general pub-sub framework and one that is specialized for *status variables* is in the information that is exchanged between the publisher and the subscriber. The former framework exchanges information that can be anything and can be published at any rate, compared to the latter that disseminates status variables of specified data types at a regular interval. Status dissemination middleware can take advantage of the semantics of status variables to provide (with some degree of guarantee) services for disseminating status variables subject to a number of QoS categories. Such services may be what the critical infrastructures need for their control and monitoring systems.

1.4 Thesis Statement

This dissertation proposes a status dissemination middleware framework that is a specialization of the more general pub-sub paradigm. The novel middleware framework was architected, designed and developed in a way that accommodates several QoS categories such as timeliness, rate, fault tolerance, security, and others. Subscribers can specify the end-to-end QoS they require and the middleware framework will guarantee these end-to-end requirements, if they are satisfiable according to the available resources and QoS policies. Pub-sub middleware frameworks already exist, but none of them are specialized or optimized to deliver *status events* with end-to-end QoS guarantees [64]. As a result, the focus of this dissertation is to investigate mechanisms that will provide services for this new class of scalable, efficient, and resilient status dissemination middleware.

The challenges for this research are to identify what mechanisms are needed to provide services for this new class of middleware frameworks. In order to design, develop, and evaluate (both performance and usefulness) a status dissemination middleware framework providing the services stated above, the following questions must be answered:

- What mechanisms are required in a scalable, efficient, and resilient status dissemination middleware framework?
- What features of status dissemination middleware and services will enhance and ease the

development of distributed applications?

This dissertation will investigate the validity of the following statement:

Thesis: middleware frameworks can provide efficient and resilient delivery of status information across a wide-area distributed computing system.

In order to test the validity of the above thesis, this dissertation makes the following research contributions:

- Design and implementation of an architecture for an efficient and resilient status dissemination middleware framework
- A baseline set of mechanisms to deliver status events and control interfaces to them (GridStat entities).
  - Leaf QoS broker and QoS broker
  - Status router and edge status router
  - Publisher, subscriber, and condensation creator
  - Interaction models at the subscriber push or pull and group subscription to temporal related variables
  - Publishing of derived values provided
- Provide mechanisms to enhance efficiency
  - Multicast rate-filtering routing for temporally related variables
  - Limited flooding
  - Condensation function
  - Alert type to bypass the FIFO queues

  - Packing of status events

- Provide mechanisms to enhance resilience

  - Operational modes

  - Recover from failures in (edge) status routers

  - Extrapolation function

  - Hooks to provide for redundant path algorithms

- An experimental evaluation of the framework's performance

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows: The GridStat framework is described in Chapter 2. The different entities that make up this framework are presented in Chapter 3. Four of the mechanisms are then presented in detail in Chapter 4 – 7. Experimental results of scalability of the framework is presented in Chapter 8, followed by related work in Chapter 9. Finally, the conclusions and future work are presented in Chapter 10.

CHAPTER 2

STATUS DISSEMINATION AND GRIDSTAT

There are many different kinds of interactions in large distributed systems, and corresponding architectures for the middleware supporting them. In the earlier days of distributed computing, many were client-server, with its 1:1 relationship between entities. In recent decades, publish-subscribe (pub-sub) has become widely used. This architecture features a 1:N topology where events generated by a single publisher are delivered to different subscribers. Pub-sub systems separate the publisher and subscribers both spatially and sometimes even temporally by an event bus, which is one or more servers which cooperate to deliver events to subscribers. Some pub-sub systems offer different kinds of QoS guarantees for delivery, typically timeliness but sometimes also reliability and persistence.

The concept of status dissemination (SD) is a specialization of pub-sub. While the pub-sub architecture provides the abstraction of a distributed event, the main abstraction provided by SD is that of a distributed variable which is periodically updated. As noted in Chapter 9, PASS [79] is the only SD system, other than this research, which we are aware of. PASS provided Boolean updates with no QoS, and was intended for tracking whether entities were currently running in a geographically large, bandwidth-constrained network while avoiding the alternative of 1:1 pinging to ascertain if an entity was up or down. This research develops richer abstractions with QoS to extend its use beyond up-down status tracking to include a wider set of data types. The abstractions in this research allow richer APIs (including the notion of a cached status variable which can be used much like a local variable) and much more fine-grained QoS management than either pub-sub or PASS do, and its mechanisms can offer a richer set of adaptations to contingencies either in the application domain (e.g., an impending electricity blackout) or the IT domain (e.g., a server crashing or a denial of service attack).

This research develops the abstractions and supporting mechanisms to provide wide-area real-time reliable delivery of information streams for a much broader set of distributed applications. In such applications, status variables are typically measurements of physical quantities or control settings which will be updated to all subscribers. Each measurement is a timestamped value called status event. A status variable is either periodic or aperiodic, depending on whether its status events are published at a given interval or not. A sequence of status events at regular intervals, a *data stream*, is the main characteristic of a periodic status variable. However, there are cases where the direct measurement itself is not as useful as the detection of the measurement meeting some particular condition, such as crossing a threshold. In such a case, the status variable is aperiodic since status events, or an *alert stream*, are generated whenever the condition occurs. An *event stream* refers to either a data stream or an alert stream. Regardless of status variable type, an event stream in our system is subject to a number of delivery guarantees, such as latency, interval, and spatial redundancy - these are also known as Quality of Service (QoS) requirements [63]. SD provides QoS guarantees by taking advantage of the static nature of periodic status variables and making resource allocation provisions for the aperiodic ones.

One application domain suitable for SD deployment is the control systems for critical infrastructures such as the electric power grid. It is observed that the topology and information flows (from producers to consumers) in these large control systems is very static [37]. A large fraction of the produced data comes from sensors reading devices at a regular interval and producing events containing the value. In SD, these periodic producers of information notify the system that they produce an event of size s every x milliseconds. In the electric power grid, there is also aperiodic data produced when exceptional cases occur. An example is a sensor that checks if a transmission line is still operational. If the transmission line sags into a tree, the sensor is triggered and produces an alert. The aperiodic producers of information notify the system that an alert stream is produced.

This research develops the abstractions and mechanisms for the GridStat framework [6, 21, 32, 45, 38]. An overview of the framework, along its interactions with endpoints (publishers, subscribers, QoS management policies) is presented. Finally, an overview of the baseline mechanisms that are provided by the framework is described.

2.1 GridStat Architecture

GridStat targets application domains where the vast majority of data is published periodically, while still allowing for a small amount of aperiodic data to be published. GridStat was developed with the needs of the electric power grid in mind, specifically the severe shortcomings of its communication system [37]. The major communication requirements were to provide end-to-end reliability and timeliness for event streams scalable into the wide-area setting. In addition, the architecture should enable the provision of redundancy so that disjoint communication paths for an event stream could be provided between data producers and consumers. The timeliness guarantee is satisfied for each disjoint path that the consumer specified to tolerate *n-1* failures, where *n* is the number of disjoint paths. The following assumptions were made in order to meet the requirements while not providing unnecessary generality which would almost certainly be less scalable and fast:

- Subscriptions are long-lived, and the vast majority of them are set up offline and in advance. Thus, event streams are fairly static, with the possibility of dynamically adding new streams. There is thus no need to focus on providing a strict (and low) upper bound on how long it takes to add a new stream, but instead, the focus is on maintaining the end-to-end QoS guarantees once it is added.

- The framework operates on a dedicated network and has complete control over the available resources (routers and communication links). GridStat could certainly be deployed where some of its redundant paths are over the best-effort Internet, but there must always be a baseline of dedicated network links providing connectivity to all entities for GridStat to work adequately.

The design of GridStats mechanisms take advantage of the highly static nature of the topologies of interactions between publishers and subscribers, by preallocating paths between the data producers and the intended recipients, in order to guarantee delivery of the requested information. Dynamic subscription requests can also be made but they are honored if they do not exhaust the available resources. An important feature of GridStat is its ability to systematically instrument its explicit topology to collect resource information from the network.

In order to facilitate the end-to-end dissemination of data within a wide area setting, there are two major categories of tasks that the framework accomplishes:

- Category 1: Scalability and extensibility of allocating data streams between publishers and subscribers. The communication paths must be allocated to meet the streams QoS guaranteed without interfering with the QoS requirements of already established event streams.
- Category 2: Forwarding events from producers to consumers at the specified interval without dropping them (as long as the system operates within its failure model), something that causes QoS violations.

A tree topology fits nicely for the tasks performed in the first category. For the requirements of the second category of tasks, an appropriate topology is that of a more general graph. The resulting GridStat architecture, depicted in Figure 2.1, combines the two topologies by supporting two planes: a *management plane* and a *data plane*. The role of the management plane is to control the resources in the data plane in order to provide QoS guarantees and scale to the wide-area setting. The role of the data plane is to forward events from the publishers to the subscribers, as efficiently as possible, so as to provide end-to-end timeliness and redundancy guarantees. The advantage of separating data from its management is that the event stream is not affected by any bottlenecks caused by management tasks. Clark in [18] discusses the design philosophy behind the IP architecture, and in his conclusion he alludes to an architecture where the routers maintain state for each flow in order to improve the resource management and accountability. The GridStat architecture fits this general model.

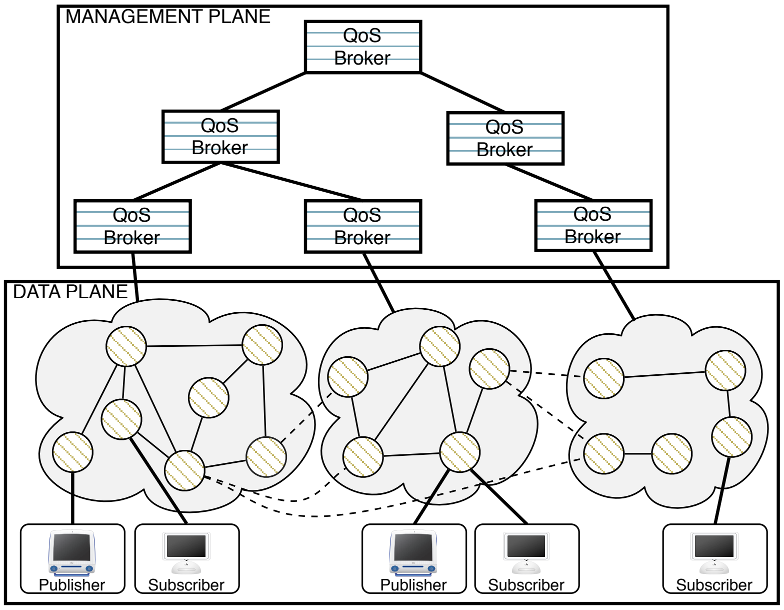


Figure 2.1: Status dissemination middleware.

Prior to a more detailed discussion of the two planes of functionality, it is essential to present the main architectural building blocks that assemble GridStat planes. GridStat is built on the publisher-subscriber paradigm and hence, its architecture is appropriately composed by components such as publishers, subscribers, and event servers. As it was mentioned earlier, status variables represent some device which has a state that changes over time. Status events are an updated value for a status variable at a given time and are encapsulated into messages that flow from publishers to subscribers. A status variable has static properties, such as publisher name, status name, and publishing interval. In addition to these, a status variable has dynamic attributes beyond its direct value that are time-related, such as moving average, rate of change, and minimum and maximum values over a specified number of generated events. These can be subscribed to in addition to the direct value of the status variable.

Producers of status events (publishers) post their information without knowing by whom or where they will be consumed. The communication infrastructure relies on a network of event servers, called *status routers*, that acts as a forwarding service in a transparent manner to the end-points (publishers and subscribers). Consumers (subscribers) subscribe to named status variables.

The capability of embedding QoS properties in the framework requires additional components to manage the requested QoS. As a result, the entire network is managed by a hierarchy of *QoS brokers*.

Below, each of the planes is explained in more detail. Then the structured datatypes that the framework uses are presented, followed by the interaction model that the framework offers to the application layer of a publisher and subscriber.

2.1.1 Management Plane

The management plane provides services that involve primarily registration (and unregistration) of publications and subscriptions, path allocation decisions, as well as secondary services such as placement of condensation functions.

The lowest level of the management hierarchy is composed of *leaf QoS brokers*, which manage a pre-defined domain consisting of a set of status routers, publishers and subscribers; that's called a *cloud*. Each leaf QoS broker has complete knowledge of its own administrative domain and its resources. These resources include *event channels* (with their bandwidth and latency), status routers (with their queues and computational resources), and any other computational resources (used by the condensation function). The responsibility of the leaf QoS broker is to allocate and deallocate *subscription paths* from the publishers to the subscribers, subject to the QoS requirements, within the cloud that it controls. Requests to allocate or deallocate a subscription path come directly from one of its subscriber or from its parent in the QoS broker hierarchy. In the latter case, the request is part of establishing an *inter-cloud* (involving multiple clouds) subscription. Leaf QoS brokers populate the routing table of their status routers as part of the path allocation process. If there are any conflicts between the QoS requirements and the cloud capacity, the leaf QoS broker resolves this conflict according to its policies.

QoS brokers form a hierarchy of QoS management, where its leaves are represented by leaf QoS brokers. The QoS brokers manage multiple clouds and are responsible for allocating and deallocating inter-cloud subscriptions. Such a subscription request is delegated from a leaf QoS broker to the QoS broker that controls all the clouds necessary to establish the subscription path. That QoS broker uses its routing algorithm to decide which clouds and inter-cloud event channels should be used for the subscription path. After that, commands are sent to its subordinates to establish the end-to-end QoS subscription path.

Scalability is a desired property when determining subscription paths. GridStat allows for scalable subscriptions by traversing the management hierarchy until a QoS broker is found that is an ancestor QoS broker for the leaf QoS brokers that administer the publisher and subscriber involved in the subscription path. This ancestor QoS broker allocates a path by recursively delegating the construction of sub-paths, whose assembling creates the desired end-to-end path. The abstractions applied to the higher levels of the hierarchy in determining and establishing inter-cloud paths are still under investigation and is not the focus of this dissertation.

2.1.2 Data Plane

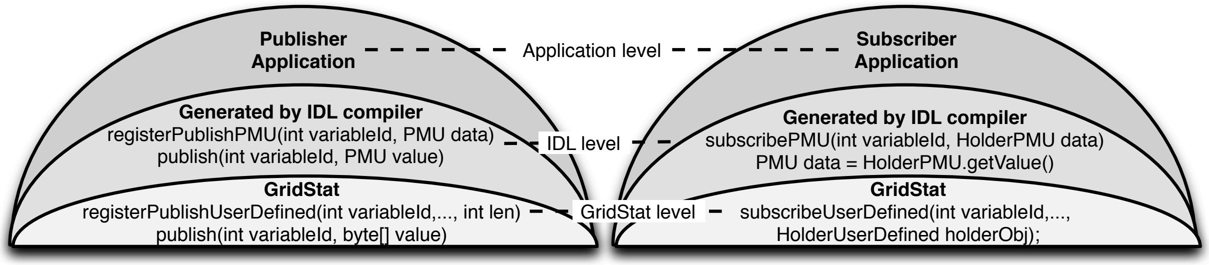
The second plane is the data plane, which provides a virtual message bus and a proxy to the management plane for publishers and subscribers. Publishers and subscribers use the virtual message bus to send and receive event streams. They request services from the management plane by sending requests to the data plane, which acts as a proxy and forwards the requests to the management plane. Hence, the end-points (publishers and subscribers) interact directly only with the data plane.

The data plane consists of status routers that together constitute a virtual message bus. Each status router is similar to an internet router with added functionality in order to forward streams of event messages. The forwarding table in the status router is populated by the management plane in a controlled manner to make sure that the resources available are not exhausted. Each of the status routers has a connection to the leaf QoS broker that controls it. There are three types of status routers and the type depends on the role of the status router. The first type of status routers is the *edge status router*. An edge status router is the one that publishers and subscribers connect to and as a result it has a dual role: it provides a proxy to the management plane for its end-points and it forwards events like any other status router. The second type of status routers is the *border status router* and this router serves as the end-points of one or more event channels that connect two clouds. Border status routers differ from status routers in the way they are used by the management plane. A QoS broker controls the inter-cloud event channel capacity rather than a leaf QoS broker. When establishing inter-cloud paths, the routing algorithms must be aware of the border status routers that have connections to other clouds so as to utilize them as a cloud's entry and exit points. Finally, the last type of status routers are those that are not edges or borders and hence have no additional functionality. They accommodate the internal workings of streaming and are transparent to publishers, subscribers, and the QoS brokers.

The data plane takes advantage of multicast, whenever possible, and filters streams of events according to the different subscriptions on the paths from the publishers to the subscribers. Since the management plane controls all the resources in the data plane and the amount of data that can be sent into the data plane is controlled, the latency from the publisher to the subscriber can be bound. The framework therefore provides timeliness guarantees to the subscribers. If there is enough redundancy in the data plane in terms of event channels and status routers, then reliability QoS is also provided to the subscriber by establishing disjoint paths. The routing algorithms to establish disjoint path subject to multiple QoS requirement is still under investigation and is not the focus of this dissertation.

2.1.3 *Structured Datatypes*

GridStat supports structured status variables and that allows for compile time type-checking. When the publishers register to publish a variable, they also register the type of the variable. The subscriber specifies the type of the variable that it is subscribing to and it also supplies an object of the same type. The type is forwarded to the QoS management and if the type the subscriber specified is different from the one that the publisher announced during registration, the subscription request is denied and an error code is returned to the subscriber.



Publisher Application
Application level
Subscriber Application
Generated by IDL compiler
registerPublishPMU(int variableId, PMU data)
publish(int variableId, PMU value)
IDL level
Generated by IDL compiler
subscribePMU(int variableId, HolderPMU data)
PMU data = HolderPMU.getValue()
GridStat
registerPublishUserDefined(int variableId,..., int len)
publish(int variableId, byte[] value)
GridStat level
GridStat
subscribeUserDefined(int variableId,...,
HolderUserDefined holderObj);

Figure 2.2: Structured data types

Figure 2.2 depicts the layering of the datatypes. At the application layer, the publishers and the subscribers are interacting with objects of the same type. The Interface Definition Language (IDL) layer is produced by an IDL compiler that marshalls/unmarshalls the variables into the transport layer format. The transport layer format is shown in Figure 2.3. The format provides for different data structures with varying length.

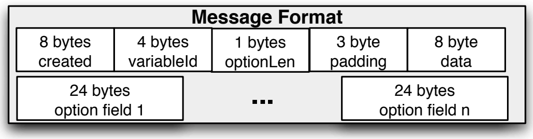


Figure 2.3: The transport message format

The framework has three built-in datatypes that are used without the involvement of the IDL compiler. These types are *int*, *float*, and *boolean* as defined in the Java language specification. The transport message format is able to carry any of these datatypes without the need to use any of the optional fields. Any user-defined datatypes must be specified using the IDL language. The marshal and unmarshall code is then produced by the IDL compiler.

Every datatype, either built-in or user-defined, has two implicit data fields: *variableId* and *timestamp*. The variableId is an 8 byte unique identifier for all the application variables that are utilizing the middleware framework. The variableId is created by the management plane and returned to the publisher when it registers a publication. The subscribers subscribe to a variable (giving the publisher name and variable name) and if the request is accepted, the variableId is returned to the subscriber. The timestamp is either set by the middleware framework by using the publisher's local clock or the publisher explicitly sets the timestamp for each event.

2.1.4 Publisher Interaction Model

The publisher API provides only a push interface, the publishers are the active entities by pushing events at the interval specified during publication registration. The framework takes control of the application level thread (blocking call) until the event is forwarded to the edge status router. The framework marshalls the event and produces an event message, which is then sent out as a packet on the underlying communication layer. The control is then returned back to the publisher with a return code whether the event was sent successfully.

The lifecycle of a publisher in the GridStat framework is as follows: first, it registers the variable and the interval that events are produced by the variable. The registration must be accepted by the management plane before the publisher starts pushing events into the system at the registered interval. The publisher should unregister the publication before it stops pushing events into the framework. A request to unregister a variable is always accepted by the management plane. After unregistering the variable, the publisher stops pushing events from the variable. If the publisher wants to start publishing events from the variable again, it has to reregister the variable.

2.1.5 Subscriber Interaction Models

The subscriber API provides both push and pull interfaces. It is possible for the subscriber to use the push interface for some of the variables that it subscribes to and the pull interface for the remaining variables. In addition to the API for the normal operation of the subscriber, the framework also provides a separate push API that the subscriber uses to register a callback object that is invoked whenever the timeliness requirement for the subscription is violated. These three models are listed below in some more detail:

- **Pull (read latest value when needed):** The subscriber creates an object of the type of the variable that it subscribed to. This object is provided to the framework during subscription. A new value is written into the object when an event is received. The subscriber reads the value from the object whenever it is needed in the application logic. An example application could be one that displays information about the state of an electric power grid. For some of the information it is only displayed when a user clicks on the visual representation of the real world device. So when the user clicks the latest value is read from the cache.

- **Push (signaled when event arrives):** In addition to the object of the subscribed variable type, the subscriber also creates a signal object. Both of these objects are provided to the framework during subscription. When an event is received, the framework uses the signal object to notify the subscriber that an event has been received. An example application could be one that performs control algorithms of the current state of an electric power grid, so whenever a new value from one of the variables is received the state has to be reevaluated by executing the algorithm with the new value. Instead of having the control application sit in a "busy loop" and wait for new values, it is initiated when the values arrives.

- **QoS Push (signaled when QoS is violated):** This style may be added to both subscription interaction models above. The subscriber provides a signal object to the framework during subscription. If the QoS that the subscriber requested during subscription is violated and the end-point middleware detects it, then this object is signaled. An example application could be the same as presented for the Push style. In that the application is only initiated when a new value is received, the problem with this is, what if this value is never received? The application can then register a call back object that will handle the case of what to do in the case that the QoS requested is violated.

The lifecycle of a subscriber starts with registering itself with the management plane. Once accepted, the subscriber can subscribe to status variables by providing the publisher name, variable name, and the QoS desired. The QoS currently supported is redundancy (number of disjoint paths), timeliness (delay starting from the time that the event is published to the time it is delivered), and interval (how often the subscriber wants to receive a new event from the publisher). The subscriber also gives the interaction model that it wants to use depending on which part of the API it uses.

For some control applications, it is more important that they collect a set of datapoints all with the same timestamp (take a snapshot of the state) than to minimize the latency of each individual datapoint. The API, provides for subscribing to such a set of temporally related status variables and this set can be updated at runtime. The assumption is that the publishers of the datapoints publish events at approximately the same time (synchronizing through GPS timestamps). Along with the variables that the subscriber wants to subscribe to, the subscriber also gives the number of events it wants to store for each variable, the *history*. The history feature allows the subscriber to view the past values in the set of variables. When the subscriber is ready to use the values of the variables in the set, it does an atomic read operation that returns a list of all the values in the set. The values will all have timestamps within the smallest publishing interval for any of the variables in the set. If the subscriber used the push interaction model when subscribing to the set of variables, then the signal object is signaled when events have been received from all the variables in the set with approximately the same timestamp.

2.2 Baseline Mechanisms

The baseline set of mechanisms that the GridStat framework provides, is divided into two categories: mechanisms that enhance resilience and mechanisms that enhance efficiency. The mechanisms to enhance efficiency are designed to save bandwidth and computation time. Efficiency is important for event stream paths optimization due to the fact that performance is critical. In addition to this, it is of paramount importance that a system that controls a critical infrastructure can operational under all conditions; predicted and unpredicted. GridStat provides mechanisms that make the system resilient, especially when it is operating close to its limits or in the presence of failures.

Resilience mechanisms include failure recovery mechanisms, extrapolation function mechanism, and limited flooding mechanism. Efficiency mechanisms include the multicast rate-filtering routing mechanism and the condensation function mechanism. The operating modes mechanism adds flexibility to the application layer and, depending on its usage, offers both resilience and efficiency as well as situational adaptability with respect to both the infrastructure and the application layer.

2.2.1 Failure Recovery: Mechanisms to Enhance Resilience

These mechanisms detect communication link and status routers failures. They also provide recovery schemes. Status routers have the capability to recover from such failures with the help of the leaf QoS broker.

Communication Link Failure

When a event channel fails, the status router that detects the failure marks it as a failed event channel and informs its leaf QoS broker with a command message. The status router tries to reconnect the failed event channel at a regular interval. All status events that are destined for this event channel are dropped. As soon as the event channel is reconnected, the status router sends a command message informing the leaf QoS broker that the event channel is operational again.

Status events that are destined for this event channel will again flow through it. A special case of communication link failure is when a link between a leaf QoS broker and one of its status routers fails. The status router tries to reconnect to its leaf QoS broker at regular intervals, while it continues its normal operation i.e., routing status events that it receives. When the link is re-established the leaf QoS broker sends to the status router updates, if any, that have occurred while the status router was disconnected.

Status Router Failure

When a leaf QoS broker detects that one of the status routers in its cloud has failed, the status router is marked as unavailable. There are two possibilities why it failed: either the communication link to the status router failed or the status router itself failed. The failed status router informs the leaf QoS broker of the reason upon reconnection to the leaf QoS broker. The case of a link failure was explained above in Section 2.2.1. If the status router failed, it restarts in a clean state. The leaf QoS broker restores the state of the status router by issuing commands to establish communication links to other status routers and to re-initialize its routing table. Each leaf QoS broker maintains enough information to restore the state of all the status routers in its cloud.

2.2.2 Extrapolation Function: A Mechanism to Enhance Resilience

The extrapolation function mechanism is an example of the extensibility of the framework by providing application level logic as a middleware mechanism at the end-points. Consider the case where a status event is missed due to some intermittent event channel failure. The extrapolation function extrapolates a missed value for the given event stream rather than causing a QoS violation. The activation of the extrapolation function is fairly simple. The user specifies with its subscription request whether or not an extrapolation function is to supply missed values instead of causing a QoS callback, the extrapolation function to be used and the number of status events (datapoints) to be used during extrapolation.

Each subscription has a holder object associated with it. This holder object has methods for reading and setting the latest value that has been received from the status variable. The holder object also has a signal object that is signaled, if the timeliness QoS is violated, but if no signal object is registered with the subscription, then this is a “null” object. If an extrapolation function is used, the holder object is extended in two directions. First, the method that sets the latest value does not overwrite the latest value with the newest one but it rather stores it to be used as one of the datapoints. Second, if the subscriber registered to be signaled when a QoS violation occurs then, instead of signaling the subscriber, the specified extrapolation function is called. The result from the extrapolation function is stored as the latest value that has been “received”. At any time, the extrapolation function mechanism is aware of whether the latest value is extrapolated or not.

The functionality of the various extrapolation functions is not the purpose of this discussion, but the extensibility of the framework is. A number of well-known extrapolation functions is implemented to show the extensibility of the framework, but there is no limitation on implementing other functions as well. GridStat provides the following built-in extrapolation functions:

- **Average:** It calculates the average of the past datapoints and predicts this average as the missed value.
- **Stoer Bullirsh:** It predicts the missed value by using the Stoer Bullirsh extrapolation function [61, 12].
- **Polynomial:** It extrapolates a value by using the polynomial extrapolation function [42] where the polynomial’s order is set by the user.
- **Average over a number of orders for Polynomial:** It predicts a missed value by using multiple polynomial extrapolation functions of different orders and averages the result over those functions.

User-defined extrapolation functions are also possible to be implemented by having the interested user extend an abstract base class. Other end-point mechanisms may easily be added to the framework, even application domain specific. End-point mechanisms, such as extrapolation functions help with reusability and ease of programming logic.

2.2.3 Multicast Rate-filtering Routing: A Mechanism to Enhance Efficiency and Predictability

The multicast rate-filtering routing mechanism are embedded in the status routers routing process. The filtering mechanism filters unnecessary events from event streams, so that only events at the interval that the subscriber has specified are delivered. If there are multiple subscribers for the same event stream, multicasting is enabled by forwarding only one copy of the event. The filtering and multicast mechanisms combined together provide for predictable filtering for temporally related variables. Suppose two publishers (Pub p_0 and p_1) publish events from one variable each at an interval of 10ms and there are two subscribers (Sub s_0 and s_1) that would like to receive events from both variables every 20ms and 30ms, respectively (see Figure 2.4). Assuming the starting time is set to 0ms, the status router SR e_0 forwards an event at time points 20ms, 30ms, 40ms, and at time 60ms only one event is sent by utilizing the multicast mechanism.

The multicast rate-filtering routing mechanism allows for distributed snapshot of variables that are published at the same time (using GPS) and at the same interval or at a multiple of that interval. In addition, different subscribers are able to take the snapshot at different intervals. This is a requirement for control systems that the forwarding of events provides for the ability to receive a set of events taken at the same time [45]. The simple approach of providing slightly better QoS to the second subscriber in the example above, would not work because the subscriber would receive events from some of the variables at interval 10ms, 20ms, and 30ms. The variables that are received at an interval of 20ms will not align with the 30ms interval that the subscriber needs to get the snapshot of the system state. The GridStat framework therefore provides for deterministic filtering with multicast for all variables to support for subscription to temporally related variables.

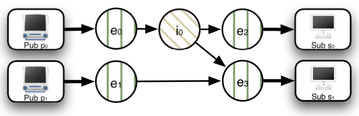


Figure 2.4: Multicast rate-filtering routing scenario

2.2.4 Limited Flooding: A Mechanism to Enhance Resilience

The limited flooding mechanism distributes events from specific alert variables to a limited region. A region consists of a single or multiple clouds, but not selective parts of clouds. The flooded region is specified by the number of levels in the management hierarchy that have to be covered, with level 1 being the leaf QoS broker, level 2 its parent QoS broker, and so on. For instance, if a status variable is set to be flooded to level 3 in the system of Figure 2.1, then events from the variable are distributed to the entire system, since it only has 3 levels. However, if a variable registered in the leftmost cloud of Figure 2.1 is specified to be flooded to level 2, then the leftmost and middle clouds receive the events. A subscriber in the rightmost cloud does not receive the event through flooding, but has to subscribe to it. The decision of when an variable is flooded, and to which level, is determined by QoS management policies.

The purpose of the flooding mechanisms is twofold. First, it minimizes the end-to-end latency of certain events by bypassing the routing mechanisms and spreading the events to all the subscribers in a limited area. Second, flooding allows subscribers to receive all flooded events instead of explicitly subscribing to each one of them. The flooded variables, depending on the policy, may reflect something special about the condition of the application and, as a result, it is possible that the set of these variables is updated over time or as the condition of the application changes. The subscribers that subscribe to all the flooded events simply log the occurrences of the event or analyze the condition of the system.

2.2.5 Operating Modes: A Mechanism to Enhance Efficiency, Resilience and Situational Adaptability

The operating mode mechanism provides for adapting to an appropriate predefined mode depending on the application or middleware infrastructure conditions. Each mode consists of a predefined set of subscriptions. During subscription, the subscriber specifies the modes the subscription operates in. A subscription path is allocated for each of the modes requested. The routing information for the modes is available to the status routers. The system always operates in one mode and the change from one mode to another is controlled by QoS management policies. There are two requirements that govern mode adaptability. The first requirement is that during mode change the resources available are not exhausted which would otherwise result in dropping of events. The second requirement is that a mode change is transparent to a subscription valid in both the current and new mode, i.e. the event stream is not interrupted.

The benefit of the mode change mechanism is that it provides for a quick way of adapting to a new set of already allocated subscriptions, whenever operating conditions change at the application or middleware layers. For example, consider an application that monitors a system that operates differently depending on the season of the year. A subscription set for each of the different seasons is predefined. Another example involves applications that predefine a set of contingency operations for crisis situations. Whenever a situation is identified as a crisis scenario, the system quickly adapts to the contingency experienced. A final usage is observed when the middleware framework experiences difficulties and as a result the application must adapt accordingly. Such a difficulty is a event channel failure.

2.2.6 Condensation Function: A Mechanism to Enhance Efficiency

A condensation function is an example of migrating application logic into the middleware. In doing so, multiple benefits are observed. The application architect is aware of the tradeoffs of migrating application logic to the middleware layer and, as a result, he/she can decide what part of the specific application may benefit from using this mechanism. The condensation function is seen as a hybrid of mobile code, reusable results, resource conservation, and simplification of application logic.

In a way, the condensation function mechanism concept was realized as an alternative to having multiple subscribers performing locally the same calculation on a given set of status variables. Each of these subscribers requests subscription paths to all the status variables and manipulates, even correlates, the corresponding status events locally. This redundant use of resources is limited if the application architect predicts a common set of calculations that are useful for a number of subscribers. A condensation function carries out the necessary operations on a status event set and publishes the result as a status event of a newly created variable.

The application architect specifies the input variables, the produced status variable and the application logic to be performed using a GUI based tool that is developed for the GridStat framework. The condensation function tool then forwards the specified information to the management plane which verifies that the information provided is valid and locates an appropriate status router to load the condensation function on. The management plane then sends the command to the status router where the condensation function will be loaded with the necessary information for that status router to create a condensation function that will perform the specified application logic. Finally, the status router reports back to the management plane if the creation of the condensation function was a success or not which is then relayed back to the condensation function tool.

CHAPTER 3

DESIGN OF THE GRIDSTAT ENTITIES

The GridStat framework consists of seven entities: publisher, subscriber, *condensation creator*, edge status router, status router, leaf QoS broker, and QoS broker. These seven entities are categorized as data plane entities (publisher, subscriber, condensation creator, edge status router, status router) and management plane entities (leaf QoS broker, QoS broker). The entities in the data plane are further subdivided into end-entities (publisher, subscriber, condensation creator) and status routers (edge status router and status router).

Figure 3.1 illustrates the high level design of the entities and their interactions. There are two types of interaction within the framework. The first is the command interaction, where one entity issues commands to another entity to perform a certain task or inform that entity about changes to its state. The second interaction is the forwarding of status events over event channels from an entity to one of its adjacent neighbor entities. The publishers, subscribers, edge status routers, and status routers use both interaction types, whereas the condensation creator, leaf QoS broker, and QoS broker only use the command interaction. The command interface that the different entities provide are specified in CORBA idl files (see Figures A.1 – A.7 in Appendix A). The publisher and subscriber command interaction has also been ported to use Microsoft's .NET instead of CORBA.

The event channels are communication links that send and receive big-endian byte-packets and the message format is illustrated in Figure 2.3. The current implementation uses the UDP protocol as its transport layer. In order to implement one of the entities in a different programing language, the language must be supported by CORBA and be able to communicate using UDP sockets. The exception is if a publisher or a subscriber application is developed then Microsoft's .NET can be used instead of CORBA.

The remainder of this chapter presents the rationale and design of the different entities of the GridStat framework. Common design issues for the entities are presented in Section 3.1. The

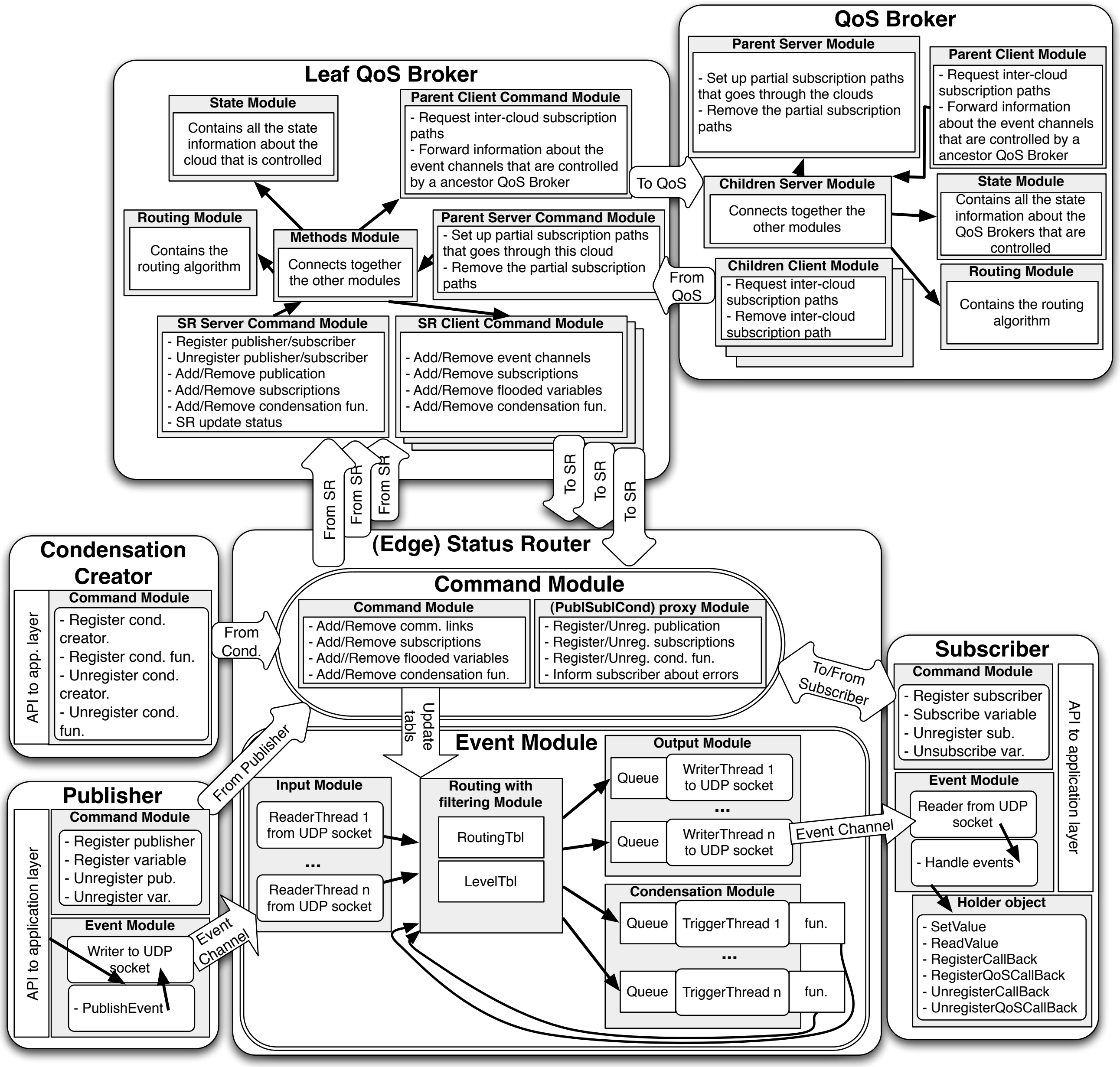

Leaf QoS Broker
State Module
Contains all the state information about the cloud that is controlled
Parent Client Command Module
- Request inter-cloud subscription paths
- Forward information about the event channels that are controlled by a ancestor QoS Broker
Routing Module
Contains the routing algorithm
Methods Module
Connects together the other modules
Parent Server Command Module
- Set up partial subscription paths that goes through this cloud
- Remove the partial subscription paths
SR Server Command Module
- Register publisher/subscriber
- Unregister publisher/subscriber
- Add/Remove publication
- Add/Remove subscriptions
- Add/Remove condensation fun.
- SR update status
SR Client Command Module
- Add/Remove event channels
- Add/Remove subscriptions
- Add/Remove flooded variables
- Add/Remove condensation fun.
To QoS
From QoS
QoS Broker
Parent Server Module
- Set up partial subscription paths that goes through the clouds
- Remove the partial subscription paths
Parent Client Module
- Request inter-cloud subscription paths
- Forward information about the event channels that are controlled by a ancestor QoS Broker
Children Server Module
Connects together the other modules
State Module
Contains all the state information about the QoS Brokers that are controlled
Children Client Module
- Request inter-cloud subscription paths
- Remove inter-cloud subscription path
Routing Module
Contains the routing algorithm
From SR
From SR
From SR
To SR
To SR
To SR
Condensation Creator
API to app. layer
Command Module
- Register cond. creator.
- Register cond. fun.
- Unregister cond. creator.
- Unregister cond. fun.
From Cond.
(Edge) Status Router
Command Module
Command Module
- Add/Remove comm. links
- Add/Remove subscriptions
- Add//Remove flooded variables
- Add/Remove condensation fun.
(PublSublCond) proxy Module
- Register/Unreg. publication
- Register/Unreg. subscriptions
- Register/Unreg. cond. fun.
- Inform subscriber about errors
To/From Subscriber
Update tabls
Event Module
From Publisher
Input Module
ReaderThread 1 from UDP socket
...
ReaderThread n from UDP socket
Routing with filtering Module
RoutingTbl
LevelTbl
Output Module
Queue
WriterThread 1 to UDP socket
...
Queue
WriterThread n to UDP socket
Condensation Module
Queue
TriggerThread 1
fun.
...
Queue
TriggerThread n
fun.
Event Channel
Publisher
API to application layer
Command Module
- Register publisher
- Register variable
- Unregister pub.
- Unregister var.
Event Module
Writer to UDP socket
- PublishEvent
Event Channel
Subscriber
Command Module
- Register subscriber
- Subscribe variable
- Unregister sub.
- Unsubscribe var.
API to application layer
Event Module
Reader from UDP socket
- Handle events
Holder object
- SetValue
- ReadValue
- RegisterCallBack
- RegisterQoSCallBack
- UnregisterCallBack
- UnregisterQoSCallBack

Figure 3.1: High level design of GridStat entities and their interactions

entities in the data plane group are presented next in Section 3.2, followed by the management plane entities in Section 3.3.

3.1 Common Design Issues

The common design issues are all related to the discovery method used by entities to locate each other and the method used to establish the event channels in order to create a network that disseminates status information. The following subsections explain in detail the "bootstrapped" technique, the common components of the command interaction, and the common components of the event channels.

3.1.1 Bootstrapping the Framework

The GridStat entities use a CORBA naming service to locate each other at startup. The location of the naming service is available to the entities in two ways: either the necessary information to locate the naming service is given as command line argument or the information is stored in an XML file named `initNaming.xml` in the `config/` directory, one level down from the directory where the application was launched. The entities start a servant object that implements its interface for the commands that it accepts and then register this servant object with the naming service. Whenever an entity needs to use the command interface of another entity, it contacts the naming service to get a reference to the appropriate servant object. The reference is narrowed down to a local object that is used to issue commands to the remote object.

Entities use the command interfaces to exchange the necessary information to establish the event channels. Each leaf QoS broker has complete knowledge of the resources in the cloud that it controls, including the event channels that the (edge) status routers have. The initial knowledge that a (leaf) QoS broker has is stored in an initialization file that conforms to the Document Type Definition (DTD) presented in Figure A.9. At run-time, additional information could be added or existing information could be removed. When a (edge) status router contacts its leaf QoS broker, the leaf QoS broker informs the (edge) status router to open the appropriate ports on the appropriate network interfaces for the event channels that the (edge) status router will have. The publishers are informed through the command interfaces of which port to connect to when they initially connect to their edge status router. On the contrary, a subscriber first opens a UDP port on one of its network interfaces and when it connects to its edge status router the subscriber gives the port information to the edge status router in order to establish an event channel to it.

3.1.2 Command Module in the Management Plane

The management plane only has a command module, since its entities do not handle status events directly; they only manage the allocation of status event flows in the data plane. Leaf QoS brokers and QoS brokers share some common components. Each entity (except the root QoS broker) in the management plane has a client component that is used for connecting to its parent QoS broker. In addition, a QoS broker has a client component to connect to each of its children (leaf) QoS brokers. The leaf QoS broker has a server component that serves requests from its parent QoS broker. The QoS broker has two server components, one to serve requests from its parent (except for the root) and one to serve requests from the children (leaf) QoS brokers.

Leaf QoS brokers and QoS brokers share the same client component whereas their server components are different. A client component provides two functionalities: locating and resolving object references (CORBA servants) from the naming service and providing stubs for the remote servant. The client component that a (leaf) QoS broker has to communicate with its parent is named `ClientToParent` and the client component that a QoS broker uses to communicate with its children is named either `ClientToQoSChild` or `ClientToLeafQoSChild` (see Figure 3.2). These components inherit from the class `Client`, which is responsible for connecting with the naming service and acquiring references to servant objects. The references to the servant object are then narrowed down to local stub objects of the remote objects. The client stub objects are `HierarchyQoSParent`, `HierarchyQoSChild`, and `HierarchyLeafQoSChild` which are generated from the IDL file in Figure A.7. The local stub is used by other components within the entities to call remote methods on the servant objects.

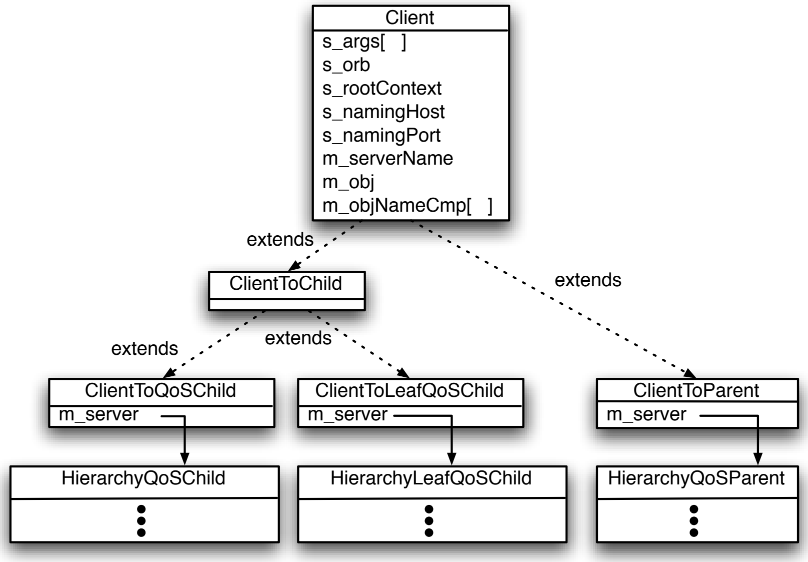


Figure 3.2: Client component of the management plane command module

3.1.3 Command Module in the Data Plane

The entities in the data plane as well as the leaf QoS broker (for its interaction with the data plane) have a command module. All the command modules have a client component and some of them have a server component as well. The client component is used by an entity to call various methods on another entity, while the server component is used to serve remote calls issued by other entities. The publisher and condensation creator do not have a server component in their command module because they don't serve any requests.

Client Component of the Command Module

The client component of the command module provides two functionalities: locating and resolving object references (CORBA servants) from the naming service and providing stubs for the remote servant. Figure 3.3 presents the different client components in the command modules for the data plane entities. The class `Client` provides the functionality of connecting with the naming service and acquiring references to servant objects. The classes corresponding to entities start with

CommandClient and end with the name of the entity. Each class is responsible for narrowing down the servant object reference to the local stub (generated from the IDL files in Figures A.3 and A.4) of the remote object. The local stub is used by other components within the entities to call remote methods on the servant objects.

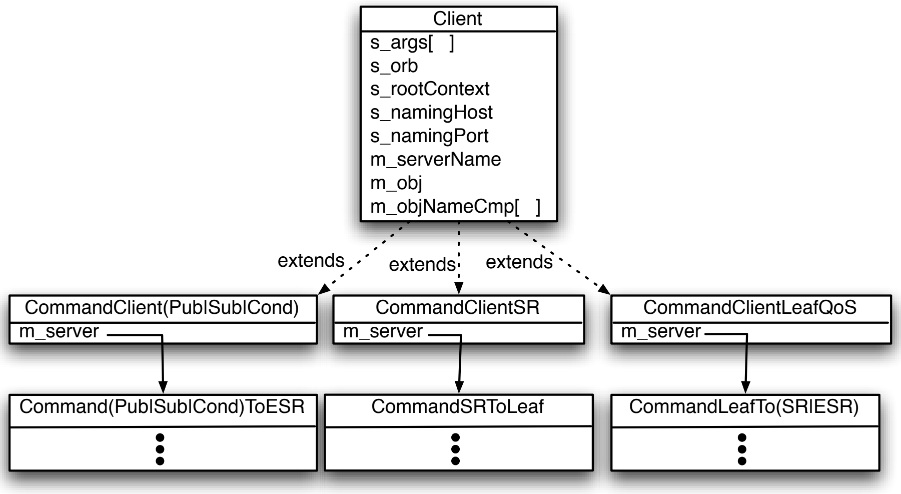


Figure 3.3: Client component of the command module

Server Component of the Command Module

The server component of the command module provides two functionalities: locating and registering of object references with the naming service and providing servant objects that serve remote requests by other entities. Figure 3.4 presents the different server components in the command modules for the entities in the data plane. The class Server provides the functionality of connecting with the naming service and registering of object references for the servant objects. The class that starts with Server and ends with the name of an entity is responsible for instantiating the servant object for a specific entity (generated from the IDL files in Figures A.3 and A.4) and serving the incoming requests. The servant objects forward requests to other components within the entity depending on the functionality of the specific method that is called.

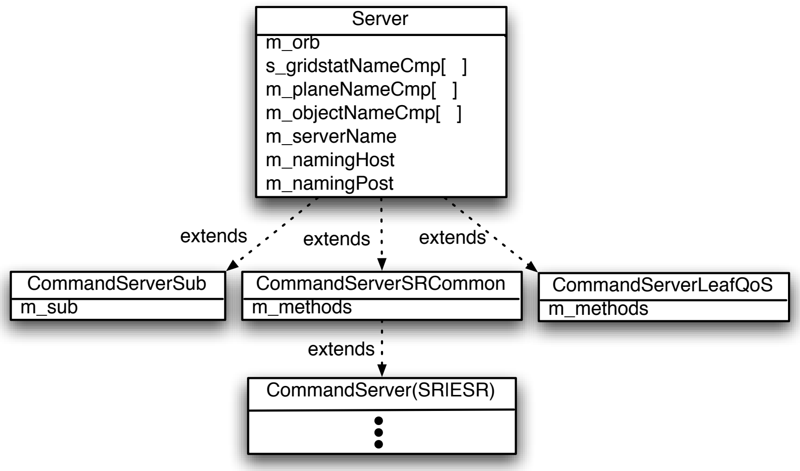


Figure 3.4: Server component of the command module

3.1.4 Event Channel Module

The event channel module is only present in the following data plane entities: publishers, subscribers, and (edge) status routers. The functionality of this module is to send and receive status events from a communication link. The base class for this module, which is used by all entities, is the `EventChannel` class (see Figure 3.5). This class provides the functionality to open a UDP socket that the entity will listen on to receive datagrams and to connect to a peer UDP socket. Entities send ping messages (at a specified interval) to each other using the event channel. Each end of an event channel keeps track of the last ping message that was received. This information is used by other modules to detect if the event channel is operational. The different entities extend this class (`EventChannel(Pub|Sub|SR)`) and provide additional functionality that the specific entity requires for the event channel (explained in the subsections for the specific entities later on).

3.1.5 Failure Model for a Failed Publisher or Subscriber

A publisher or subscriber could fail in two ways: the connection to its edge status router is lost, or it just terminates unexpectedly (crash). In the case of a crash, the edge status router is not able to determine if the end-entity crashed or if the connection was lost. Therefore, the edge status router

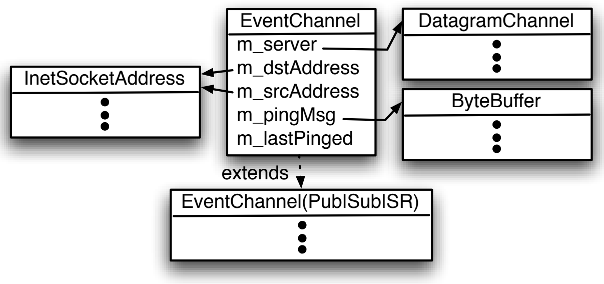


Figure 3.5: Status event module

(and the leaf QoS broker) maintains its current state about the end-entity until it is explicitly told otherwise. When the end-entity restarts, it registers itself again with the edge status router. Due to the fact that the edge status router maintained state information about the end-entity, it can deduce that it was not a communication failure. The edge status router informs the leaf QoS broker that the specified end-entity reconnected in a clean state. If the end-entity was a publisher, then all the publications are unregistered and similarly, in the case of a subscriber all its subscriptions are unregistered. The edge status router sets up a new event channel for the end-entity and informs the end-entity that it has failed and restarted. The end-entity can now start to register publications and subscriptions.

In the case that the edge status router terminates unexpectedly, publishers and subscribers will detect it, but can not deduce if it's a communication failure or an edge status router failure. The end-entities try to reconnect to their edge status router at a specified interval. When the edge status router restarts, the end-entities re-establish connection and try to reregister themselves. For each reregistration, the edge status router contacts its leaf QoS broker to verify that the end-entities are legitimate end-entities (e.g. they were connected to the particular edge status router before it failed.) The edge status router informs the end-entities that it failed and restarted during the reregistration phase.

The design decision of maintaining the state of publishers, publications, subscribers, and subscriptions in both edge status routers and leaf QoS brokers until the end-entities explicitly unregister themselves was made based on the application domain that the framework is envisioned to be used for. A different approach would be to have the end-entities reregister themselves as well as their current publications (subscriptions) that they are currently providing (requesting) at a specified interval. If the end-entity fails to reregister for a specified number of epochs then the state is removed and the end-entity has to register itself as if it were the first time. The reason that this approach was not chosen for this framework is the assumption that the application domain is static i.e. the variables published and the requested subscriptions do not change much. In this case, the overhead of reregistering the publications and subscriptions is quite large. Due to the static nature of the application domain an unexpected termination of an end-entity would result in the entity being restarted and register themselves with the framework.

3.2 Data Plane Entities

The data plane consists of the following entities: publisher, subscriber, condensation creator and (edge) status routers. These are explained in the subsequent subsections.

3.2.1 Publisher

The publisher is the part of the framework that provides the abstraction of publishing status event streams of status variables. The publisher has a client command module, an event channel module and the API that is provided to the publisher application. The client command module and the common components of the event channel module were presented in Section 3.1. This section describes them in more detail. The rationale for the publisher is presented first, followed by the design of the above modules and the interactions among them. Finally, the failure model of the publisher is described.

Rationale for the Publisher

The rationale for the API that the publisher provides to the application layer is simplicity; it should be easy for a software developer to implement a publisher application for the framework. It is assumed that the majority of the devices that act as publishers are "dumb" devices with limited computational powers, many in dedicated hardware. Their roles are merely to measure different properties of physical devices and publish these measurements as status events to interested parties. The publisher does not perform any filtering, but instead it forwards all generated status events. The API that the publisher provides to the application layer is presented in Figures A.10 – A.16.

In addition to the status events, the publisher API also provides for the publication of *derived values*. Examples of derived values include the moving average, maximum value during the last x interval or minimum value during the last x interval. This mechanism differs from the condensation function mechanism (see Chapter 7) in that the former mechanism only provides derived values for a single status variable, whereas the latter can provide derived values from *1 - n* status variables.

The motivation behind the derived value mechanism was to provide the application architect a tool for moving application logic into the middleware. Moving application logic into the middleware is beneficial in three ways. First, the application logic at the subscriber can be simplified, i.e. if the average is needed, it can be just retrieved from the variable holder object. Second, computational resources can be saved, i.e. one publisher performs computation that multiple subscribers could potentially use. Finally, resources in the data plane could be saved, i.e. if the subscribers are only interested in the average over an x interval, then only one event is sent every x time interval.

However, the use of derived values also consumes resources. Every event that is published with derived values is encapsulated into larger packets (with respect to number of bytes.) The application architect has to evaluate if the benefits of providing derived values outweighs the extra packet overhead; this depends on the application requirements and usage of different variables by subscribers. The application architecture can combine the flexibility of the derived value and condensation function mechanisms. For example, assume that the variables that are used as inputs to a condensation function provide the average derived value. If the logic of the condensation function is to provide the average of all its input variables, then it can subscribe to its input variables at a larger interval because it can use the average derived value of its input variables and combine these to calculate the average of all the input variables. Resources are saved on the path from the publisher to the status router where the condensation function is loaded (due to the derived value mechanism). Resource usage is further saved on the path from the status router (where the condensation function is loaded) to the subscribers (due to the condensation function mechanism).

Design of the Publisher

From the high level design (see Figure 3.1) it can be seen that the publisher consists of two parts: the command module and the event channel module. The publisher uses the command interface `CommandPubToESR` (see Figure A.3) to issue commands to the management plane to inform it of its publications, etc. The publisher does not provide its own command interface, i.e. a does not provide a servant object. During the registration phase, a publisher is given information about the event channel (UDP socket) that it will use for its event flows. This event channel is used by the publisher to forward all status events that are generated for its registered status variables.

The design of the different Java classes and their interaction are illustrated in Figure 3.6. The different classes are now overview.

The Publisher API The `Publisher` class (see label 0 in Figure 3.6) implements the `PublisherInterface` presented in Figures A.10, A.11, and A.12. It is the class that connects together all the classes of a publisher and this is the interface that is provided to the application layer. If the publisher application needs to use derived values then the `PublisherWithDV` (see label 1 in Figure 3.6) is used. This class implements the `PublisherWithDVInterface` (see Figure A.13) which is the API for a publisher with the derived value mechanism. From the point of the application developer, he/she will only use one of these two classes (`Publisher`, `PublisherWithDV`) when implementing a publisher application using the GridStat framework.

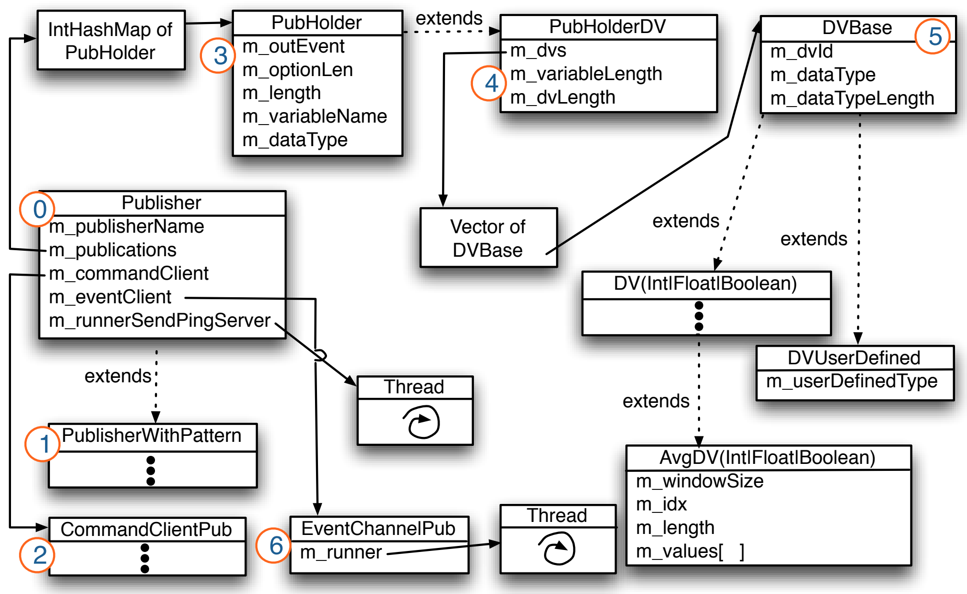
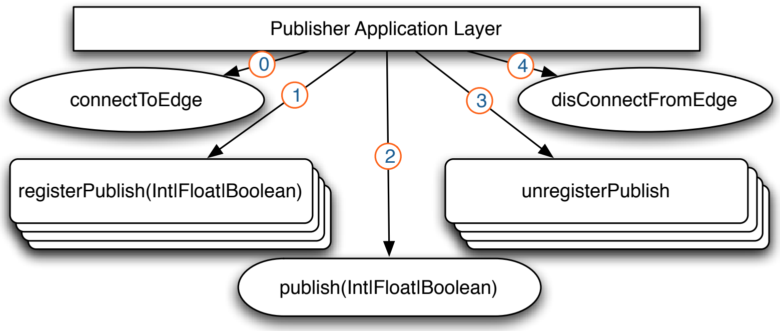


Figure 3.6: Design of the publisher

The `Publisher` class has two constructors and the difference between the two lies in the way that the application becomes aware of the location of the naming service. The first constructor assumes that a file named `initNaming.xml` is placed in the directory `config/` that contains naming service information. The second constructor expects information of the location in the form of two parameters. The lifecycle of a publisher application after the constructor has been called is depicted in Figure 3.7. The publisher first calls the method `connectToEdge` (label 0). This method connects the publisher to its edge, registers the publisher with the management plane, and establishes the event channel to the edge status router. Upon successful connection to the edge status router, the publisher application can register one or more variables using the `registerPublish(Int|Float|Boolean)` methods (label 1.) After a variable is registered, the application should call the appropriate `publish(Int|Float|Boolean)` method (label 2) at the interval that was specified in the registration. When the publisher application no longer needs to publish a variable, the method `unregisterPublish` (label 3) is called. Before the publisher application terminates, it needs to call the method disConnectFromEdge (label 4). This method will unregister the publisher and all its published variables. Note that the application lifecycles of register, publish, and unregister status variables are independent with respect to other variables.



Publisher Application Layer
0
connectToEdge
1
registerPublish(Int|Float|Boolean)
2
publish(Int|Float|Boolean)
3
unregisterPublish
4
disConnectFromEdge

Figure 3.7: The lifecycle of a publisher application

The Publisher Command Module The CommandClientPub class (see label 2 in Figure 3.6) provides the command interaction between a publisher and its edge status router. The common functionality of the client command module was presented in Section 3.1.3. The publisher client stub is generated by the IDL compiler for the interface CommandPubToEST shown in Figure A.3. The connectToEdge method in the Publisher class uses these classes to locate the naming service, get the reference to the edge status router servant, narrow this object reference into the stub and use this stub to register the publisher. The stub is also used during the registration and unregistration of variables. The command module in the publisher will finally be used to unregister the publisher before it calls the disConnectFromEdge method to disconnect from the edge status router.

The Publisher Event Channel Module The event channel module in the publisher consists of two parts: the EventChannelPub (label 6) and PubHolder (label 3) shown in Figure 3.6. If the publisher is using derived values, then the PubHolderDV (label 4) class should be used instead of the PubHolder class. When a variable is registered to be published, an instance of the

`PubHolder` class is created. The publisher places this instance in its lookup table (IntHashMap). The holder for the variable holds a ByteArray and its fixed fields are set. When a new value is published for a specific variable, the value field in the holder is replaced with the new value. The timestamp field is also updated. A reference to the ByteArray is given to the `EventChannelPub`, which then forwards the ByteArray to the edge status router.

In the case that the application registers a variable that contains derived values, an instance of the `PuBHolderWithDV` is created. The derived values that are used for the variable are passed as a list of `DVBase` objects (label 5). The holder creates a ByteArray to hold the events for the variable and the fixed fields are set. The fixed fields for the derived values are also set. When a value is published, the publisher examines each of the derived values, gets the latest derived value and sets the new value for the variable. A reference to the ByteArray is passed to the `EventChannelPub`, which in turn forwards the ByteArray to the edge status router.

Failure Model

Two failures could occur with the publisher: its connection to its edge status router is lost, or the publisher terminates unexpectedly (crash). The unexpected termination case was covered in Section 3.1.5, so only the case of a lost communication connection is discussed here.

The event channel has a built-in mechanism to detect if the connection is lost. The mechanism requires each side of the event channel to send a ping message to its peer at a specified interval. Each side checks if it received a ping message from its peer. There are two possible explanations for losing communication with the edge status router. The first is that the underlying communication network experiences failures, while the second one is that the edge status router has failed. For the first case, the repair is easy because each side of the event channel has maintained its state. For the second case, the publisher will need to reregister itself with the edge status router, which in turn will verify with its leaf QoS broker that this particular publisher is indeed a legitimate publisher. The edge status router will open a new port for the event channel with the publisher and return this information back to the publisher during the reregister phase.

When the publisher detects that the event channel is not operational, it takes the following action: for every published status event, an error code is returned to the application layer. It is left to the application layer how to handle this. The publisher middleware tries to reconnect to the edge status router at regular intervals until connection is successful. Once the event channel is reestablished, a return code to the application indicates that the event was sent successfully.

3.2.2 Subscriber

The subscriber is the part of the framework that provides the abstraction of receiving status event streams of status variables in order for the application layer to use the "latest" value of the status variable. The subscriber has both a client and a server command module, an event channel module and the API that is provided to the subscriber application. The client and server command modules and the common components of the event channel module were presented in Section 3.1. This section describes them in more detail. The rationale for the subscriber is presented first, followed by the design of the above modules and the interactions among them. Finally, the failure model of the subscriber is described.

Rationale for the Subscriber

The rationale for the API that the subscriber provides to the application layer is simplicity and flexibility; it should be easy for an application developer to implement a subscriber application for the framework while at the same time provide multiple interaction models with the subscribed status variables. The API that the subscriber provides to the application layer is presented in Figures A.17 – A.30 and can be used for a wide range of applications; from simple ones that only display (strip charts) or log the values of the subscribed variables to applications that use the values of the status variables as inputs to control algorithms. The results of the computations that a subscriber performs could also get published as new status variable.

Due to the different subscriber application needs, the framework provides two interaction models: pull and push. For the pull interaction, the subscriber application reads the latest value of the status variable like any other variable. For the push interaction, the subscriber application registers a callback object (see Figure A.30) that is being called whenever a new status event is received. In addition to the two interaction models, the subscriber application can also register callback objects (see Figure A.30) if any of the specified QoS properties of a subscription is violated.

The multicast rate-filtering routing mechanism (see Chapter 4) provides for deterministic filtering of temporally related variables. In this case, the subscriber provides for subscription to groups of temporally related status variables (see Figure A.25). This API also gives the application layer the ability to specify the history (number of past value sets) stored at the subscriber. With this functionality a subscriber application can request the last *n* set of values. The rationale for providing this interface is that for some control applications, like in the electric power grid, it is of paramount importance to receive the measurements of all input variables sampled at "exactly" (GPS time-stamped) the same time.

Design of the Subscriber

Figure 3.1 illustrates the subscriber's modules: the command module and the event channel module. The subscriber uses the command interface `CommandSubToESR` (see Figure A.3) to issue commands to the management plane to inform it of its subscriptions, etc. The subscriber provides the command interface `CommandESRToSub` (see Figure A.4) so that the management plane can inform the subscriber about changes to subscriptions or a mode change (See Chapter 6). During the registration phase, a subscriber is given information about the event channel (UDP socket) that it will use for its event flows. This event channel is used by the edge status router to forward the subscribed event streams to the subscriber.

The design of the different Java classes and their interactions are illustrated in Figure 3.8. The different classes are now overview.

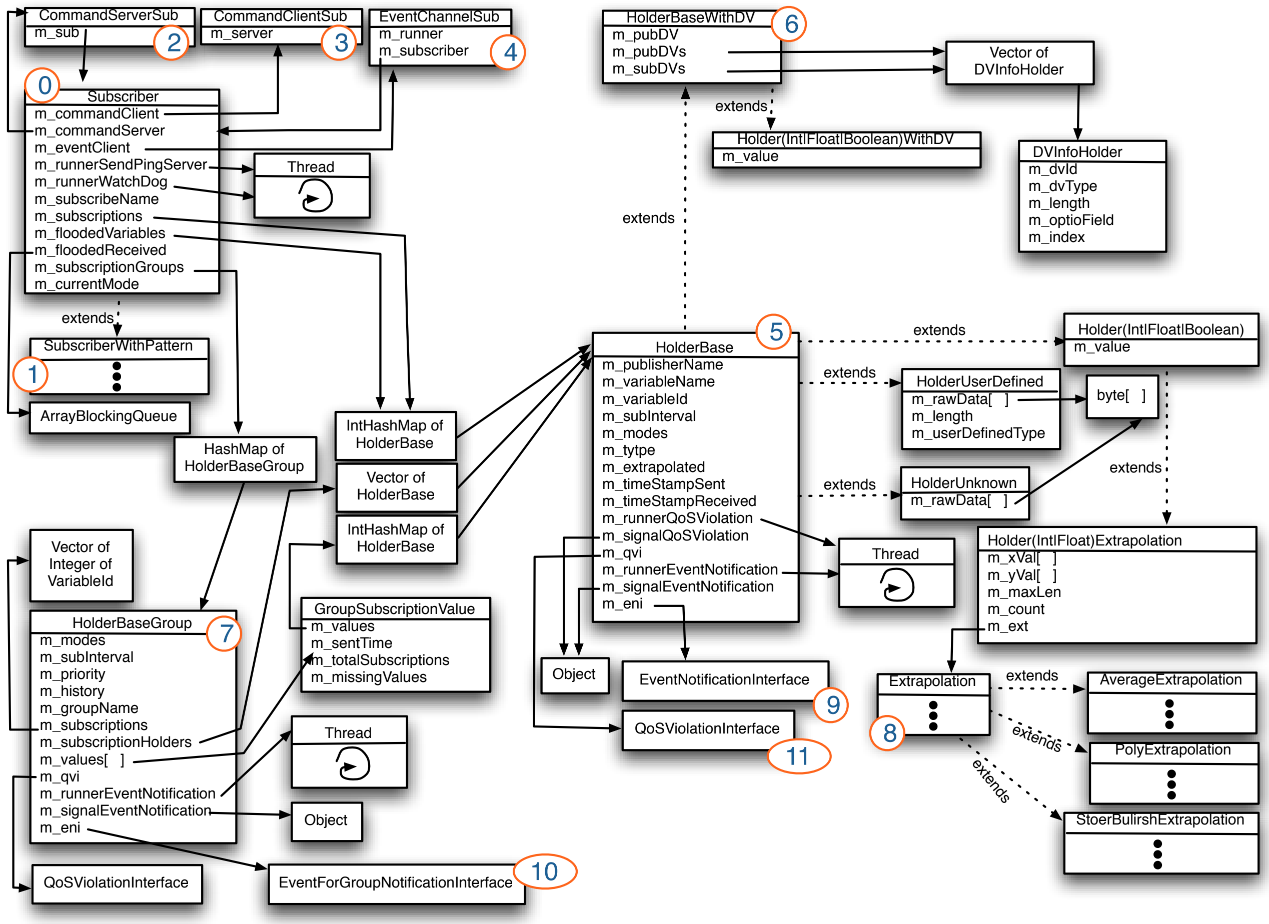


Figure 3.8: Design of the subscriber

The Subscriber API The `Subscriber` class (see label 0 in Figure 3.8) implements the `SubscriberInterface` presented in Figures A.17, A.18, and A.19. It is the class that connects together all the classes of a subscriber and this is the interface that is provided to the application layer. If the subscriber application needs to subscribe to variables with derived values then the `SubscriberWithDV` (see label 1 in Figure 3.8) is used. This class implements the `SubscriberWithDVInterface` (see Figure A.20) which is the API for a subscriber with the derived value mechanism.

For each subscription that a subscriber application wants to register, a holder object is created that is shared between the subscriber application and the framework. The holder objects are all typed objects and the framework performs a type-check to make sure that the holder object matches the type of the status variable that is being subscribed to. There are three different categories of holder objects that are differentiated by the mechanisms used for the specific subscription. The first category is the `HolderBase` class (see label 5 in Figure 3.8) which implements the `HolderBaseInterface` presented in Figures A.21 and A.22. This is an abstract class, but the framework provides for derived classes that hold integer, float, and boolean status variables. In addition, the framework provides a holder for user-defined datatypes; the framework's IDL compiler (see Section 2.1.3) generates the user-specific holder that inherits from the `HolderUserDefinedInterface` (see Figure A.24). The second category of holder objects is a specialization of the first category that also embeds the derived value mechanism. This category uses the `HolderBaseWithPatttern` class (see label 6 in Figure 3.8), which implements the `HolderBaseWithDVInterface` presented in Figure A.26. The third category of holder objects for status variables is the `HolderBaseGroup` class (see label 7 in Figure 3.8) which implements the `HolderBaseGroupInterface` presented in Figure A.25. This holder is used for group subscription and contains a holder for each status variable in the subscription group. If the subscriber application needs to use the extrapolation function mechanism (see Section 2.2.2), then a derived version of the first holder category is used (see label 8 in Figure 3.8). The subscriber application instantiates an object of the type of the status variable that it subscribes to and gives a reference to this object when it registers the subscription with the framework.

The subscriber application interacts with its subscriptions using either *pull* or *push*. For the pull interaction, the subscriber application calls the method `getValue` in the holder object of a specific subscription whenever the latest value of that subscription is needed. In the case of push interaction, the application developer must implement a callback object that implements the `EventNotificationInterface` (see Figure A.30) for subscriptions and `EventForGroupNotification` for group subscription. This callback object is registered with the holder object that is associated with the subscription, using the `registerEventNotificationListener` method. When a new event or a complete set of a group subscription is received, the `eventReceived` method in the callback object is called. The subscriber could also register a QoS violation callback object for a subscription. A QoS violation callback object must implement the `QoSViolationInterface` presented in Figure A.30 and this is registered with the subscription holder object using the method `registerQoSViolationListener`. If a QoS violation occurs, the method `qosViolated` is called in the QoS violation callback object.

An application developer first specifies which subscriber interface to use (`SubscriberInterface` or `SubscriberWithDVInterface`) depending on whether or not the derived value mechanism is to be used. For each subscription, the appropriate holder object must be used: it must match the type of the status variable and provide the mechanisms that the application would like to use. After that, the application developer must decide on the interaction model (pull or push). If push is desired, then the application developer must implement the `EventNotificationInterface`. If pull is desired, then the method `getValue` is used. Finally, if QoS violation callback is desired the application developer must provide callback objects that implement the `QoSViolationInterface`. These are the only interfaces that the application developer needs to interact with in order to write a subscriber application using the

GridStat framework.

The `Subscriber` class has two constructors similar to the publisher class constructors explained earlier. The lifecycle of a subscriber application, after the constructor has been called, is depicted in Figure 3.9. The subscriber first calls the method `connectToEdge` (label 0). This method connects the subscriber to its edge, registers the subscriber with the management plane, and establishes the event channel to the edge status router. Upon successful connection to the edge status router, the subscriber application is ready to register one or more subscriptions using the `subscribe(Int|Float|Boolean)` methods (label 1.) After a subscription is registered, the application uses `getValue` (label 2) if the pull interaction model is used. If push interaction model is used, then `eventReceived` (label 4) is called when a new event is received. If the application has registered a QoS violation callback, then `qosViolated` method is called. When the subscriber application no longer needs a subscription, the method `unSubscribe` (label 5) is called. Before the subscriber application terminates, it calls the method `disConnectFromEdge` (label 6). This method unregisters the subscriber and all its subscriptions (if any). Note that the application lifecycles of `subscribe`, `getValue` or `eventReceived`, and `unSubscribe` are independent of the lifecycle of other subscriptions.

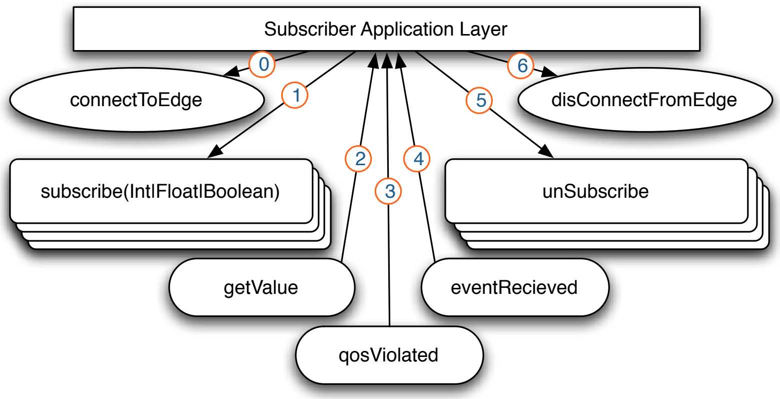


Figure 3.9: The lifecycle of a subscriber application

The Subscriber Command Module The CommandServerSub class (see label 3 in Figure 3.8) provides the command interaction between an edge status router and its subscribers. The common functionality of the server command module was presented in Section 3.1.3. The subscriber provides the CommandESRToSub interface as its servant object for the management plane. The management plane uses this interface to inform the subscriber of any changes to its subscriptions or operating mode changes. The servant object interacts with the Subscriber class to notify the holder object of a subscription of any changes to the subscription. One such notification occurs when the publisher of the status variable unregisters the publication. In this case the subscriber is informed about it. As far as mode changes is concerned, the servant object interacts with the subsriber class to inform it about the new mode. The subsriber class keeps in its state the current mode as well as a mode change that is in progress.

The CommandClientSub class (see label 3 in Figure 3.8) provides the command interaction between a subscriber and its edge status router. The common functionality of the client command module was presented in Section 3.1.3. The subscriber client stub is generated by the IDL compiler for the interface CommandSubToEST shown in Figure A.3. The connectToEdge method in the Subscriber class uses these classes to locate the naming service, get the reference to the edge status router servant, narrow this object reference into the stub and use this stub to register the subscriber. The stub is also used during the registration and unregistration of subscriptions. The command module in the subscriber is used to unregister the subscriber before it calls the disConnectFromEdge method to disconnect from the edge status router.

The Subscriber Event Channel Module The event channel module in the subscriber consists of two parts: the EventChannelSub (label 4) and the different subscription holder interfaces shown in Figure 3.8. The EventChannelSub receives ByteArrays from its UDP socket. The received messages are forwarded to the Subscriber class, which decodes them. The decoder first records the time that the message was received and then extracts the variable identifier from the message. The variable identifier is used to lookup the subscription table to locate the subscription for this variable id. If there is such a subscription and its holder is of one of the built-in types, then the subscriber extracts the value from the message and sets it in the holder object. If the status variable is of a user-defined type, the subscriber extracts the raw bytes from the message and sets those in the holder object. If the push interaction model is used, then the decoder signals that a new event is received for this subscription.

If the subscriber intends to use derived values, the holder object for such subscription must be derived from the `HolderbaseWithDVInterface`. For each of the derived values that the subscriber wants to have access to, the subscriber has to register the derived value with the holder object for the subscription. As status events are received, the middleware sets the value of the status variable as well as the values of all registered derived values. At any time during the subscription, the subscriber application can choose to remove any of the registered derived values.

Another mechanism available to the subscriber is the extrapolation function mechanism. The subscriber application instantiates either `HolderIntExtrapolation` or `HolderFloatExtrapolation` as the holder object for the subscription. The holder object for the extrapolation takes as one of its constructor argument the extrapolation function to be used. At run time, if the subscriber middleware detects that the timeliness QoS property is violated, it will invoke the extrapolation function in the holder object to extrapolate the missed status event. The application layer could also provide its own extrapolation function by extending the abstract `Extrapolation` class.

As far as group subscription is concerned, the application layer first calls the `createGroupSubscription` method in the `Subscriber` class, which returns a reference to a holder object of type `HolderBaseGroup`. After the group is created, the application layer can register one or more subscriptions to the newly created group. This is done by using the `subscribe(Int|Float|Boolean|UserDefined)ForGroup` methods in the `Subscriber` class. There is no need for the application layer to create holder objects for the individual subscriptions that belong to a group; the middleware does this during the registration of a group subscription. There are also two interaction models (pull and push) that the subscriber application can use to interact with a group subscription. For the pull interaction model the subscriber application will use the interface of the group subscription holder to read the set of status values. For the push interaction model the application layer registers a callback object of type `EventForGroupNotificationInterface` with the group subscription holder object. The application layer can remove individual subscriptions that belong to a group subscription at any time by invoking the `unSubscribeForGroup` method in the `subscriber` class. In order to terminate the group subscription, the subscriber application invokes the `removeGroupSubscription` method in the `subscriber` class. This will also unsubscribe all the individual subscriptions of that group.

Failure Model

Two failures could occur with the subscriber: its connection to its edge status router is lost, or the subscriber terminates unexpectedly (crash). The unexpected termination case was covered in Section 3.1.5, and therefore only the case of a lost communication connection is discussed here.

The event channel has a built-in mechanism to detect if the connection is lost. The mechanism requires each side of the event channel to send a ping message to its peer at a specified interval. Each side checks if it received a ping message from its peer. There are two possible explanations for losing communication with the edge status router. The first is that the underlying communication network experiences failures, while the second one is that the edge status router has failed. For the first case the repair is easy, because each of the sides of the event channel has maintained its state. For the second case, the subscriber will need to reregister itself with the edge status router, which in turn will verify with its leaf QoS broker that this particular subscriber is indeed a legitimate subscriber. The edge status router will open a new port for the event channel with the subscriber and return this information back to the subscriber during the reregister phase.

The subscriber does not take any explicit action to inform the subscriber application that it has lost connection with its edge status router. If the application has registered QoS violation callback objects then those will be invoked if any of the QoS gets violated during the connection failure. The application layer has the option to query the middleware layer to check if it has lost connection with the edge status router by invoking the method `isConnected`. The rationale for not informing the application layer about a connection failure is to keep the required interaction between the application layer and the middleware as simple as possible. If the connection failure is an intermittent and short, then the failure might not affect any of the subscriptions. Any additional information that the middleware could provide regarding the QoS violation is most likely not of interest to the application layer; either it gets its events streams or it doesn't. And when it doesn't get them there is nothing the application layer can do with the additional knowledge.

3.2.3 Condensation Creator

The condensation creator is the part of the framework that provides the abstraction of specifying a condensation function and forwarding it to the management plane. The management plane uses this information to find a location for the condensation function, set up the necessary subscriptions, and issue a command to the (edge) status router, where the condensation function will be placed on, to instantiate the specified condensation function. The GridStat framework has a simple GUI version of condensation creator application built in Java, but others could be built as well. The condensation creator has a client command module and the API provided to the application layer.

The rationale for the condensation creator is presented first, followed by the design of the above modules and the interactions among them. Finally, the failure model of the condensation creator is described.

Rationale for the Condensation Creator

The task of specifying a condensation function is performed by an end-entity and the task of placing and instantiating the condensation function is one of the tasks of the management plane.

The application architect is the one that decides what condensation functions the application needs and where it needs this functionality to the accessible from the application layer. The API for the condensation creator has two features: first, specify a condensation function and issue a command to create it and second, issue a command to remove a condensation function.

Design of the Condensation Creator

According to Figure 3.1, the condensation creator consists of only one part, the command module. The condensation creator uses the command interface `CommandCondToESR` (see Figure A.4) to issue commands to the management plane regarding its condensation functions.

The design of the different Java classes and their interactions are illustrated in Figure 3.10. We will now overview the different classes.

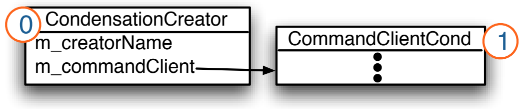


Figure 3.10: Design of the condensation creator

The Condensation Creator API The `CondensationCreator` class (see label 0 in Figure 3.10) implements the `CondensationCreatorInterface` presented in Figures A.31 and A.32. It is the class that connects together all the classes of a condensation creator and this is the interface that is provided to the application layer.

The `CondensationCreator` class has two constructors and the difference between the two lies on the way that the application becomes aware of the location of the naming service. The first constructor assumes that a file named `initNaming.xml` is placed in the directory `config/` that contains naming service information. The second constructor expects information of the location in the form of two parameters.

The lifecycle of a condensation creator application after the constructor has been called is depicted in Figure 3.11. The condensation creator calls the method `connectToEdge` (label 0).

This method connects the condensation creator to its edge status router and registers the condensation creator with the management plane. The condensation creator only needs to register itself with the management plane the first time it interacts with it. This is done by calling the `registerCondCreator` (label 1) method. Once registered, the condensation creator remains registered until explicitly calling the method `unregisterCondCreator` (label 5). As a registered condensation creator, it can register new condensation functions or remove condensation functions that were registered earlier. In order to register a new condensation function, the application must first add one or more subscriptions (label 2) that serve as the input status variables for the condensation function. Then, the application calls `createCondensation` (label 3) which issues a command to the management plane to create the condensation function. In order to remove a condensation function, the application calls the method `removeCondensation` (label 4). It is required that the same condensation creator that created the condensation function is the only one allowed to remove it. Before the application terminates, the method `disConnectFromEdge` is called (label 6). The condensation functions created will still be running until the condensation creator that created them explicitly removes them. In order to remove a condensation creator from the state of the management plane, the condensation creator application calls the method `unregisterCondCreator` (label 5), which removes all condensation functions that were created by this condensation creator, if any, and then removes all information about this condensation creator.

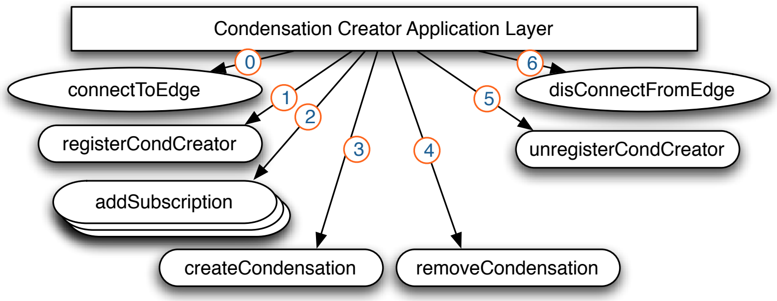


Figure 3.11: The lifecycle of a condensation creator application

The Condensation Creator Command Module The `CommandClientCond` class (see label 1 in Figure 3.10) provides the command interaction between a condensation creator and its edge status router. The common functionality of the client command module was presented in Section 3.1.3. The condensation creator client stub is generated by the IDL compiler for the interface `CommandCondToEST` shown in Figure A.4. The `connectToEdge` method in the `CondensationCreator` class uses these classes to locate the naming service, get the reference to the edge status router servant, narrow this object reference into the stub and use this stub to register the condensation creator. The stub is also used during the registration and removal of condensation functions. When the condensation creator application is done with its session, it calls the `disConnectFromEdge` method to disconnect from the edge status router.

Failure Model

Two failures could occur with the condensation creator: its connection to its edge status router is lost, or the condensation creator terminates unexpectedly (crash). If the condensation creator loses connection to its edge status router, any calls that it issues to its edge status router will fail and return an error code. The application layer retries the command until it either aborts or the connection is reestablished. If the condensation creator application crashes, the application has to reconnect with the edge status router whenever it gets restarted.

3.2.4 Edge Status Router and Status Router

The (edge) status router is the part of the framework that routes status event streams from the publishers to the subscribers. In addition, the edge status router acts as a proxy for the management plane to the end-entities, i.e. it forwards all the commands that the end-entities issue to the management plane. The (edge) status router is transparent to the application layer but interacts with the middleware of the end-entities and its leaf QoS broker. The (edge) status router can be viewed as an extended router that forwards events streams by taking advantage of the semantics of the status variables. The (edge) status router has both a client and a server command module, as well as an event channel module. The client and server command modules and the common components of the event channel module were presented in Section 3.1.

The difference between the edge status router and the status router is that the edge status router acts as a proxy for the command interaction that the end-entities have with the management plane. Additionally, the operating modes mechanism (see Chapter 6) works differently for the two router types. As a result, the command interface that the edge status router and the status router offer to its leaf QoS broker are different for this mechanism. The difference will be explained in the operating modes mechanism chapter.

The rationale for the (edge) status router is presented first, followed by the design of the above modules and the interactions among them. Finally, the failure model of the (edge) status router is described.

Rationale for the (Edge) Status Router

The rationale for the (edge) status router is to provide a message bus between the publishers and subscribers that is specialized to forward status event streams. The (edge) status router has mechanisms to take advantage of the semantic of status variables and these mechanisms are controlled by the management plane. The mechanisms that the (edge) status router provides are: multicast rate-filtering routing mechanism (See Chapter 4), limited flooding mechanism (See Chapter 5), operating modes mechanisms (See Chapter 6), and condensation function mechanisms (See Chapter 7). The management plane uses these mechanisms to provide an efficient, adaptable, and deterministic status dissemination network.

Design of the (Edge) Status Router

From the high level design Figure 3.1 it can be seen that the (edge) status router consists of two parts: the command module and the event channel module. The (edge) status router uses the command interface `CommandSRToLeaf` (see Figure A.4) to issue commands to the management plane and forward commands from the end-entities (edge status routers only). The status router provides the command interface `CommandLeafToSR` (see Figure A.3) and the edge status router provides the command interface `CommandLeafPubSubCondToESR` (see Figure A.4) so that the management plane can control the mechanisms that the (edge) status router has and the end-entities commands can be forwarded to the management plane (edge status router only). When the (edge) status router first connects to its leaf QoS broker, the leaf QoS broker issues commands to the (edge) status router about the event channels that it will have. The (edge) status router sets up these event channels and starts the different threads that handle the events received from these channels.

The design of the different Java classes and how they interact are illustrated in Figure 3.12. The explanation of the different classes and their interactions are presented next.

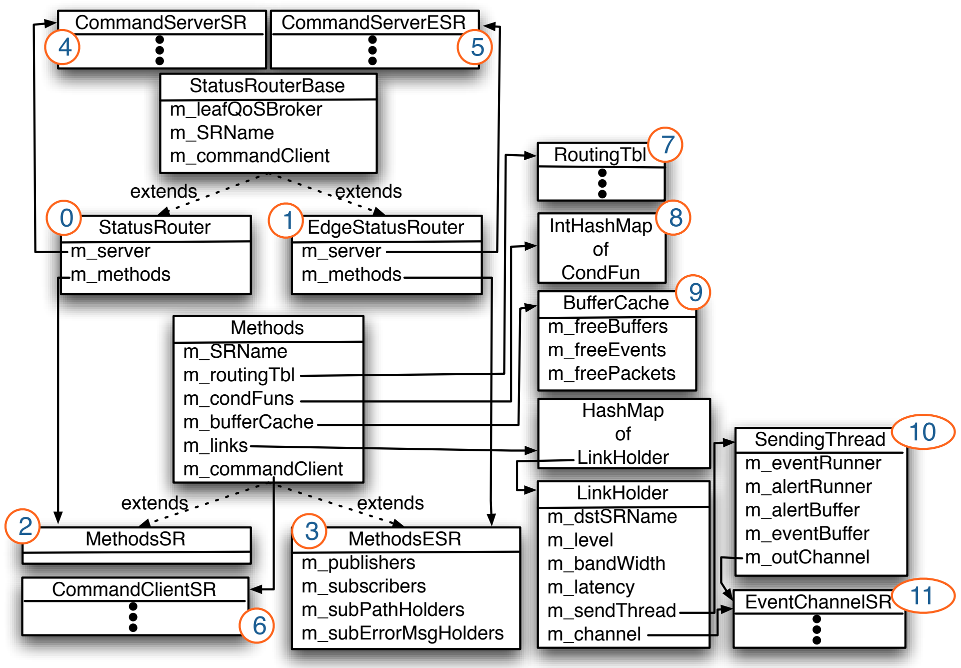


Figure 3.12: Design of the (edge) status router

The (Edge) Status Router Command Module The common functionality of the command module in the data plane was presented in Section 3.1.3.

The `CommandServerSR` class (see label 4 in Figure 3.12) provides the command interaction that the leaf QoS broker has with a status router. The status router provides the `CommandLeafToSR` interface as its servant object for the management plane. The management plane uses this interface to control the mechanisms of a status router. The servant object interacts with the `MethodsSR` class, which again controls the mechanisms of the status router.

The `CommandServerESR` class (see label 5 in Figure 3.12) provides the command interaction that the leaf QoS broker, publishers, subscribers, and condensation creator have with an edge status router. The edge status router provides the `CommandLeafPubSubCondToESR` interface as its servant object for the management plane and end-entities. The management plane uses this interface to control the mechanisms of the edge status router. The servant object interacts with the `MethodsESR` class, which again controls the mechanisms of the edge status router. The `MethodsESR` class in the edge status router keeps also track of the currently connected end-entities.

The `CommandClientSR` class (see label 6 in Figure 3.12) provides the command interaction that a (edge) status router has with its leaf QoS broker. The (edge) status router client stub is generated by the IDL compiler for the interface `CommandSRToLeaf` shown in Figure A.4. The `connectToServer` method in the `CommandClientSR` class uses these classes to find the naming service, get the reference to the leaf QoS broker servant, narrow this object reference into the stub and use this stub to inform the leaf QoS broker about the state of the (edge) status router and its event channels. The stub is used to forward all the commands that the end-entities issue to the management plane via their edge status routers. The status router and edge status router use the same client command module, but the status router doesn't use any of the functionality for forwarding requests on behalf of end-entities.

The (Edge) Status Router Mechanism Module The common mechanisms that both the status router and the edge status router have are now explained. The `RoutingTbl` (label 7) is the mechanism that is used for routing, filtering, and multicast of the event streams. The servant object issues commands to add and remove entries to and from the `RoutingTbl`. The `CondFun` (label 8) contains all the active condensation functions that the (edge) status router has, and the servant object issues commands to instantiate condensation functions and to remove them. The `EventChannelSR` (label 10) contains all the event channels that the (edge) status router has. Again, the servant object issues commands to add or remove these event channels.

The (Edge) Status Router Event Channel Module The event channel module in the (edge) status router consists of four parts: `RoutingTbl` (label 7), `BufferCache` (label 9), `SendingThread` (label 10), and `EventChannelSR` (label 11). The `EventChannelSR` gets a buffer from the `BufferCache` and uses it to read in a packet from the UDP socket. The packet is then sent to the `RoutingTbl`, which places a reference to the buffer into each of the `SendingThread` (outgoing queues) that the status event is to be routed to. The `SendingThread` forwards the event to the `EventChannelSR` which writes the status event to a UDP socket before the buffer is returned to the `BufferCache`.

The `RoutingTbl` organization and the routing algorithm specifics are discussed in detail in Chapter 4.

Failure Model

Three failures could occur with the (edge) status router: its connection to its leaf QoS broker is lost, one of the event channels loses connection, or the (edge) status router terminates unexpectedly (crash). The lost connection to the leaf QoS broker and the failure of one of the event channels were discussed in Section 2.2.1. The recovery of a crash for a (edge) status router was also presented in Section 2.2.1.

3.3 Management Plane Entities

The management plane consists of the following entities: leaf QoS broker and QoS broker. These are explained in the subsequent subsections.

3.3.1 Leaf QoS Broker

The leaf QoS broker is the part of the framework that directly controls all the resources and mechanisms in a cloud and provides functionality to its parent QoS broker to use these resources and mechanisms. The leaf QoS broker is transparent to the application layer; it interacts with the middleware of the end-entities and the (edge) status routers. The leaf QoS broker has two server command modules: one to serve requests from the (edge) status routers and another one to serve requests from its parent QoS broker. It also has two client command modules: one to issue requests to the (edge) status routers that reside in its cloud and another one to issue requests to its parent QoS broker.

The rationale for the leaf QoS broker is presented first, followed by the design of the above modules and the interactions among them. Finally, the failure model of the leaf QoS broker is described.

Design of the Leaf QoS Broker

The leaf QoS broker consists of three major parts: the command module for the (edge) status routers, the command module for the parent QoS broker, and its mechanisms. For the data plane entities, the leaf QoS broker uses the command interfaces `CommandLeafToSR` and `CommandLeafToESR` (see Figure A.3) to issue commands to these entities. It also provides the servant interface `CommandSRToLeaf` for these entities. For the management plane entities, the leaf QoS broker uses the command interface `HierarchyQoSParent` to issue commands to its parent QoS broker and provides the servant interface `HierarchyLeafQoSChild` (see Figure A.7) to its parent QoS broker. Upon initialization, the leaf QoS broker registers with its parent QoS broker unless it is the “root” node of the hierarchy tree.

The design of the different Java classes and their interactions in the two modules are illustrated in Figure 3.13. We will now overview the different classes.

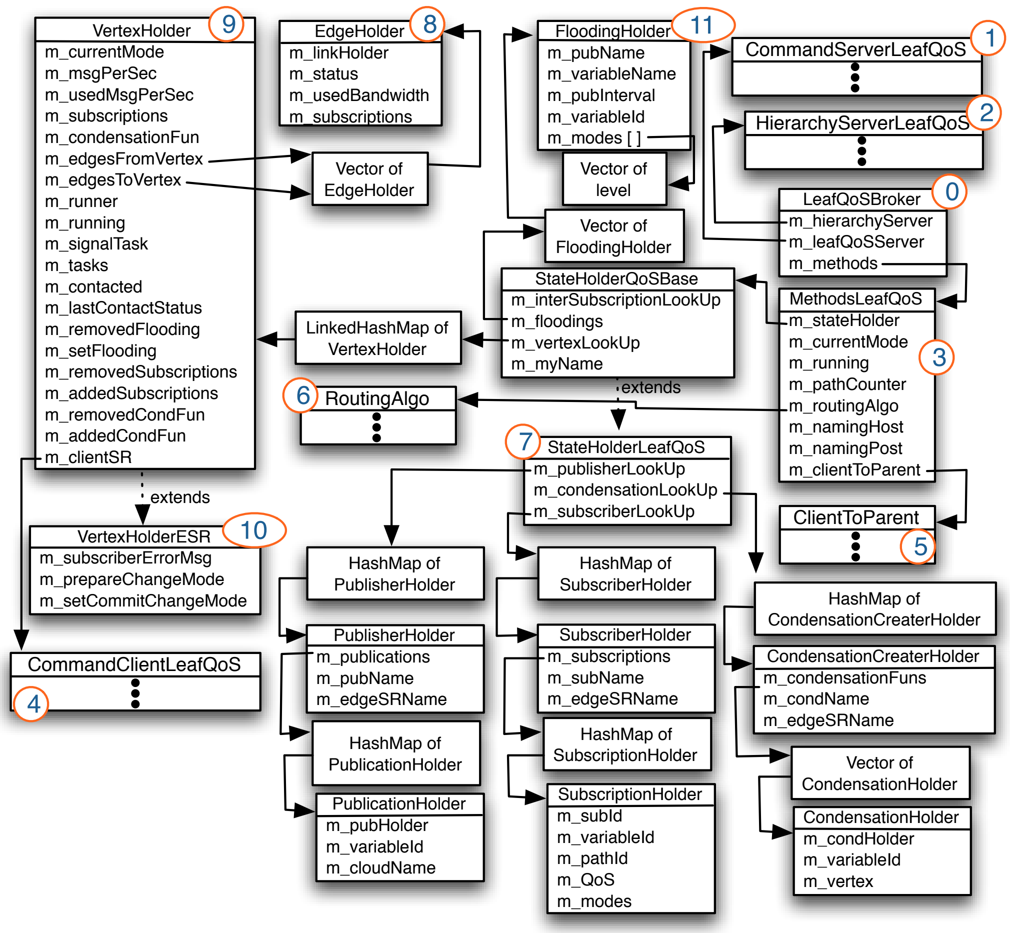


Figure 3.13: Design of the leaf QoS broker

The Leaf QoS Broker API

The `LeafQoSBroker` class (see label 0 in Figure 3.13) implements the `LeafQoSBrokerInterface` presented in Figure A.37. It is the class that connects together all the classes of a leaf QoS broker and provides an interface to the application layer of the leaf QoS broker. The intention is to have a policy-driven leaf QoS broker (the policy engine is not designed or implemented yet), but for the time being, a simple API is provided to be used by applications that control the operation of the leaf QoS broker. This

API provides for adding and removing event channels and status routers at runtime, and using the operating modes and limited flooding mechanisms.

The Leaf QoS Broker Command Module for the Data Plane The common functionality of the command module for the data plane was presented in Section 3.1.3. The `CommandClientLeafQoS` class (see label 4 in Figure 3.13) provides the command interaction that the leaf QoS broker will have with the (edge) status routers in its cloud. The leaf QoS broker provides the `CommandSRToLeaf` interface as its servant object for the (edge) status routers. The servant object is contained in the `CommandServerLeafQoS` (label 1) which interacts with the `MethodsLeafQoS` (label 3).

The Leaf QoS Broker Command Module for Management Plane The common functionality of the command module for the management plane was presented in Section 3.1.2. The `ClientToParent` class (see label 5 in Figure 3.13) provides the command interaction that the leaf QoS broker has with its parent QoS broker. An instance of this class is a member of the `MethodsLeafQoS` class (label 3), which is used to invoke requests to the parent QoS broker. The leaf QoS broker provides the `HierarchyLeafQoSChild` interface as its servant object for its parent QoS broker. The servant object is contained in the `HierarchyServerLeafQoS` (label 2), which uses the mechanisms in the `MethodsLeafQoS` class (label 3) to service requests from the parent QoS broker.

The Leaf QoS Broker Mechanism Module The class `MethodsLeafQoS` (label 3) is the class that connects together all the other classes in the leaf QoS broker. Both servant objects that the leaf QoS broker has, invoke the methods in this class to service requests from the remote invoker. The `MethodsLeafQoS` class uses the `RoutingAlgo` class (label 6) to find the subscription paths. In the `StateHolderLeafQoS` class (label 7), the leaf QoS broker maintains its own state and the state of all the (edge) status routers that it controls (see `VertexHolder` class (label 9).) Each `VertexHolder` contains the state of all the event channels (see `EdgeHolder` (label 8)) that the specific (edge) status router interacts with. Also, each `VertexHolder` contains the command client, `CommandClientLeafQoS` (label 4), for this specific (edge) status router.

Failure Model

The current failure model for the leaf QoS broker is that it will not fail. It is assumed that replication techniques will be used to provide uninterrupted, around the clock, service from a leaf QoS broker. The leaf QoS broker can recover from any communication failure, either to (edge) status routers that it controls or to its parent QoS broker. Upon reconnection the state is merged between the leaf QoS broker and the entity that it lost connection to.

3.3.2 QoS Broker

The QoS broker is the part of the framework that provides the functionality for inter-cloud subscriptions. The QoS brokers are organized as a hierarchy to improve scalability. It is envisioned that higher layers in management plane can abstract higher level knowledge of the resources beneath them, something that allows using "divide and conquer" approaches to control these resources. The QoS broker has two server command modules: one to serve requests from children (leaf) QoS brokers that it controls and another one to serve requests from the parent QoS broker (if any). It also has two client command module: one to issue requests to its children (leaf) QoS brokers and another one to issue requests to its parent QoS broker.

The rationale for the QoS broker is presented first, followed by the design of the above modules and the interactions among them. Finally, the failure model of the QoS broker is described.

Design of the QoS Broker

From the high level design (see Figure 3.1) it can be seen that the QoS broker consists of two major parts: the command module for the parent QoS broker and its mechanisms. The QoS broker uses the command interface `HierarchyQoSParent` to issue commands to its parent QoS broker and provides the servant interface `HierarchyQoSChild` (see Figure A.7) to its parent QoS broker. If the QoS broker is the "root" of the hierarchy, then these interactions are not used. In order to issue commands to its children (leaf) QoS brokers, the QoS broker uses the interfaces `HierarchyLeafQoSChild` if the child is a leaf QoS broker and `HierarchyQoSChild` if the child is a QoS broker. Upon initialization, the QoS broker registers with its parent QoS broker unless it is the "root" node of the hierarchy tree.

The design of the different Java classes and their interactions in the two modules are illustrated in Figure 3.14. We will now overview the different classes.

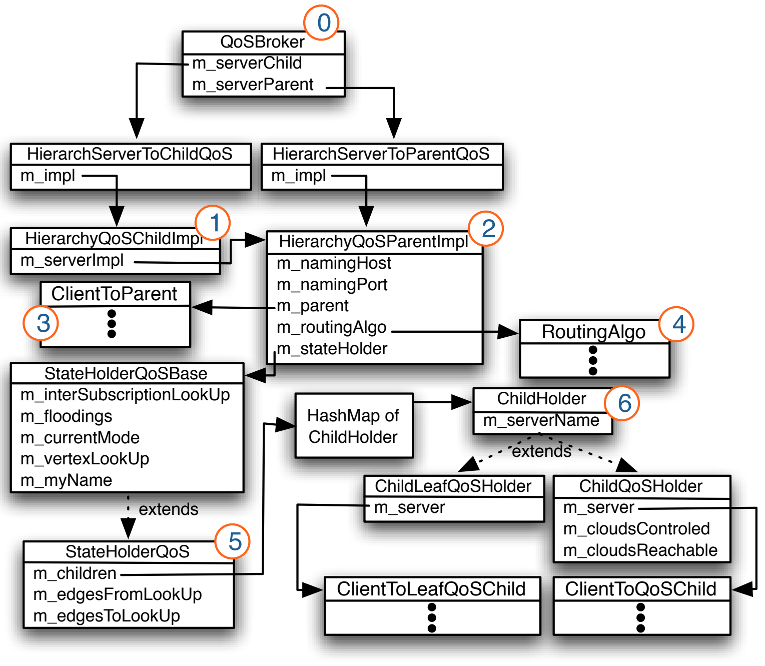


Figure 3.14: Design of the QoS broker

The QoS Broker Command Module

The common functionality of the command module for the management plane was presented in Section 3.1.2.

The `ClientToParent` class (see label 3 in Figure 3.14) provides the command interaction that the QoS broker has with its parent QoS broker. An instance of this class is a member of the `HierarchyQoSParentImpl` class (label 2), which uses it to invoke requests to its parent QoS broker. The QoS broker provides the `HierarchyQoSChild` interface as its servant object for its parent QoS broker. The servant object is contained in the `HierarchyQoSChildImpl` (label 1) which uses the mechanisms in the `HierarchyQoSParentImpl` class (label 2) to service requests from the parent QoS broker.

The `ChildHolder` class (see label 6 in Figure 3.14) holds the client classes that implement the interfaces that the children provide to the QoS broker. The leaf QoS brokers provide the `HierarchyLeafQoSChild` interface and a child QoS broker provides the `HierarchyQoSChild` interface that the QoS broker uses to issue requests to its children. The servant object is contained in the `HierarchyQoSParentImpl` class (label 2) which provides the `HierarchyQoSParent` interface that serves requests issued by the children of the QoS broker.

The QoS Broker Mechanism Module The class `QoSBroker` (label 0) bootstraps the QoS broker, instantiates and registers the servant objects. It is the role of the `HierarchyQoSParentImpl` class (label 2) to service requests that are received by the QoS brokers children and the parent QoS broker. In order to service the requests, the QoS broker maintains its state in the `StateHolderQoS` class (label 5). In its state, the QoS broker holds information about its children (leaf) QoS brokers and the inter-cloud subscriptions that the QoS broker was involved in allocating a sub-path or a complete path. The `RoutingAlgo` class (label 4) contains the QoS brokers routing algorithm. Currently, the only mechanism that the QoS broker has is to allocate inter-cloud subscription paths. The requests for such subscription paths are either received by the children or the parent QoS broker. When a request is received from a child, the QoS broker determines if it is an ancestor of both the publisher and subscriber. If this is the case, the QoS broker issues commands to its children to allocate the different sub-paths. If this is not the case, the request is forwarded to its parent QoS broker. If the request for a subscription path comes from the parent QoS broker, that means the QoS broker will establish a sub-path of the inter-cloud subscription path. The QoS broker uses its routing algorithm and issues commands to its children to establish this sub-path.

Failure Model

Currently the failure model for the QoS broker is the same as the one for the leaf QoS broker; QoS brokers will not fail. It is assumed that replication techniques will be used to provide uninterrupted, around the clock, service from a QoS broker. The QoS broker can recover from any communication failure, either to its children (leaf) QoS brokers or to its parent QoS broker.

CHAPTER 4

MULTICAST RATE-FILTERING ROUTING MECHANISM

The multicast rate-filtering routing mechanism provides for multicast of event streams and deterministic filtering for temporally related status variables. What is meant with multicast is that only one copy of a status event is sent over an event channel even-though there are multiple subscription paths for this variable. Temporally related variables are variables that are all published at the same interval and events are published at the same time (using a GPS clock.) This provides for the possibility of taking a snapshot of the state of such a set of variables if all the events that are published at the same time are delivered to the subscriber. Different subscribers may want to receive such snapshots at different subscription intervals, so what is meant by deterministic filtering here is that the filtering mechanism will forward such sets of variables at all the requested intervals.

This mechanism is placed on the (edge) status routers and does not need any management from the management plane. This mechanism is provided by the data-structure and the routing algorithm that the (E)SRs are using, so it is transparent to the end-points and always activated. The multicast feature is a side-affect of the data-structure used in the (E)SR, and as a result no extra computational resources are needed. The filtering algorithm does use computational resources and the resource usage is analyzed later in this chapter.

The requirements for the multicast rate-filtering routing mechanism are the following:

1. If a subscription is at a larger subinterval[1] than the published interval, the unnecessary events will be filtered.

2. Filtering will occur as close to the source as possible, without violating any of the downstream subscriptions.

3. If a set of variables is temporally related, then, when subscribed to at the same interval, the

[1]Subinterval refers to an interval that is a multiple of another interval

complete set (in the same phase) will be delivered to the subscriber.

4. Only one copy of a status event with the same timestamp will be forwarded over an event channel (multicast).

The rationale for the multicast rate-filtering routing mechanism is presented, followed by the design of the data-structure and routing algorithm that provide the mechanism. The design of the mechanism is presented next and finally, an example of the mechanism is described.

4.1 Rationale for the Multicast Rate-filtering Routing Mechanism

The rationale for this mechanism is to optimize the usage of the resources in the data plane. If only the data that is required by the subscribers is forwarded and only one copy of each of the data items is routed through a (E)SR or forwarded over an event channel, then the resource usage in the data plane is minimized.

The rationale for the filtering part of this mechanism is to forward only the amount of data that is required by the end-points and drop the excess data as close to the source as possible. The filtering mechanism takes advantage of the semantic of the published status variables in order to efficiently forward only the needed status events. When a status variable is published, its publication interval is registered as well, and similarly, when a subscriber registers a subscription, it requests a subscription interval. This information is given to the routing algorithm that is used by the (E)SRs to route the status events.

The application domain (the electric power grid) that the framework is envisioned for disseminates temporally related variables. Therefore, the filtering algorithm has to be deterministic as to forward exactly the same set (created at the same time) of status events for these variables. For instance, suppose that there are 10 variables that are published every 10 ms and subscribed to at an interval of 40 ms. If 5 of the variables are delivered every 30 ms and the remaining are delivered every 40 ms the set of all 10 variables is not a consistent snapshot. Although all the variables are delivered within the specified interval, it is not the same set of variables and for certain application domains the received information is useless. The filtering algorithm deployed for this mechanism supports the deterministic filtering of temporally related variables and hence can be used to take a snap-shot of the state of the subscribed variables.

The rationale for the multicast part of this mechanism is to forward only one copy of the same status event over an event channel. As a side affect of this, an (E)SR will only receive one copy as well. By only forwarding one copy, both computational and network resources are saved in the data plane. It is possible that a subscription could be set up that will not consume any resources from the data plane if another subscriber connected to the same ESR had already requested the same subscription.

4.2 Design of the Multicast Rate-filtering Routing Mechanism

The multicast rate-filtering routing mechanism consists of two parts; the data-structure of the routing table (with inherent multicast) and the routing (forwarding) algorithm (with filtering) in the (edge) status routers. Two different approaches of how the routing table should be constructed and how the structure of the routing table affects the computational cost of forwarding events are presented. The difference between the two is that the first, called *close routing*, is optimized when the range of the subscription intervals (for variables that are to be routed through an (E)SR) is small (close), while the second, called *sparse routing*, is optimized when this range is large (sparse). The two different data-structures also have their own routing algorithm. The choice of which of the two approaches to use is set as a compile time option, but there is flexibility in the sense that some of the (E)SRs could use *close routing* while the remaining would use *sparse routing*.

The following assumptions were made when designing the data-structure and routing algorithm:

Assumption 1: The publishers will publish status events at the registered interval.

Assumption 2: The subscribers only request the status event from the first phase of a subscription interval. For example, a status variable is published every 10 ms and subscribed to every 30 ms. Assuming that the publication of events started at time 0, then events with timestamp 0, 30, 60, 90, ... (the first phase within the 30 ms subscription interval) are forwarded while events from the two other phases (10, 40, 70, 100, ... and 20, 50, 80, 110, ...) are dropped.

Assumption 3: If the requested subscription interval is not a multiple of the publishing interval, then the subscription interval will be converted to the closest (but always less than) multiple of the publishing interval.

The filtering algorithm is presented first, followed by the data-structure of the routing table. Then the routing algorithms and an analysis of the two approaches will be presented. Finally, it is explained how the design satisfies the requirements for this mechanism.

4.2.1 The Filtering Algorithm

The main idea behind the filtering algorithm is the following function:

$$if\ ((event.TS + \lceil pubInt/2 \rceil\ \%\ subInt[0..i] < pubInt)\ then\ forward\ event \qquad (4.1)$$

Where:

- event.TS is the timestamp embedded inside the status event.
- pubInt is the publishing interval of the status variable.
- subInt is the subscription interval.
- i is the number of subscriptions with unique subInt for this variable.

Figure 4.1 illustrates a timeline and shows when events from four publishers are sent. The four publishers are supposed to publish at the same time every 50 ms, but because they are different application instances they didn't start at exactly the same time. So, events that are within each square and angle bracket are grouped together and should be considered as a set of events with the same timestamp. For example, events that are supposed to have timestamp 100 ms will be any event that has a timestamp between 100ms – $\lceil$ pubInt/2 $\rceil$ and 100ms + $\lfloor$pubInt/2$\rfloor$ – 1.

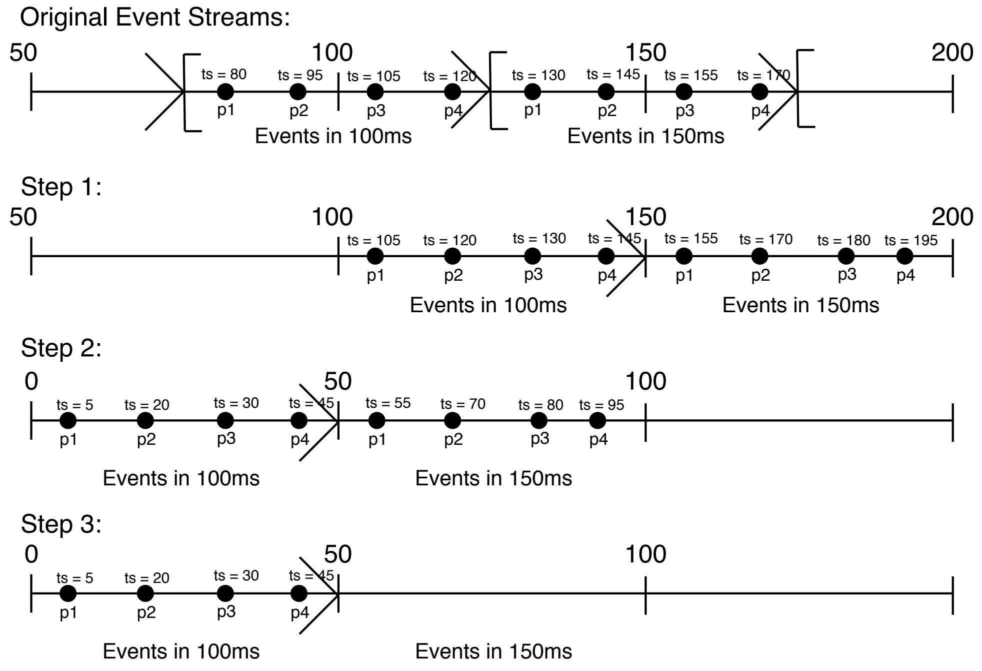


Figure 4.1: Illustration of the steps of the filtering function

Function 4.1 is divided into three steps as illustrated in Figure 4.1.

Step 1: The first step is to shift all the events timestamps with half the publishing interval to the right. In this way, the events that arrive in the 100 ms timestamp will have the timestamp value (after the addition) of 100 ms to 149 ms.

Step 2: Assume that the subscriber has subscribed at an interval of 100 ms. That means all the events in the 100 ms, 200 ms, 300 ms intervals are supposed to be forwarded, and all other events should be dropped. To do this, the mod of the subscription interval is taken of the result of Step 1. The result of the mod operation will give events with timestamp 0 to 99.

Step 3: After step 2 there are two phases with events from the publishers because subInt/pubInt equals 2. The framework will always pick the first phase, so the result of step 2 is compared to test if it is less than the pubInt (the first phase). If this is the case then the event is forwarded, if not it is dropped. Events with original timestamp value between 75 and 124 will be forwarded, while the others (timestamp value between 125 and 174) will be dropped.

4.2.2 The Data-structure and Routing Algorithms

The data-structure for the two different routing approaches is presented in Figure 4.2. The data-structure changes during runtime as subscriptions are added and removed from it. For each variable a `RoutingEntry` (label 2) is created, where `pubInterval` and `pubIntervalHalf` (pubInterval/2) are stored. The `RoutingEntry` also has a table of modes where the subscriptions for the different operational modes are stored. The `RoutingEntry` is inserted into the routing table hashed on the `variableId`. When a subscription is added to a status router and it is the first subscription for the `variableId` then a new `RoutingEntry` is created, otherwise the existing `RoutingEntry` is used. For every subscription added to the routing table a `PathHolder` (label 10) is created where information about each subscription is stored. The `PathHolder` is placed in a list in `RoutingHolder` (label 9). The `RoutingHolder` will be explained later because it is different for the two approaches. When a subscription is removed its `PathHolder` is removed and the `RoutingHolder` is updated accordingly. If the subscription was the last one for a specific mode then the `ModeHolder` for that mode is also removed. Similarly, if the subscription was the last for a specific status variable, then the `RoutingEntry` is removed.

The specifics of the data-structure, how it is updated (when subscriptions are added and removed), and the routing algorithm for the two different approaches are explained. Lastly, a big-O analysis of the two approaches is presented.

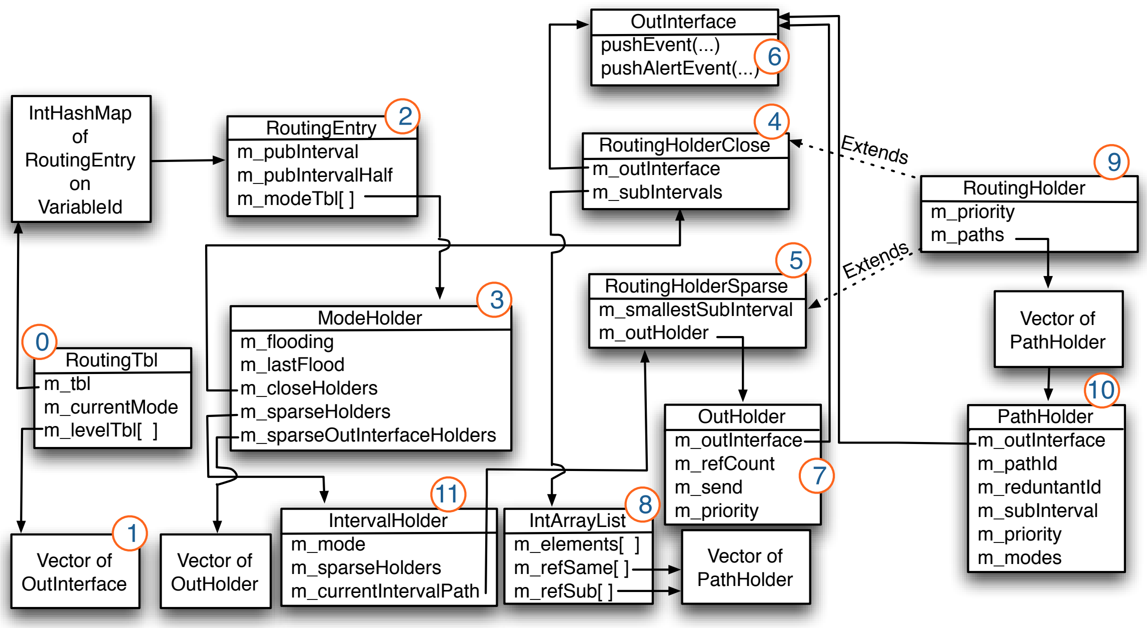


Figure 4.2: Design of the routing table in the (edge) status router

Close Routing Data-structure and Routing Algorithm

The *close routing* approach uses the class `RoutingHolderClose` as the specialization of the `RoutingHolder`. A `RoutingHolderClose` is created for each of the event channels that the status variable is to be sent on. If a subscription path is the first one for a specific variable to be sent out on a event channel, a new `RoutingHolderClose` is created and the member variables are set. If it is not the first path for the event channel, then an existing `RoutingHolderClose` is used. When a path is added to a `RoutingHolderClose` it is inserted sorted into the `IntArrayList` (label 8) class. `IntArrayList` class maintains an array (`m_elements[]`) of the smallest unique subintervals that status events from this variable are to be forwarded. The member variables `m_refSame[]` and `m_refSub[]` are used to keep track of subscriptions with the same subscription interval and subscription intervals that are multiples of the interval in `m_elements[]`. The values in `m_elements[]` are sorted from smallest to largest. In the case of a path with a smaller subscription interval than what it currently contains, the contents of all three variables (`m_elements[]`, `m_refSame[]`, and `m_refSub[]`) are shifted one position down to keep the list sorted.

Below is an example of how a specific `RoutingHolderClose` would be updated as subscription paths are added and removed. If the first path to be added to a `RoutingHolderClose` has a subscription interval of 90 ms, then `m_elements[0]` gets the value 90 and the path is added to `m_refSame[0]`. If the second path to be added also has a subscription interval of 90 ms, then it is added to the list of `m_refSame[0]`. The third path has a subscription interval of 180, and it is added to the list of `m_refSub[0]`. If the first two subscriptions are removed then `m_elements[0]` gets the value 180. The third path that was added is the only entry in the list of `m_refSame[0]` and the list `m_refSub[0]` is empty.

The routing algorithm for the close routing approach is presented in Figure 4.3. The routing algorithm goes through all the `RoutingHolderClose` (line 29) that forwards the variable for some subscription interval. Then for each of these `RoutingHolderClose` it goes through all the unique subscription intervals (line 34) to test (see Function 4.1) if any of them are satisfied (line 36). For the first interval that is satisfied, the event is sent and the inner-loop terminates. If the event is of type alert the `pushAlertEvent` method (line 42) is used to send the event, otherwise the method `pushEvent` (line 45) is used. In the case that the queues in the sending module are full, those two methods will return `false`. Therefore, the algorithm must check if it has to be responsible for the memory management of the event (line 55 – 60).

The tradeoffs for this approach are listed next:

Pros:

- A very easy algorithm.
- Read only operations.

Cons:

- For those outgoing event channels that an event will not be forwarded on the inner for-loop for each of these outgoing channels is iterated max times. i.e. worst case searching for a non-existing element.
- Computational cost for each subscription depends on how many other subscriptions for this variable go to other outgoing event channels.

Sparse Routing Data-structure and Routing Algorithm

The *sparse routing* approach uses the class `RoutingHolderSparse` as the specialization of the `RoutingHolder`. The strategy of the close routing approach is to organize the routing table based on the event channel that the event should be sent on and then within each of these event channel holders maintain the subscription intervals (filtering). The sparse routing uses the opposite strategy; first organize the lookup table on the subscription interval (filtering) and then for each of these intervals have a list of the event channels. Therefore, this approach performs the filtering first and the routing afterwards.

When the first subscription path is added to the routing table a `RoutingHolderSparse` (label 5), `OutHolder` (label 7), and `IntervalHolder` (label 11) are created. The `RoutingHolderSparse` holds one or more paths where all these paths are being forwarded on the same event channel and their subscription interval is a multiple of each other. The smallest interval from all the paths that belong to this holder is stored in `m_smallestSubInterval`. For each of the smallest subscription interval that is not a subinterval of one of the other smallest subscription interval an instance of the class `IntervalHolder` is created. This means that in the member variable `m_sparseHolder` in the `ModeHolder` class will maintain a list of all the subscription intervals that this variable should be sent out on one or more of the event channels. The member variable `m_sparseOutInterfaceHolders` maintains a list of all event channels (`OutHolder` see label 7) that status events for this variable are to be forwarded on. During the routing algorithm the member variable (flag) `m_send` in the class `OutHolder` will be set if the event is to be forwarded over this event channel. Once the event is forwarded the flag is reset.

Following is an example of how a specific RoutingHolderSparse and IntervalHolder would be updated as subscription paths are added and removed. If the first path to be added has a subscription interval of 90 ms, then a RoutingHolderSparse is created and the member variable m_smallestSubInterval is set to 90. An OutHolder is also created and set as a reference in the m_outHolder in the RoutingHolderSparse. It is added to the member variable list m_sparseOutInterfaceHolders in the ModeHolder. An instance of the class IntervalHolder is also created and the RoutingHolderSparse that was just created is added to the member variable list m_sparseHolders and the member variable m_currentIntervalPath is a reference to this RoutingHolderSparse as well. The next path that is added has a subscription interval of 180 ms and is to be forwarded over the same event channel as the first one. The path is added to the member variable list m_paths in the RoutingHolder class, but nothing else is needed to be done. The third path has a subscription interval of 180 ms, but is to be forwarded to a different event channel than the first two paths. A new instance of the RoutingHolderSparse is created and the member variable m_smallestSubInterval is set to 180. A new OutHolder is created and a reference to it is set in the member variable m_outHolder in the newly created RoutingHolderSpars instance. This OutHolder is also added to the member variable list m_sparseOutInterfaceHolders in the ModeHolder. The newly created RoutingHolderSparse is added to the member variable list m_sparseHolders in the IntervalHolder that was created for the first path.

If the two first subscriptions are removed then the first RoutingHolderSparse instance and OutHolder instance are removed. The member variable m_currentIntervalPath member variable of the IntervalHolder instance is changed to reference the RoutingHolderSparse that was created for the third subscription. This means that the m_smallestSubInterval is now 180.

The routing algorithm for the sparse routing approach is presented in Figure 4.4. The routing algorithm goes through all the `IntervalHolders` (line 29) and tests (see Function 4.1) if the status event should be sent to at least one event channel (line 34). Then the inner-loop starts (line 37) to test which of the event channels this event should be sent on. If it is to be sent, then the flag `m_send` is set for this event channel (line 46). After all the event channels that this status event should be sent to are marked, the algorithm loops through all the event channels (line 56) to locate these ECs. For each of these ECs, if the event is of type alert then the `pushAlertEvent` method (line 65) is used to send the event, otherwise the method `pushEvent` (line 68) is used. In the case that the queues in the sending module are full, those two methods will return `false`. Due to this, the algorithm must check if it has to be responsible for the memory management of the event (line 76 – 80).

The tradeoffs for this approach are listed next:

Pros:

- The worst-case problem of the close routing approach is eliminated

Cons:

- Need a bookkeeping table.
- Write operations are used during the routing algorithm.

Analysis of the Two Solutions

Tables 4.1 and 4.2 show the cost (runtime) of forwarding an event to 1, 1/2, all the event channels that the status router has. Table 4.1 shows the costs when multicast is active, that is multiple subscriptions with different subintervals, but to the same event channel that gets satisfied. For example, for the event with timestamp 0 all the subscriptions will be satisfied. Table 4.2 shows the cost when there is no such multicasting going on. In reality the multicast will occur for some of the events of a stream, but not all; look at the example in Section 4.3.

As it can be seen from the tables, the complexity of close routing approach is not affected if multicast occurs or not. The sparse routing approach is affected in terms of its worst case and average case of sending on *ol/2* and *ol* event channels. The reason for the multiplication is that of multicast for different subintervals that will all forward the specific event.

The terms used for the complexity analysis of the two routing approaches are:

- l: total number of event channels for all the subscriptions to a specific variable.
- ol: outgoing event channels, with at least one subscription for the specific event to be routed.
- si: distinct subscription intervals for the specific event to be router.
- runtime: the computational cost of routing the event.

	#ec	Close Routing	Sparse Routing
worst case	1	(ol-1)*si+(si-1)	si+l
	ol/2	(ol/2)*si	si*(ol/2)+l
	ol	ol*si	si*ol+l
avg. case	1	(ol-1)*si+ si/2	si+l
	ol/2	(ol/2)*(si/2)+(ol/2)*si	si/2+(si/2)*(ol/2)+l
	ol	ol*(si/2)	si/2+(si/2)*ol+l
best case	1	(ol-1)*si+1	si+l
	ol/2	(ol/2)+(ol/2)*si	si+(ol/2)+l
	ol	ol*1	si+ol+l

Table 4.1: Complexity of the routing approaches when multicast does not occur

	#ec	Close Routing	Sparse Routing
worst case	1	(ol-1)*si+(si-1)	si+l
	ol/2	(ol/2)*si	si+(ol/2)+l
	ol	ol*si	si+ol+l
avg. case	1	(ol-1)*si+ si/2	si+l
	ol/2	(ol/2)*(si/2)+(ol/2)*si	si+(ol/2)+l
	ol	ol*(si/2)	si+ol+l
best case	1	(ol-1)*si+1	si+l
	ol/2	(ol/2)+(ol/2)*si	si+(ol/2)+l
	ol	ol*1	si+ol+l

Table 4.2: Complexity of the routing approaches when multicast occurs

4.2.3 Design Satisfaction of the Requirements

Four requirements were specified for the multicast rate-filtering routing mechanism and by examining the design of the mechanism and the algorithm used, it will be shown that the design satisfies those requirements.

The first requirement that only the requested status events are received by the subscribers is satisfied by both the filtering function (Function 4.1) and the forwarding algorithm used in the (edge) status routers. To satisfy this requirement it is enough that the edge status router, that the subscriber is connected to, will perform the necessary filtering. Assume that this edge status router receives the events for the specific variable with a smaller interval than what the subscriber subscribes to. In the close routing, the edge status router tries to route the unnecessary events to the event channel that connects to the specific subscriber they will be filtered in line 36 of the routing algorithm (see Figure 4.3). Similarly, in the sparse routing algorithm, the `RoutingHolderSparse` for the event channel to the subscriber will have the subscribed interval as the value for the member variable `m_smallestSubInterval`. Hence, the unnecessary events would be filtered at line 34 of the routing algorithm (see Figure 4.4).

The second requirement is also satisfied. For the close routing approach, it is always the subscription with the smallest multiple of the published interval that is placed in the member variable `m_elements` in the class `IntArrayList`. These are the intervals that are used by the routing algorithm when it tests if an event should be forwarded or not (see line 36 in Figure 4.3). Since there is no subscription with a smaller interval than the subscription with the smallest interval, the events are filtered as close to the source as possible. Similarly, in the sparse routing algorithm, each of the `IntervalHolder` instances contains the smallest interval of all the subscriptions that are multiple of each other. Each of the `RoutingHolderSparse` instances holds the smallest interval of all the subscriptions that are a multiple of each other and are to be forwarded to a specific event channel. In line 29 of the sparse routing algorithm (see Figure 4.4) only the events that have at least a subscription for them are considered. Then, at line 42 only events that have at least one subscription (for the given event channel) that satisfies the timestamp of the event are forwarded to that specific event channel. So events are only forwarded to an event channel if there exists at least a subscription that the event's timestamp is satisfied for.

The third requirement that temporally related variables are filtered in a deterministic way as to provide a complete set to a subscriber is satisfied. Temporally related variables by definition will generate events at the same publishing interval and the events will have the "same" timestamp. A subscriber that wants to subscribe to such a set, will subscribe to all the variables with the same subscription interval. The filtering function algorithm, Function 4.1, uses only the publishing interval (this is the same for all the variables), the timestamp of the event (this is the "same" for all the events in the set), and the subscription interval (which is the same for all the variables). Since all the information that the filtering algorithm uses to forward the events are the same, then the filtering will be deterministic.

The fourth requirement that only one copy of a status event with the same timestamp is forwarded over an event channel is satisfied. For the close routing algorithm when a status event is forwarded to an event channel (see line 40 – 46 in Figure 4.3), then the loop is terminated in line 49. This will make sure that at most one status event with the same timestamp is sent over an event channel. For the sparse routing algorithm it only forwards events to event channels in line 56 – 73 of the routing algorithm (see Figure 4.4). Those lines loop through all the event channels that this status event could potentially be sent over, since each event channel is only visited once in the loop at most one status event with the same timestamp is sent over an event channel. Hence the requirement is satisfied for both design approaches.

4.3 Example of the Multicast Rate-filtering Routing Mechanism

An extended example of the two different designs for the multicast rate-filtering routing mechanism is presented. Figure 4.5, illustrates how the five (edge) status routers are connected and to which edge status router the different publishers and subscribers are connect to.

The following publications and subscriptions are registered and established:

- Pub p_0 publishing with interval: 30 ms.
- Sub s_0 subscription interval: 90 ms.
- Sub s_1 subscription interval: 180 ms.
- Sub s_2 subscription interval: 630 ms.
- Sub s_3 subscription interval: 240 ms.
- Sub s_4 subscription interval: 630 ms.

Figure 4.6, illustrates how the data-structure for the close routing approach looks like at SR i_0. The `ModeHolder` contains three `RoutingHolderClose`, one for each of the outgoing event channel. The `RoutingHolderClose` for the event channel that is connected to ESR e_1 has two entries in the `IntArrayList`. One for each of the smallest subintervals for the status events that are to be forwarded on this event channel. Similarly, the two other `RoutingHolderClose` only have one entry in their `IntArrayList` since only one subscription is registered to be forwarded over each of these event channels.

Figure 4.7, illustrates how the data-structure for the sparse routing approach looks like at SR i_0. The `ModeHolder` contains three `IntervalHolder`, one for each of the distinct subscription intervals, namely 90, 240, and 630. For the first two intervals the `IntervalHolder` has only one `RoutingHolderSparse` each, since for these two intervals the subscriptions are only to be forwarded to one event channel each. For the third interval the member variable list `m_sparseHolders` has two `RoutingHolderSparse` entries. This is because for this interval events are to be forwarded to two different event channels.

The member variable list `m_sparseOutInterfaceHolders` (in the `ModeHolder` instance) has three `OutHolder` instances in it. One for each of the event channels that this variable has at least one subscriptions for.

Figure 4.8 shows a time line and how the event streams for this example are filtered from the source to the different destinations. For the hop ESR e_0 to SR i_0 events have to be forwarded every 90 ms to satisfy Sub s_0, in addition, it must also forward the events that belong to the 240 interval for Sub s_3. For the hop SR i_0 to ESR e_1 events have to be forwarded every 90 ms to satisfy Sub s_0, and since this is a subinterval of 180 and 630 the subscriptions that Sub s_1 and s_2 have are multicast. For the hop SR i_0 to ESR e_2 events have to be forwarded every 240 ms to satisfy Sub s_3, and no other subscriptions need to be forwarded over this event channel. Similarly, for the hop SR i_0 to ESR e_3 events have to be forwarded every 630 ms to satisfy Sub s_4.

```
public boolean routeEvent(EventHolder event, SocketAddress fromAdr)
{
  RoutingEntry entry;
  ModeHolder modeHolder;
  int variableId = event.m_event[0].getInt(Constants.EVENT_VARIABLE_ID_OFFSET);
  long timeStamp = event.m_event[0].getLong(Constants.EVENT_CREATED_OFFSET);

  // Do we route this variable ?
  if ((entry = (RoutingEntry)this.m_tbl.get(variableId)) == null)
    return false;

  // Do we route this variable in the current operating mode ?
  if ((modeHolder = entry.m_modeTbl[this.m_currentMode]) == null)
    return false;

  // Do we flood the event ?
  if (modeHolder.m_flooding > 0)
  {
    ........
    return true;
  }

  // Route this event
  size = modeHolder.m_closeHolders.size();

  // Set the expected ref count for this event
  event.incrementRef(size);

  for (int i = 0; i < size; ++i)
  {
    routingHolder = (RoutingHolderClose) modeHolder.m_closeHolders.get(i);

    // Should we forward this event on this Event Channel ?
    for (int j = 0; j < routingHolder.m_subIntervals.m_size; ++j)
    {
      if ((timeStamp + entry.m_pubIntervalHalf) %
           routingHolder.m_subIntervals.m_elements[j] < entry.m_pubInterval)
      {
        // Is this an alert ?
        if (routingHolder.m_priority == CommConstants.PRIORITY_ALERT)
        {
          if (routingHolder.m_outInterface.pushAlertEvent(event))
            ++sentTo;
        } // Send the event with its priority
        else if (routingHolder.m_outInterface.pushEvent(event,routingHolder.m_priority))
          ++sentTo;

        // Terminate the loop
        j = routingHolder.m_subIntervals.m_size;
      }
    }
  }

  // Decrement the reference for all the ECs that the event was not sent on
  if ((size - sentTo) > 0)
  {
    if (event.decrementRef(size - sentTo) == 0)
      return false; // This event must be recycled
  }

  return true; // Event is sent|filtered and memory management is completed
}
```

Figure 4.3: Filtering and multicast using the close routing approach

```java
public boolean routeEvent(EventHolder event, SocketAddress fromAdr)
{
  RoutingEntry entry;
  ModeHolder modeHolder;
  int variableId = event.m_event[0].getInt(Constants.EVENT_VARIABLE_ID_OFFSET);
  long timeStamp = event.m_event[0].getLong(Constants.EVENT_CREATED_OFFSET);

  // Do we route this variable ?
  if ((entry = (RoutingEntry)this.m_tbl.get(variableId)) == null)
    return false;

  // Do we route this variable in the current operating mode ?
  if ((modeHolder = entry.m_modeTbl[this.m_currentMode]) == null)
    return false;

  // Do we flood the event ?
  if (modeHolder.m_flooding > 0)
  {
    ........
    return true;
  }

  IntervalHolder intervalHolder;
  RoutingHolderSparse routingHolderSparse;
  OutHolder outHolder;
  int missed = 0;
  int sentTo = 0;

  for (int i = 0; i < modeHolder.m_sparseHolders.size(); ++i)
  {
    intervalHolder = (IntervalHolder)modeHolder.m_sparseHolders.get(i);

    // Should this interval be forwarded
    if ((created + entry.m_pubIntervalHalf) %
         intervalHolder.m_currentIntervalPath.m_smallestSubInterval < entry.m_pubInterval)
    {
      for (int j = 0; j < intervalHolder.m_sparseHolders.size(); ++j)
      {
        routingHolderSparse = (RoutingHolderSparse)intervalHolder.m_sparseHolders.get(j);

        // Have we already marked this one and should it be marked ?
        if (routingHolderSparse.m_outHolder.m_send == false &&
          ((created + entry.m_pubIntervalHalf) %
           routingHolderSparse.m_smallestSubInterval < entry.m_pubInterval))
        {
          routingHolderSparse.m_outHolder.m_send = true;
          ++sentTo;
        }
      }
    }
  }

  event.incrementRef(sentTo);

  // Send the event
  for (int i = 0; i < modeHolder.m_sparseOutInterfaceHolders.size(); ++i)
  {
    outHolder = (OutHolder)modeHolder.m_sparseOutInterfaceHolders.get(i);

    if (outHolder.m_send)
    {
      // Is this an alert
      if (routingHolder.m_priority == CommConstants.PRIORITY_ALERT)
      {
        if (!outHolder.m_outInterface.pushAlertEvent(event))
          ++missed;
      }
      else if (!outHolder.m_outInterface.pushEvent(event, outHolder.m_priority))
        ++missed;

      outHolder.m_send = false;
    }
  }

  // Make sure that we don't encounter mem. leaks
  if (missed > 0)
  {
    if (event.decrementRef(missed) == 0)
      return false; // This event must be recycled
  }

  return true; // Event is sent|filtered and memory management is completed
}
```

Figure 4.4: Filtering and multicast using the sparse routing approach

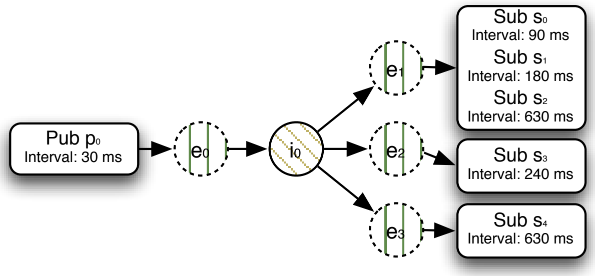


Figure 4.5: Filtering and multicast example topology

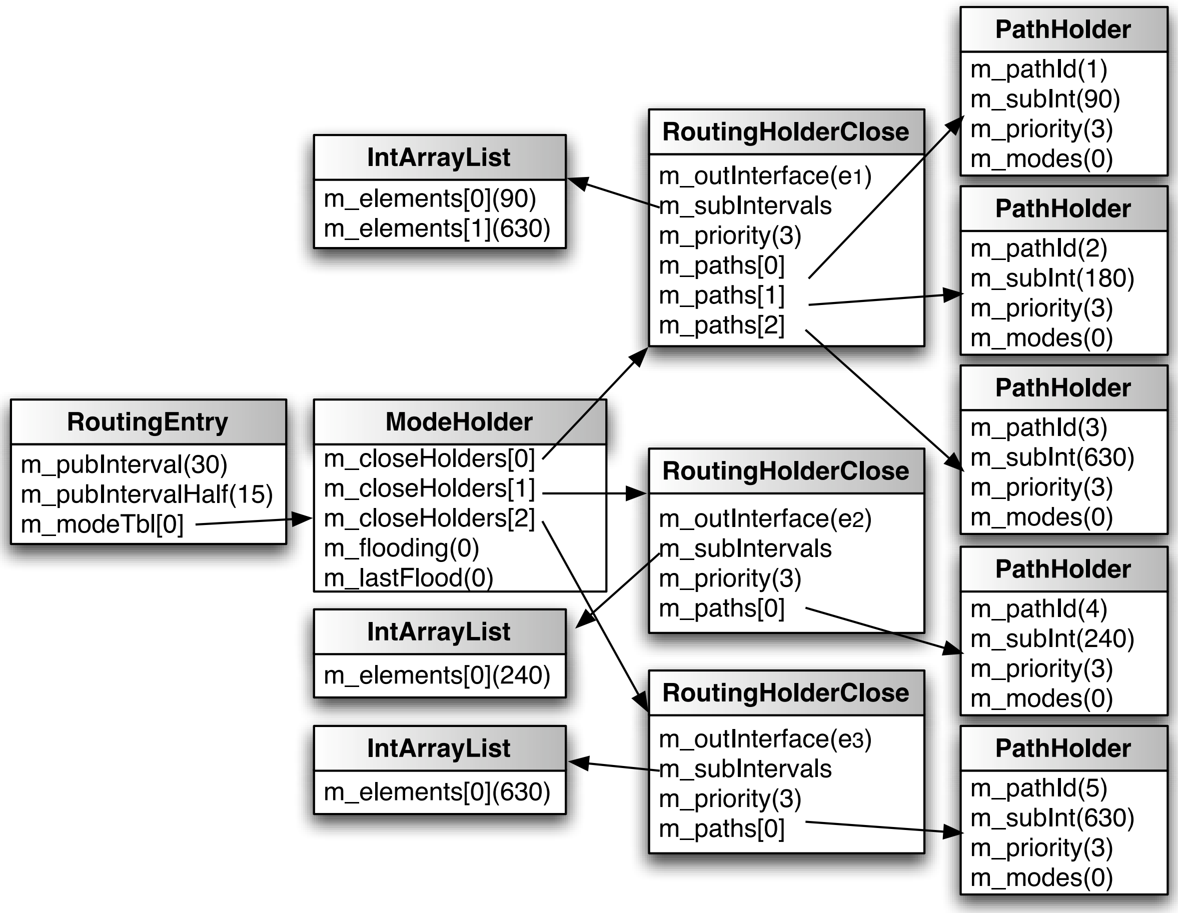


Figure 4.6: The data-structure for close routing

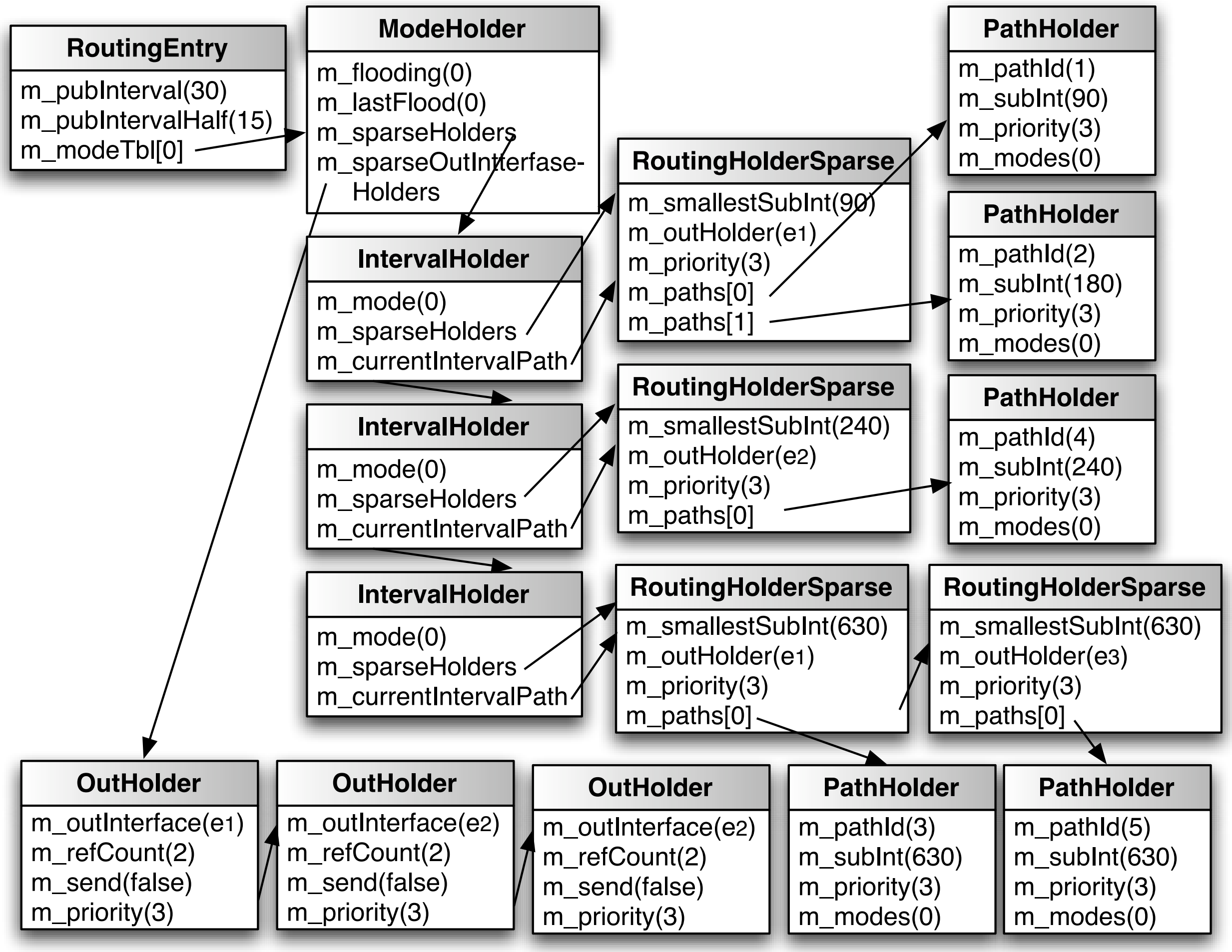


Figure 4.7: The data-structure for sparse routing

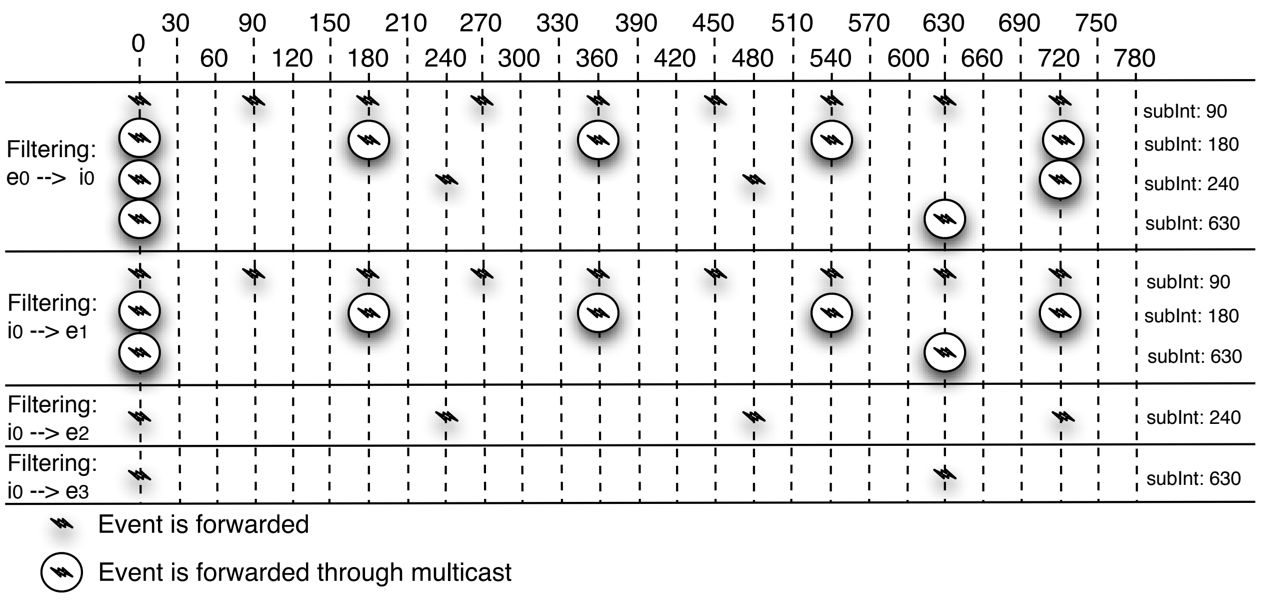


Figure 4.8: Timeline of event streams

CHAPTER 5

LIMITED FLOODING MECHANISM

The limited flooding mechanism distributes events from specific variables to a hierarchical scope. A hierarchical scope consists of a single or multiple clouds, but not selective parts of clouds. The need for this new mechanism is to be able to quickly adapt between subscribed event streams only to flooding a variable within a limited region and vice-versa. A variable may be flooded as to reduce the end-to-end latency or the variable may be of interest to subscribers that have no other way of inferring this interest except to receive events from the variable. The decision of when a variable is flooded, and to which hierarchical scope, is determined by management plane policies or initiated by control applications in the management plane.

The hierarchical scope to be flooded is specified by the number of levels of a QoS broker whose descendants will all be flooded. For instance, consider Figure 2.1 if a status variable is set to be flooded to level 3 in the system, then events are distributed to the entire system, since it only has 3 levels. However, if a variable registered in the leftmost cloud of Figure 2.1 is specified to be flooded to level 2, then only the leftmost and middle clouds receive the events. A subscriber in the rightmost cloud does not receive the events through flooding, but instead it has to subscribe to it.

The requirements for the limited flooding mechanism are the following:

1. Flooding of a variable is transparent to subscribers that are already subscribed to it or will subscribe to it in the future. This means that during the transition to flooding the existing subscriptions to the variable that is flooded will not be affected.

2. Subscribers to the specific variable that is currently being flooded will not notice any interruptions in their event streams as the flooding is being terminated. This means that during the transition from flooded variable to event stream only variable no events will be dropped from event streams (subscriptions) that have been established for the variable.

3. Subscribers to a specific variable that are not in a cloud where the variable is being flooded will not notice any interruptions during transitions to and from flooding in any other cloud. This means that as a variable is flooded within some clouds the subscriptions that are established from subscribers in other clouds will not notice the flooding.

4. (Edge) status router may receive *ec* copies of each status event of a flooded variable (where ec is the number of event channels that the (edge) status router has), but only the first copy will be forwarded to all the event channels (except the one where the event was received from).

The rationale for the limited flooding mechanism is presented, followed by a discussion on how the commands that initiate a flooded variable within a hierarchical scope are propagated from the management plane to the data plane. The design of the mechanism is presented next and finally, examples of how events from a flooded variable propagate through the flooded hierarchical scope are illustrated.

5.1 Rationale for the Limited Flooding Mechanism

The purpose of the flooding mechanism is twofold. First, it minimizes the end-to-end latency of events by bypassing the routing mechanisms and spreading the events to all the subscribers in a limited area. Second, flooding allows subscribers to receive all flooded events without explicitly subscribing to each one of them. The flooded variables, depending on the policy, might reflect something special about the condition of the application and, as a result, it is possible that the set of these variables is updated over time or as the condition of the application changes. The subscribers that subscribe to all flooded events typical log the occurrences of the event or some events will trigger further analyzation of the condition of the system.

The tradeoffs for the flooding mechanism are related to the end-to-end latency, bandwidth usage for the event channels, and computational usage in the (edge) status routers. There are four cases to consider when talking about these tradeoffs.

Case 1: If there are very few subscribers that are interested in few events from a flooded variable, then lower end-to-end latency for these few subscribers are traded off against more bandwidth and computational resource usage.

Case 2: If there are very few subscribers that are interested in all events from a flooded variable, then lower end-to-end latency for these few subscribers are traded off against more bandwidth and computational resource usage.

Case 3: If there are many subscribers that are interested in few events from a flooded variable, then lower end-to-end latency for these few subscribers are traded off against more bandwidth and computational resource usage.

Case 4: If there are many subscribers that are interested in all the events from a flooded variable, then there are no tradeoffs. Lower end-to-end latency to the subscribers, less computational resources used and the bandwidth usage are the same as if the variable was not flooded.

An example of the use of this mechanism could be for a monitor and control application for the electric power grid. If the stability of the power grid, within a hierarchical scope, depends on the operation of a few major transmission lines; then, when the operational state of one of these lines changes, most of the entities (subscribers) within the hierarchical scope would be interested in this information. Instead of having these entities subscribe to these important variables, they could be flooded within the hierarchical scope using the limited flooding mechanism.

5.2 Initiate the Limited Flooding Mechanism

Currently, the application layer in the management plane explicitly initiates a variable to be flooded using the API. It is envisioned that in the future this process will be policy driven, i.e. the (leaf) QoS broker sets and removes variables to be flooded at run time based on its current state and policy (adapt to its situation). The application layer uses the method `setFlooding` to set a variable to be flooded and the `removeFlooding` method to remove a previously set flooded variable (see Figure A.36). Whenever a variable is set to be flooded, the given modes in which the variable is to be flooded in must also be specified. A parent QoS broker could also use the methods `setFlooding` and `removeFlooding` from the interface `HierarchyChildCommon` (see Figure A.7).

A leaf QoS broker could be instructed to set a variable to be flooded within the cloud that it controls, this can either come from the application layer or from its parent QoS broker. If the flooding request is initiated by the application layer, the leaf QoS broker issues commands to its (edge) status routers that the specified variable is to be flooded within this cloud. If the request comes from its parent, the leaf QoS broker issues commands to its (edge) status routers that the specified variable is to be flooded to whatever level the parent QoS broker has specified.

A QoS broker could also be instructed to issue commands that set a variable to be flooded in the clouds that it controls. If the request originates at the application layer, the QoS broker issues a command to all its children to flood the specified variable to a level equal to the height of the QoS broker within the management hierarchy. If the request comes from the parent QoS broker, the QoS broker forwards the request to all its children (leaf) QoS brokers. The QoS broker uses the method `setFlooding` in the `HierarchyChildCommon` interface (see Figure A.7) to issue a request to its children leaf QoS brokers to flood a variable.

A (leaf) QoS broker can have multiple flooding entries for the same variable to be flooded to different levels in different modes. For instance a leaf QoS broker can specify to flood a specific variable within its cloud in mode 1, while the parent QoS broker has specified that the same variable is to be flooded to level 2 in modes 4, 5, and 7. If there is a conflict between what the application has specified and what the parent QoS has specified, the parent QoS brokers request takes precedence.

5.3 Design of Limited Flooding Mechanism

The management plane and data plane combined together provide the limited flooding mechanism. The management plane accepts commands to set up a variable to be flooded either from a management application or from its own policies. Commands are issued down in the management-hierarchy and terminate when the (edge) status routers in the data plane receive the commands. The data plane adds entries to its routing tables to perform the flooding of the specified variable.

Some properties of the limited flooding mechanism are:

- Each (edge) status router receives each event from a flooded variable on all its incoming event channels with height at the same or lower level, as the flooding height.

- Each event from a flooded variable generates as many copies as the number of event channels in the hierarchical scope that the variable is flooded.

- The number of dropped copies of each of the events from a flooded variable is given by:

$$Dropped_{ev} = EC\ in\ hierarchical\ scope - (E)SR\ in\ hierarchical\ scope \tag{5.1}$$

- The (edge) status router with the largest delay from the time it first receives a specific event from a flooded variable to the last copy of the event it receives, is given by:

$$(E)SR_{MinMax} = MIN\_PATH_{(publisher,(E)SR)}\ and\ MAX\_PATH_{(publisher,(E)SR)} \tag{5.2}$$

  where, MIN_PATH is the path with minimum length (latency) from the publisher to a (edge) status router and MAX_PATH is the path with maximum length (latency) from the publisher (going through (edge) status routers that have not seen the specific event) to a (edge) status router.

- An event from a flooded variable persists in the hierarchical scope as follows:

$$Persistence_{Delay} = Largest\ MAX\_PATH \tag{5.3}$$

The design of the limited flooding mechanism in the management plane is presented, followed by the design in the data plane. Finally, a discussion of how the design satisfies the requirements for the limited flooding mechanism is presented.

5.3.1 Design in the Management Plane

A (leaf) QoS broker is instructed to set or remove (a previously set flooded variable) either by its parent QoS broker or by the application layer. The (leaf) QoS broker stores its state in the `StateHolderQoSBase` class (see Figure 3.13), which also stores the flooded variables. Each flooded variable is stored in an instance of the `FloodingHolder` class (label 11 in Figure 3.13). The levels that a variable is to be flooded in for each of the operating modes are stored in this holder.

When the (leaf) QoS broker receives a command to flood a variable that is currently not flooded, a new instance of a `FloodingHolder` is created and added to the list of flooded variables in the `StateHolderQoSBase` class. If the command involves an already flooded variable, but in another operating mode then the holder for this flooded variable is updated.

When a command to remove a flooded variable from specific modes results in that the variable is not flooded in any operating modes, then the `FloodingHolder` is removed from the list of flooded variables in the `StateHolderQoSBase` class. Again, commands are issued to the (edge) status routers or children (leaf) QoS brokers to remove the flooding entry.

5.3.2 Design in the Data Plane

The (edge) status routers receive commands from their leaf QoS broker to set or remove (a previously set) flooded variable in specific operating modes. In Figure 4.2, the routing table for the (edge) status routers is presented. Two components of the routing table are used to provide the limited flooding mechanism. The first is the level table (`m_levelTbl[]`) in the `RoutingTbl` class (label 0) and the second is the `ModeHolder` class (label 3), and to be more specific, the two member variables `m_flooding` and `m_lastFlood`.

Each entry in the level table array is a list of references to event channels for the specific level. For instance, `m_levelTbl[1]` contains a list of all the event channels that connect to other (edge) status routers in the same cloud. Similarly, the `m_levelTbl[2]` contains the list of event channels that are of level 2, i.e. the status routers belong to two different leaf QoS brokers that are controlled by the same QoS broker. When a status event is to be flooded to level 2, the routing algorithm loops through the list of event channels at entries `m_levelTbl[1]` and `m_levelTbl[2]` and places the status event into the output queues of each of these event channels.

Each status variable that is to be routed through a (edge) status router has a `RoutingEntry` (label 2) in the `RoutingTbl`. The `RoutingEntry` has an array of `ModeHolders` where there is an entry for each operating mode that this variable is to be routed. For each mode that the variable is to be flooded, the member variable `m_flooding` is set to the value of the level of the flooding. For instance, if the variable is not flooded in a mode then the `m_flooding` is set to 0, but if it is to be flooded to level 2 then the `m_flooding` is set to 2. The second member variable, `m_lastFlood`, is used to store the timestamp of the last event that was flooded for this variable in this mode. The reason for storing the timestamp is that only one copy of each status event should be flooded and the timestamp is used to make sure that only status events with larger timestamp than the one stored is forwarded.

Figure 5.1 presents part of the member method `routeEvent` of the `RoutingTbl` class. The method first extracts the variableId and timestamp from the received event. Then, the routing table is used to find the `RoutingEntry` and the `ModeHolder` for the variable. After that, it is tested whether or not this variable is currently set to be flooded and if so it is checked if this is the first copy of the event to be flooded. If it is not the first copy of the event, then it is simply

```
public boolean routeEvent(EventHolder event, SocketAddress fromAdr)
{
  RoutingEntry entry;
  ModeHolder modeHolder;
  int variableId = event.m_event[0].getInt(Constants.EVENT_VARIABLE_ID_OFFSET);
  long timeStamp = event.m_event[0].getLong(Constants.EVENT_CREATED_OFFSET);

  // Do we route this variable ?
  if ((entry = (RoutingEntry)this.m_tbl.get(variableId)) == null)
    return false;

  // Do we route this variable in the current operating mode ?
  if ((modeHolder = entry.m_modeTbl[this.m_currentMode]) == null)
    return false;

  // Do we flood the event ?
  if (modeHolder.m_flooding > 0)
  {
    // Have we already seen this event ?
    if (modeHolder.testAndSetFlooded(timeStamp) == false)
      return false;

    // Flood this event to the set level
    for (int i = 1; i <= modeHolder.m_flooding; ++i)
    {
      if (this.m_levelTbl[i] == null)
        continue;

      for (int j = 0; j < this.m_levelTbl[i].size(); ++j)
      {
        ((OutInterface)this.m_levelTbl[i].get(j)).pushAlertEvent(event, fromAdr);
      }
    }

    // Use the routing algo. to route this event for the other levels
    .........

    return true;
  }

  // Route this event
  .........
}
```

Figure 5.1: Pseudo code for the flooding part of the routing algorithm

dropped. On the other hand, if the event is to be flooded the method starts to loop through the array of lists of event channels from 1 to level, where level is set in `m_flooding`. For each list of event channels, the event is placed in the event channels outgoing event queues. The member variable `m_lastFlood` is then updated to the timestamp of the event that was just flooded. The routing algorithm continues to route the event to event channels that are at a greater level than the current flooding level. This is done to make sure that subscriptions in clouds where the event is not flooded will still receive event streams from this variable.

The design of the limited flooding mechanism has a security vulnerability in it. If a perpetrator is able to inject a false status event of a flooded variable, then the perpetrator could set the timestamp field to a large value. This malicious event would be flooded to all (edge) status routers in the hierarchical scope and all of them would update the member variable `m_lastFlood` to this large value. The consequence of this is that any subsequent events from the flooded variable would be dropped at the first hop. This occurs because of the way the flooding algorithm was designed to only forward the first copy of an event for a flooded variable.

Below are a few approaches on how to eliminate the vulnerability, but at the expense of higher processing time at the (edge) status routers:

1. At a specified time interval, the (edge) status router will go through its routing table and reset the member variable `m_lastFlood` to 0 for all the variables that are currently flooded. The overhead is that the (edge) status routers will forward two copies of an event. If the first copy was received before the reset and the second copy was received after the reset. This is not a large overhead; each of the (edge) status routers may forward two copies of a flooded variable at each reset interval. The longest time interval that a perpetrator could interrupt the flooding of a variable is the reset interval.

2. Every time that a variable is set to be flooded, the member variable `m_lastFlood` is set to 0. Whenever a received event is to be flooded it will be dropped if its timestamp is more than

x publisher interval larger than the member variable `m_lastFlood`, unless `m_lastFlood` is 0 then it will be forwarded. There is a problem with this approach and that is that if all the event channels fail for a time interval larger than *x* publisher interval, then the (edge) status router will drop all events from the flooded variable after the event channels are fixed.

3. Suppose that the timestamps in the events are accurate and from a global time source (GPS clock). Also assume that the (edge) status routers have access to the same clock. In this case, the (edge) status routers can check the event's timestamp against its local clock. If the timestamp in the event is much larger than the local time then the event is dropped.

5.3.3 Design Satisfaction of the Requirements

The four requirements that were specified for the limited flooding mechanism are met by the design.

The first two requirements stating that the events streams should not be interrupted during flooding are met. The (edge) status routers in the hierarchical scope that is going to be flooded start to flood the variable independently of each other. There is no synchronized switching between flooding and not flooding; the individual (edge) status routers receive the commands and they initiate the action. When a (edge) status router starts to flood a variable then all events from that variable are sent to all the event channels that the (edge) status router has. Hence, any event streams that are established for that variable will at least receive the events that they subscribed to (in fact they will receive all events). During the transition some (edge) status routers will flood the variable while some will route the variable. No events will be dropped during the transition; either the variable is flooded or it is routed. The same occurs for the transition from flooding to not flooding. Some (edge) status routers will flood the variable, and some will route the variable. Since the variable is either flooded or routed there is no way that any of the subscriptions will notice any interruptions in the event streams.

The third requirement that subscriptions outside the flooded hierarchical scope will not be affected by the flooding of the variable is also met. The flooding algorithm shown in Figure 5.1 first floods the variable, but then it routes the variable using the routing algorithm (presented in Chapter 4) to the event channels that have a higher level (line 36 of Figure 5.1) than what the variable was flooded to. Due to the fact that the algorithm routes the variable to the event channels at a higher level than the variable is flooded to, subscriptions outside the flooded hierarchical scope will not be affected by the flooding.

The fourth requirement that only the first copy of each status event from a flooded variable is forwarded to all the event channels is met. On line 20 in Figure 5.1 a test is performed for each event of a flooded variable to make sure that only the first copy is flooded. If a copy of the event has already been flooded, then it is simply dropped.

5.4 Examples of the Limited Flooding Mechanism

Two examples of the limited flooding mechanism are presented. The first example shows how events from a flooded variable are propagated within a cloud, while the second one explains how flooded variables propagate between clouds.

5.4.1 Propagation of Events Within a Cloud

The first example illustrates the propagation of events from a variable that is flooded between two clouds. Figure 5.2 shows the configuration used for the example. There is one Publisher (Pub p_0) connected to ESR e_0 which resides in the cloud controlled by leaf QoS broker q_2. For this example it is assumed that the latency of all the event channels is the same.

The labels on the event channels indicate in which round the events from the flooded variable traverse the EC. Each round of the progress of a single event that are being flooded will now be presented in detail.

Round 1: First copy of the event is received at SR i_0 and e_1.

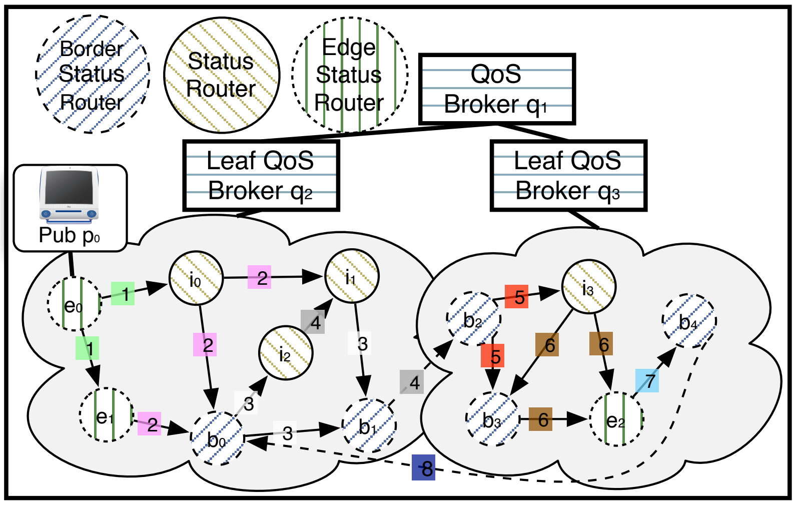


Figure 5.2: Propagation of events within a cloud

Round 2: First copy of the event is received at SR i_1 and b_0, second copy of the event is received (and dropped) at SR b_0.

Round 3: First copy of the event is received at SR i_2 and b_1, second copy of the event is received (and dropped) at SR b_1.

Round 4: First copy of the event is received at SR b_2, second copy of the event is received (and dropped) at SR i_1.

Round 5: First copy of the event is received at SR i_3 and b_3.

Round 6: First copy of the event is received at SR e_2, second copy of the event is received (and dropped) at SR b_3 and e_2.

Round 7: First copy of the event is received at SR b_4.

Round 8: Third copy of the event is received (and dropped) at SR b_0.

After round 8, the flooding of the event is terminated. During the flooding, 6 events were dropped, which is the number of event channels that are redundant. Redundant here means that the SR has

1 redundant event channel if it has 2 incoming event channels. Similarly, it has 2 redundant if it has 3 incoming event channels, and so on.

If the assumption about an equal latency for all event channels does not hold, then the propagation might be different. Assume that EC $c_{e_1-b_0}$ (event channel between SRs e_1 and b_0) and EC $c_{i_0-b_0}$ have a latency that is 14 times larger than that of the other event channels. Also assume that the flooding latency through a SR is the same as the latency of the event channels. In this case, the SR b_0 would receive its first copy of the events through EC $c_{b_4-b_0}$ and the MAX_PATH would be that from the Publisher to SR b_0.

5.4.2 *Propagation of Events Among Clouds*

The second example illustrates how flooded events propagate among a hierarchical scope of multiple clouds. Figure 5.3 shows the configuration used for the example. There are two publishers, Pub p_0 residing in cloud *A* which is controlled by leaf QoS broker q_2 and Pub p_1 residing in cloud *F* which is controlled by leaf QoS broker q_3. For this example it is assumed that within the clouds there always exists a path between any pair of status routers, the latency of all the event channels is the same, and the latency to traverse a cloud is also the same.

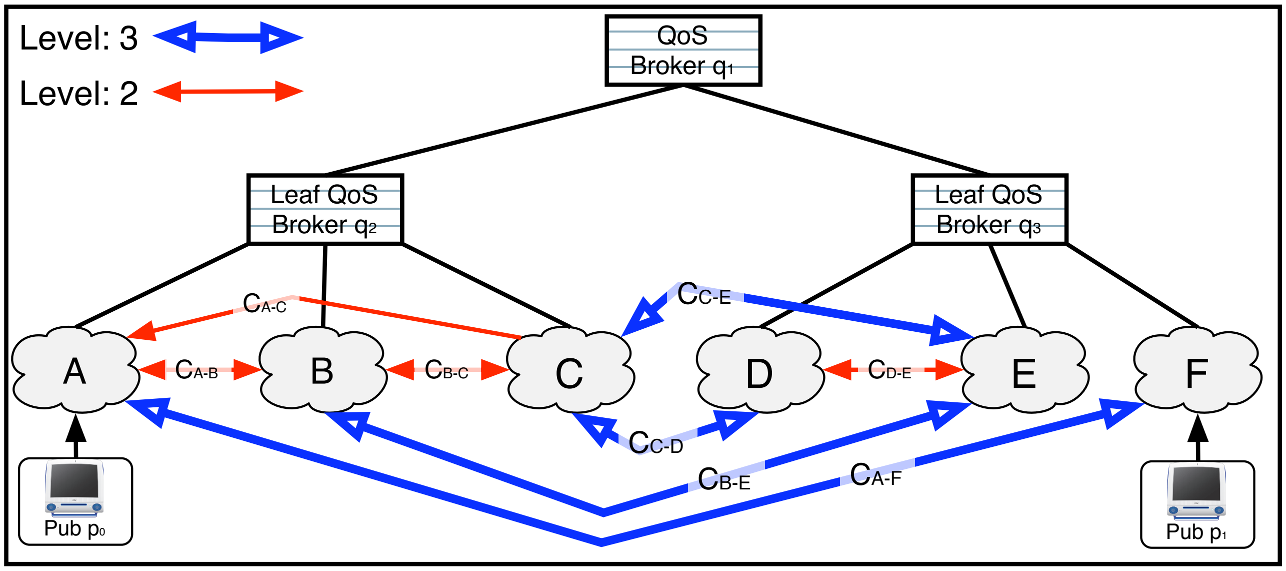


Figure 5.3: Propagation of events among clouds

Scenario 1

For the first scenario a variable that is published by Pub p_0 is set to be flooded to level 2. This means that the events from this variable will be flooded in clouds *A*, *B*, and *C*. The events are flooded as follows:

Round 1: The event is spread within cloud *A* and the event reaches the BSR that has the event channel to cloud *B*.

Round 2: First copy of the event is received at BSR in cloud *B*.

Round 3: The event is spread within cloud *B* and the event reaches the BSR that has the event channel to cloud *C*.

Round 4: First copy of the event is received at BSR in cloud *C*.

Round 5: The event is spread within cloud *C* and the event reaches the BSR that has the event channel to cloud *A*.

Round 6: Second copy of the event is received (and dropped) at BSR in cloud *A* from cloud *C*.

After round 8, the flooding of the event is terminated. During the inter-cloud flooding only 1 event was dropped.

Scenario 2

For the second scenario, a variable that is published by Pub p_1 is set to be flooded to level 2. This means that the events from this variable should be flooded in clouds *D*, *E*, and *F*. There exists a path where the events can be spread to all the clouds at level 2, but then the events have to traverse some event channels at level 3. As a result, the events will not reach clouds *E* and *F*. The events from this variable will be flooded as follows:

Round 1: The event is spread within cloud *F* and the event reaches the BSR that has the event channel to cloud *A*.

Round 2: The event will not be forwarded over the event channel to cloud *A*, because that is at level 3.

After round 1 the flooding of the event is terminated.

Scenario 3

For the third scenario, a variable that is published by Pub p_1 is set to be flooded to level 3. This means that the events from this variable will be flooded in clouds *A*, *B*, *C*, *D*, *E*, and *F*. The events from this variable will be flooded as follows:

Round 1: The event is spread within cloud *F* and the event reaches the BSR that has the event channel to cloud *A*.

Round 2: First copy of the event is received at BSR in cloud *A*.

Round 3: The event is spread within cloud *A* and the event reaches the BSR that has the event channel to cloud *B*.

Round 4: First copy of the event is received at BSR in cloud *B*.

Round 5: The event is spread within cloud *B* and the event reaches the BSR that has the event channel to clouds *C* and *E*.

Round 6: First copy of the event is received at BSR in clouds *C* and *E*.

Round 7a: The event is spread within cloud *C* and the event reaches the BSR that has the event channel to clouds *A*, *D*, and *E*.

Round 7b: The event is spread within cloud *E* and the event reaches the BSR that has the event channel to clouds *C* and *D*.

Round 8a: First copy of the event is received at BSR in cloud *D* from cloud *E*.

Round 8b: Second copy of the event is received (and dropped) at BSR in cloud *A* from cloud *C*.

Round 8c: Second copy of the event is received (and dropped) at BSR in cloud *E* from cloud *C*.

Round 8d: Second copy of the event is received (and dropped) at BSR in cloud *C* from cloud *E*.

Round 8e: Second copy of the event is received (and dropped) at BSR in cloud *D* from cloud *C*.

Round 9: The event is spread within cloud *D* and the event reaches the BSR that has the event channel to cloud *C*.

Round 10: Third copy of the event is received (and dropped) at BSR in cloud *C* from cloud *D* (from round 9).

After round 1 the flooding of the event is terminated. During the flooding, 5 events were dropped.

CHAPTER 6

OPERATING MODES MECHANISM

For some application domains, like the electric power grid, contingencies of what to do when certain conditions occurs are known before hand. To handle a contingency different information may be required or the information may be needed with a different QoS. In other words different subscriptions or the QoS of the current subscription may change. Therefore, the GridStat framework provides the operating modes mechanism, which provides for quickly changing routing tables. Each mode is a set of subscriptions that are pre-allocated and loaded as a routing table into the data plane, and the system can rapidly change between the routing tables depending on the application or middleware infrastructure conditions.

During subscription, the subscriber specifies the modes the subscription operates in. A subscription path is allocated for each of the modes requested. The (edge) status routers maintain a routing table for each of the modes. The system always operates in one mode (except for when a mode change is in progress) and the change from one mode to another is controlled by policies in the management plane or initiated by the control application in the management plane.

The requirements for the operating modes mechanism are the following:

1. No resources such as bandwidth, status router computation resources, and queues are exhausted during the change from the current mode to a new one.

2. Subscriptions present in both the old and new modes, will not be interrupted by the change as long as the subscription intervals in the different modes are subintervals of each other. Such subscriptions will receive the events with the QoS specified in the current mode and during one of the phases of the mode change algorithm they will start receiving the event streams with the QoS of the new mode.

3. Active subscribers will be informed about the mode change before it occurs to avoid QoS

violation callbacks.

The rationale for the operating modes mechanism is presented, followed by the five phases of the mode change algorithm. The design of the mechanism is presented next and, finally, two mode change examples are discussed.

6.1 Rationale for the Operating Modes Mechanism

The benefit of the mode change mechanism is that it provides for a quick way to adapt to a new set of already allocated subscriptions, whenever operating conditions change at the application or middleware layers. For example, if the application monitors a system (like the electric power grid) that operates differently depending on the season of the year, then a subscription set for each of the different seasons is predefined. Another example involves applications that predefine a set of contingency operations for crisis situations. Whenever a situation is identified as a crisis scenario, the system quickly adapts to the contingency experienced. A final usage is observed when the middleware framework experiences anomalies and as a result the application must adapt accordingly. An example of such an anomaly is an event channel failure.

There are a numerous ways that modes could be used in an application domain:

- The information that flows through the system changes depending on an outside alert level that is mapped to the different modes. The root QoS broker issues change of mode for the entire or part of the hierarchy based on this outside alert level.

- The mode space is divided between the levels in the hierarchy. Each level is now in charge of changing modes for the subscriptions that it controls. This provides for local adaptations of the subscriptions that are controlled by the different levels.

- Each cloud can adapt their mode independently, but have to inform the other clouds that could be affected by the change of mode. The reason for change might be application-dependent or the GridStat IT infrastructure-dependent.

6.2 Initiation of the Operating Modes Mechanism

Currently, the application layer in the management plane explicitly initiates the change of the operating mode by using the API. It is envisioned that in the future this process will be policy driven. The application layer uses the method `changeMode` to initiate the mode change (see Figure A.36) whereas a parent QoS broker uses the method `changeMode` from the interface `HierarchyChildCommon` (see Figure A.7).

6.2.1 Initiation of the QoS Broker

A QoS broker could also be instructed to change the operating mode for the clouds that it controls by its application layer or by its parent QoS broker. Either way, the QoS broker initiates the mode change for all the clouds that it controls. The mode change is completed after all its children have returned successfully.

6.2.2 Initiation of the Leaf QoS Broker

Similarly, a leaf QoS broker could also be instructed to change the operating mode of the cloud that it controls and this command can either come from the application layer or from its parent QoS broker. The leaf QoS broker initiates the mode change for its cloud that it controls in five phases, which are explained below.

Phase 1 of the Mode Change

During the first phase of the mode change, the leaf QoS broker informs all the edge status routers in the cloud that it controls about the upcoming mode change by using the method `informChangeMode` in the interface `CommandLeafToESR` (see Figure A.3). When an edge status router is informed about the mode change, it informs all its subscribers by calling the method `newModePreparation` in the interface `CommandESRToSub` (see Figure A.4). In this way, as subscriptions are terminated, the subscribers are aware that this happens due to the mode change. The subscriptions that exist in both the old and the new mode will still be received at the QoS specified in the old mode during this phase. The leaf QoS broker waits for all the calls to the edge status routers to complete before starting phase 2. Incase of a failed ESR the leaf QoS broker waits until it has restarted before continuing to the next phase.

Phase 2 of the Mode Change

The second phase of the mode change requires the leaf QoS broker to issue a command to all the edge status routers in its cloud to prepare for mode change by calling the method `prepareChangeMode` in the interface `CommandLeafToESR` (see Figure A.3). When this method is executed at the edge status router, a temporary routing table is created. This routing table consists of those subscriptions from the old mode that also exist in the new mode. The subscriptions in the temporary routing table are subject to the subscription intervals of the old mode. Once the temporary routing table is created, the edge status router uses this as its routing table.

In short, phase 2 removes all subscriptions in the old mode that will not be active in the new mode. Due to the fact that subscriptions are removed, it is not possible for resources in the cloud to be exhausted during this phase of the mode changing algorithm.

Phase 3 of the Mode Change

The third phase is to change the mode of all the status routers (including border status routers) in the cloud. The leaf QoS broker issues the command `changeMode` using the interface `CommandLeafToSR` (see Figure A.3) to all the status routers in its cloud. When this method is executed at the status routers they immediately change their routing tables to those for the new mode. The status routers will still receive the event streams at the interval specified in the old mode, because the edge status routers have not completed their mode change yet. The subscribers will observe the following:

- If the subscription interval in the new mode is a smaller subinterval than that of the old mode, then the subscriber will receive the event stream with the subscription interval of the old mode.

- If the subscription interval in the new mode is a larger subinterval than that of the old mode, then the subscriber will receive the event stream with the subscription interval of the new mode.

During this phase, resources cannot be exhausted because the largest of the new and old subscription interval will be used.

Using the largest subscription interval of the two modes doesn't pose any problem for individual subscriptions, since the subscriber was informed in phase 1 about the mode change. The subscribers that get the interval of the old mode will believe that the mode change hasn't occurred yet while those in the new mode will believe that the change has already occurred. If the new mode has a larger subscription interval than the old mode, the change will occur when the first SR on the path changes its mode. When this happens, that status router will filter events according to the new mode so that any SRs further down the path will receive events conforming to the new mode. The SRs further down the path will forward all events, but would have expected more. These SRs don't keep any state as to which events they have received or not; the missing events will not be noticed by these SRs. This implies that the order of when the SRs in the cloud changes mode doesn't matter, as long as it occurs after phase 1 is completed.

There is one more issue that concerns subscriptions to a set of variables. In the case that the interval in the new mode is a larger subinterval than that in the old mode, then the interval of the new mode is used, i.e. filtering is performed by the SRs. During this phase the SRs could be divided into two sets; the first set includes those that have changed mode and the second set include those that haven't. Suppose there are two variables in a group subscription and the first variable's subscription path uses only SRs in the first set, while the subscription path for the second variable uses only SRs from the second set. The events from the first variable will be received at the interval of the new mode and the events from the second variable will be received at the interval of the old mode. This will not create any problems as the subscriber middleware is informed about the mode change, hence it will only provide sets of events at the interval of the new mode. As far as the application is concerned it is now operating in the new mode.

Phase 4 of the Mode Change

During the fourth phase of the mode change the leaf QoS broker issues a command to all its edge status routers to complete the mode change by calling the method `changeMode` in the interface `CommandLeafToESR` (see Figure A.3). When this method is executed, the edge status router switches to the routing table for the new mode. When all the edge status routers have completed the switch, the cloud operates in the new mode. Incase of a failed ESR the leaf QoS broker waits until it has restarted before continuing to the next phase.

Phase 5 of the Mode Change

The fifth and final phase to complete the mode change is to inform the subscribers in the cloud that the mode change is completed and that the cloud operates completely in the new mode. The leaf QoS broker will issue the command `commitChangeMode` using the interface `CommandLeafToESR` (see Figure A.3) to all the edge status routers in its cloud. When an edge status router executes this method, it calls the method `newMode` in the interface `CommandESRToSub` (see Figure A.4) at all the subscribers that are connected to it. This informs the subscribers that the mode change is completed and that it should expect to receive all the subscriptions in the new mode at the QoS specified for this mode.

6.3 Design of the Operating Modes Mechanism

The management plane and data plane are combined to provide the operating modes mechanism. The management plane accepts commands to change the mode and these commands are propagated down in the management hierarchy to reach the leaf QoS brokers that go through the five phases to change the mode in their clouds. In the data plane the edge status routers perform phases 1, 2, 4, and 5 and the status routers perform phase 3.

The design of the mode change mechanisms in the management plane is presented, followed by the design in the data plane. Finally, it is explained how the design satisfies the requirements for the operating modes mechanism.

6.3.1 Design in the Management Plane

The (leaf) QoS broker stores its state in the `StateHolderQoSBase` class (see Figure 3.13) and the current operating mode of the (leaf) QoS broker is stored as the member variable `m_currentMode` of this class. When the (leaf) QoS broker is issued a command to change mode it first checks if it is currently operating in the requested mode. If this is the case the request is ignored. If the request is for a different mode than the current one, the QoS broker loops through the list of all its children and informs them to change mode. This delegation iterates down the management tree until the leaf QoS brokers are reached. A leaf QoS broker performs the five phases outlined earlier to complete the mode change of the cloud that it controls. After the QoS broker has delegated the change of mode to its children, it updates the `m_currentMode` member variable to the new mode. The leaf QoS broker does the same after it has completed the five phases. A (leaf) QoS broker operates in the new mode until the next mode change.

6.3.2 Design in the Data Plane

The mode change mechanism in the data plane is fairly simple and involves the routing table (see Figure 4.2). Each `RoutingEntry` has a table of 16 entries (edge status routers has 17 entries), one for each mode. A subscription is added to each of the `m_(close|sparse)Holder` in the `RoutingHolders` for the modes that the subscription is set to operate. Each of the modes has its own `RoutingHolders`, since the subscriptions differ among the modes, but there is only one copy of the `PathHolder` for each subscription for all the modes (the different modes have a reference to this `PathHolder`).

Design in the Data Plane for the Status Router

The status routers are only part of the mode change mechanism during phase 3. In this phase all the status routers are commanded to change the mode that they are operating in. The only operation the status routers need to do in order to change mode is to set the value of the member variable `m_currentMode` in the class `RoutingTbl` (see Figure 4.2) to the new mode. Once this member variable is changed, the routing algorithm will use the new value for its lookup table, resulting in having the event streams routed according to the new mode.

Design in the Data Plane for the Edge Status Router

The edge status routers are part of the mode change mechanism during phases 1, 2, 4, and 5. Phases 1 and 5 involve informing the subscribers connected to a specific edge status router about the mode change in progress and when the mode change is completed. Phases 2 and 4 involve the change of the routing table that a edge status router has.

The class `MethodsESR` (see Figure 3.12) has a member variable named `m_subscribers`, which is a list of subscribers that are connected to a specific edge status router. When a ESR is commanded by its leaf QoS broker to execute phase 1 of the mode change (`informChangeMode` in the interface `CommandLeafToESR`, Figure A.3) it loops through the list of subscribers to inform them. The same procedure occurs for phase 5; the leaf QoS broker commands the ESR to execute phase 5 and the ESR informs all its subscribers about the completion of the mode change.

When the ESR is commanded to execute phase 2 of the mode change, it first modifies its routing table as to have a 17th mode. This is done by looping through each of the entries in the `m_tbl` in the class `RoutingTbl` (see Figure 4.2) and if there is an `ModeHolder` for the old and the new mode in the routing table, then a reference to the `ModeHolder` for the old mode is inserted into the 17th entry in the `m_modeTbl[]`. Once this is done the value of the member variable `m_currentMode` in class `RoutingTbl` is changed to 16 which will use the 17th entry in the `m_modeTbl[]` during the routing of event streams.

During phase 4, a ESR does what a SR does in phase 3; it changes the value of the member variable `m_currentMode` in class `RoutingTbl` from 16 (which is the temporary mode) to the value of the new mode. Once the value has been set, the ESR will route the event streams according to the new mode.

6.3.3 Design Satisfaction of the Requirements

The design of the operating modes mechanism meets the three requirements listed at the beginning of this chapter.

The first requirement, stating that no resources should be exhausted during the change from one mode to another, is satisfied. It is assumed that the management plane will make sure that no resources are exhausted in any of the modes, i.e. over-subscriptions will not occur. If this is the case, the operating modes mechanism will not exhaust any resources during the change of mode. Phases 1 and 5 do not affect the event streams, but only inform the subscribers about the change of mode. In phase 2 subscriptions are removed, hence no exhaustion of resources is observed here either. Phase 3 can affect the subscription streams that have a larger subscription interval in the new mode and they will receive these streams at the QoS of the new mode. Subscription streams that have a smaller subscription interval will still receive these streams at the interval of the old mode, because the ESR will filter based on the QoS specified for the old mode. Since in phase 3 no subscriptions are added and the subscription interval of some of the streams may get increased (less traffic) not decreased, it is not possible for the resources in the data plane to be exhausted. Phase 4 completes the mode change, in that the subscription streams that had a larger subscription interval in the old mode than in the new mode, will now be forwarded according to the subscription interval of the new mode.

The second requirement is also satisfied. In phase 2 the event streams that are present in both modes will be forwarded according to the subscription intervals specified in the old mode. In phase 3, if the subscription interval is a larger subinterval in the new mode, then the event streams will be forwarded with the interval of the old mode until the first SR on the subscription path completes phase 3. If the subscription interval is a smaller subinterval in the new mode, then the event streams will be forwarded with the interval of the old mode during and after phase 3 has completed. In phase 4, the subscriptions with a smaller interval in the new mode will start to receive the events with the interval specified in the new mode as soon as the ESR of the publisher and the ESR of the subscriber have completed phase 4.

The third requirement is covered by phase 1 and 5. In phase 1 all active subscribers will be informed about the mode change that is to take place. Until the subscribers are informed that the mode change has completed in phase 5, event streams will stop, others will start, and QoS of event streams present in both modes might change.

6.4 Examples of the Operating Modes Mechanism

Two examples of the operating modes mechanism are presented. The first example shows how the operating mode is changed within a cloud, while the second shows how a QoS broker changes mode in the subtree that it controls.

6.4.1 Changing Mode Within a Cloud

The first example illustrates how the change of mode is performed within a cloud, i.e. it shows the interaction between the leaf QoS broker and the (E)SR. During the different phases, the subscriptions of the cloud are also affected.

Also assume that the following publication and subscriptions are present in Figure 6.1.

- Pub p_0 is publishing 2 variables named var_1 and var_2 both at an interval of 10 ms.
- Sub s_0 is subscribing to var_1 at an interval of 20 ms in mode 1 and 40 ms in mode 2. It uses the subscription path: e_0 to i_0 to e_1.
- Sub s_1 is subscribing to var_1 at an interval of 40 ms in mode 1 and 20 ms in mode 2 and var_2 at an interval of 40 ms in mode 1 only. It uses the subscription path: e_0 to i_1 to b_0 to e_2.

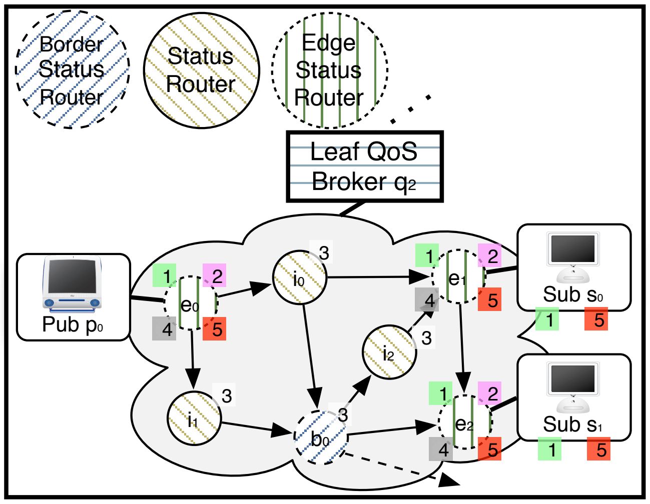


Figure 6.1: Changing mode within a cloud example

The labels on the (E)SR in Figure 6.1 indicate the different phases. Below are the five phases when the cloud changes from mode 1 to mode 2:

Phase 1: The leaf QoS broker commands all the ESR (e_0, e_1, and e_2) to perform phase 1. When each ESR receives the command, it informs the subscribers (s_0 and s_1) connected to it about the mode change.

Phase 2: The ESR creates and changes to the temporary routing table after being commanded by the LQB to do so. The result of this is that the subscription to var_2 ceases to exist.

Phase 3: The LQB informs all SRs (i_0, i_1, i_2, and b_0) to change to mode 2. As soon as i_0 changes to mode 2, events from var_1 will be received at an interval of 40 ms at s_0. As for s_1 it will still receive events from var_1 at the interval of 40 ms.

Phase 4: Again the LQB commands all the ESR to perform the mode change. s_0 will not notice any change, but changes do occur along the subscription path. After phase 3 i_0 filters every other event, because e_0 will forward events to it at an interval of 20 ms. After phase 4 e_0 will forward the events at an interval of 20 ms so i_0 does not need to do any filtering. As soon as both e_0 and e_2 have completed the phase the subscription at s_1 will be received with an interval of 20 ms.

Phase 5: For the last phase the ESRs are commanded to inform the subscribers that are connected to them that the mode change has been completed.

6.4.2 Changing Mode in the Management Plane

The second example illustrates how the change of mode is performed within the management plane, i.e. it shows the interaction between the QoS broker and the descendant (leaf) QoS brokers.

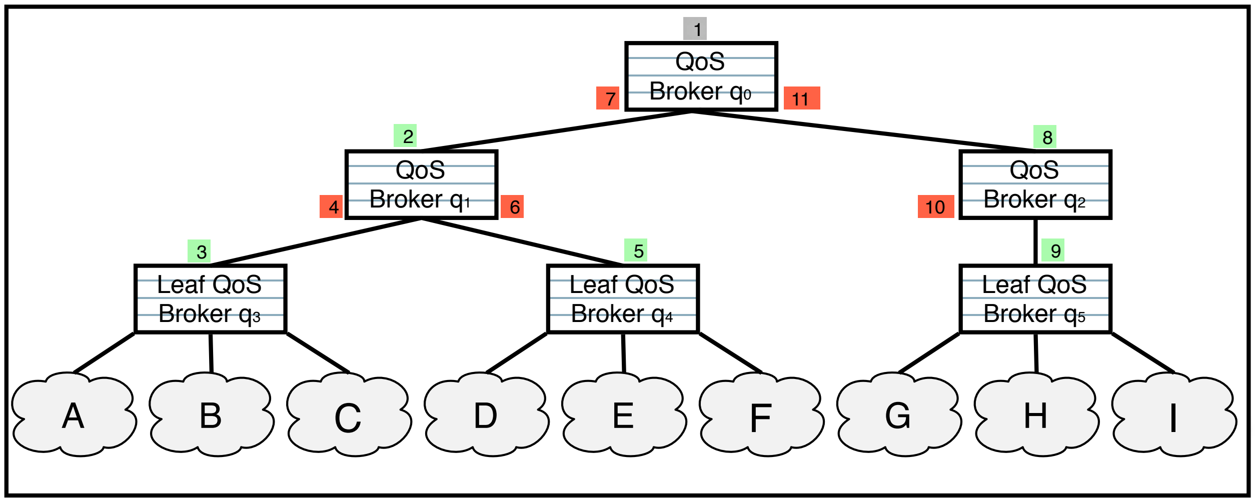


Figure 6.2: Changing mode in the management plane example

Below are the steps (call order) that are taken when a QoS broker initiates a mode change that is to be applied to all its descendants. The labels in Figure 6.2 indicate the step in the process of completing the mode change.

Step 1: The change of mode is initiated either by the application layer or by policy at QoS broker q_0.

Step 2: QoS broker q_0 commands q_1 to change mode which initiates the mode change at q_1.

Step 3: QoS broker q_1 commands q_3 to change mode which initiates the mode change at q_3.

Step 4: QoS broker q_3 has completed the mode change and return control back to q_1.

Step 5: QoS broker q_1 commands q_4 to change mode which initiates the mode change at q_4.

Step 6: QoS broker q_4 has completed the mode change and return control back to q_1.

Step 7: QoS broker q_1 has completed the mode change and return control back to q_0.

Step 8: QoS broker q_0 commands q_2 to change mode which initiates the mode change at q_2.

Step 9: QoS broker q_2 commands q_5 to change mode which initiates the mode change at q_5.

Step 10: QoS broker q_5 has completed the mode change and return control back to q_2.

Step 11: QoS broker q_2 has completed the mode change and return control back to q_0.

CHAPTER 7

CONDENSATION FUNCTION MECHANISM

In many application domains, like the electric power grid, many entities perform the same calculations on the same set of variables. Each of these entities must subscribe to the set of variables, which have to be sent from the sources to all the entities and then each of the entities performs the same calculation. If it would be possible to perform this calculation close to the sources and provide the result as a new first class variable, then resource can be saved both in the communication infrastructure and at the application layer.

The condensation function is a mechanism that provides for migrating application logic into the middleware layer. In doing so, multiple benefits are observed. The application architect is aware of the tradeoffs of migrating application logic to the middleware layer and, as a result, he/she can decide what part of the specific application may benefit from using this mechanism. The condensation function combines features from mobile code, reusable results, resource conservation, and simplification of application logic.

The rationale for the condensation function mechanism is presented, followed by the design of the mechanism. Then the creation and placement of the condensation function are presented and finally, some examples of the mechanism are described.

7.1 Rationale for the Condensation Function Mechanism

The condensation function mechanism concept was designed as an alternative to having multiple subscribers performing locally the same calculation on a given set of status variables. Each of these subscribers requests subscription paths to all the status variables and manipulates, even correlates, the corresponding status events locally. This redundant use of resources is minimized if the application architect predicts a common set of calculations that are useful for a number of subscribers. A condensation function carries out the necessary operations on the status event set for the variables and publishes the result as a status event of a newly created variable. The benefits of implementing condensation functions in midddleware include the ones listed below:

- If the condensation function is placed close to the source of the input status variables, then network resources are conserved. The resource optimizations are, with respect to event channel bandwidth and computational resources, used by status routers when forwarding status events (see Figure 7.1).
- Resources are conserved both at the middleware layer and at the application layer when multiple subscribers are interested in the output of a condensation function.
- Application logic is simplified. The application user subscribes to a single variable representing the result of the desired calculation instead of subscribing to the individual status variables and then performing the logic for the calculation.
- Even if only one subscriber is subscribing to the result of a condensation function, a substantial amount of resources could potentially be saved. Suppose the condensation function forwards an event only if the result of the calculation satisfies certain criteria defined by the application developer. Instead of disseminating all the input variables every s second, the process is reduced to sending only one event every h hour. This is a situation where the application architect can potentially decide which part of the application logic may be beneficial to move into the application layer depending on the operation and behavior of the overall system.

A strategy for using condensation functions that have not yet been investigated is to have powerful computational centers in each cloud where all the condensation functions are placed. In this case, more computational intensive application logic can be performed by these functions. The benefits of such an architecture depend heavily on the application. Some application domains can

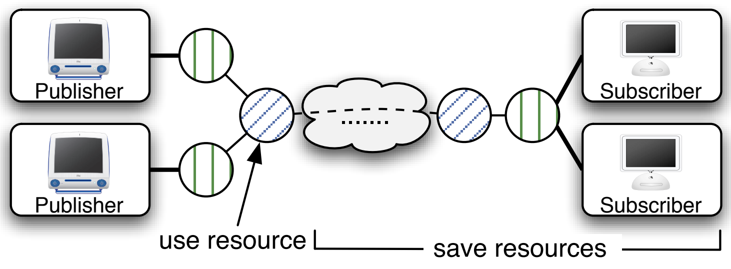


Figure 7.1: Resources saving with condensation function

be envisioned where each cloud is responsible for monitoring the overall conditions at the application layer for the cloud. This may involve some complex computation. Other clouds may need only the overall condition of their neighbor and may not be willing to spend a lot of computational resources to independently determine it. In this cooperative environment, each cloud would compute its own condition and share it with the others instead of every cloud having to do this calculation for each of the other domains that it is interested in.

7.2 Design of Condensation Function Mechanism

A condensation function consists of four modules, as shown in Figure 7.2: input filter, input trigger, calculator, and output filter, with input filter and output filter modules being optional.

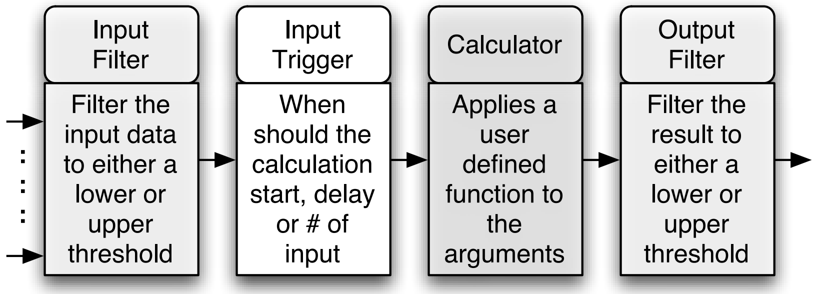

Input Filter
Filter the input data to either a lower or upper threshold
Input Trigger
When should the calculation start, delay or # of input
Calculator
Applies a user defined function to the arguments
Output Filter
Filter the result to either a lower or upper threshold

Figure 7.2: Condensation function modules

In Figure 7.3, a detailed design of the condensation function is illustrated. It includes all the Java classes that make up the condensation function and which are depicted as the `Cond module` in Figure 3.1.

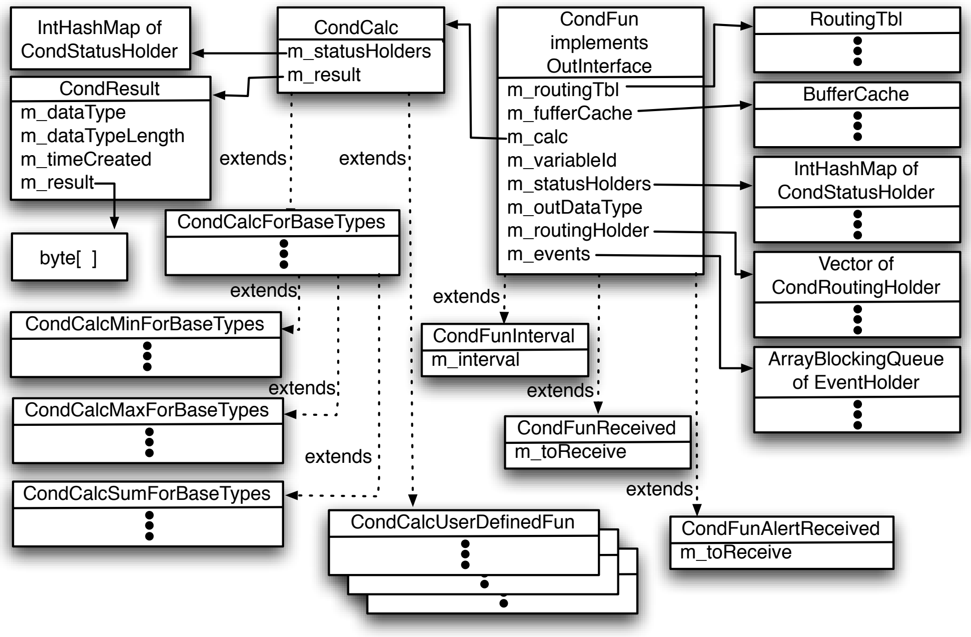


Figure 7.3: Condensation function design

7.2.1 Input Filter

The input filter module is an optional module. It filters status events from the input event streams so as to be excluded from activating the triggering mechanism (explained next). The filtering process compares the status event value with the upper and/or lower pre-defined thresholds and decides whether to drop the event or not. If the types of the status variables used in the input filter are not included in the built-in type set of the framework, then filtering objects must be provided by the developer. The user-defined filtering object is then loaded into the condensation function.

7.2.2 Trigger

The trigger module is responsible for receiving events, after filtering (if present), from the input status variables and initiating the calculation function (explained next). For each input variable, there is an entry into a data structure where the trigger module stores the latest value from each of them. This data structure is shared with the calculator module so that the calculator has access to the latest values. If any of the variables are user-defined, then unmarshalling has to precede the storage in the data structure. The second task of the trigger module is to initiate the calculator module. There are three built-in triggering mechanisms, but others can be created by extending a base class. The built-in triggering mechanisms are the ones below:

- **Time triggered:** The calculation function is triggered at a specific interval. A time triggered condensation function produces a periodic status variable.
- **Event triggered:** The calculation function is triggered when the trigger module receives new events from *x* of the variables. The result of an event triggered condensation function is either periodic or sporadic (if an input filter is present the resulting stream may become aperiodic). A periodic stream is produced if *x* is set to the number of input variables. In this case, an event is produced after the condensation function has received an event from each of the input variables, and the interval of the new event is the interval of the input variable with the largest interval. On the other hand, a sporadic resulting stream is available when setting *x* to *1*. The shortest interval, in which an event is produced, is the smallest interarrival time between any two of the input variables and the largest interval is the largest interarrival time between any two variables. This allows for a very flexible and predictable triggering mechanism.
- **Alert event triggered:** The calculation function is triggered when the trigger module receives alert (aperiodic) events from *x* of the variables. The result of an alert triggered condensation function is an aperiodic variable, where events are produced when *x* aperiodic events have been received. Again, this is a flexible mechanism that is applicable in different circumstances. The function may monitor many periodic variables and *1* alert variable. As long as the alert variable does not generate any events, the condensation function does not produce any events. When, however, an alert is received, then some calculation is performed on the periodic variables. Another example could be to monitor only aperiodic variables and

when a certain threshold (number of these variables has produced an alert) is reached, then the condensation variable produces an alert.

7.2.3 Calculator

The calculator module is provided by the developer of the condensation function. The two methods that are implemented by the developer are:

- Initialization method that allows the calculator access to the shared data structure
- Calculator method that manipulates the input variables and produces events for the condensation output variable

7.2.4 Output Filter

The last module is the optional output filter. The output filter is similar to the input filter with the difference that it filters the result produced by the calculator. If an output filter is present, the resulting stream of events can be periodic, sporadic, or aperiodic. It may be the case that the resulting stream is periodic for a while and then change to sporadic before becoming an aperiodic stream. Prior to the addition of this module, the application architect must understand the effects of filtering on the resulting stream.

The reasoning behind the support of the output filter module includes the benefits below:

- **Ease application logic:** The application developer can easily provide outgoing filtering for built-in datatypes without developing the programming logic.
- **Reusability:** For the user-defined datatypes, the output filtering objects may be reused by multiple condensation function implementations, i.e., there is one filter object per user-defined datatype with parameters to define its filtering functionality. Reuse of calculator objects is possible as well in that the functionality of the calculator object can be changed by using different output filters. An example is a calculator object that produces the average

of the input variables. If the object is used with no output filter, then a stream of events containing the average value is produced. On the contrary, the same calculator object with a filter results in an aperiodic stream of alerts when the average reaches a threshold.

7.3 Condensation Function Creation and Placement

The GridStat framework supports a GUI-based tool for specifying condensation functions. The GridStat management plane is responsible for determining the appropriate location in the data plane to load the condensation function. The specification and placement of condensation functions are now explained.

7.3.1 Defining Condensation Function

All four modules described in the previous section are defined by the condensation function tool. Starting with the input variables of either the input filter module or trigger module, if any of them have user-defined datatypes, the IDL compiler is run to generate the unmarshalling code corresponding to the new types. Similarly, if the type of the variable produced by the condensation function is user-defined, the marshalling code for the datatype needs to be generated. If the developer wants to do filtering on any user-defined datatypes, the base filtering class is extended to provide custom filtering after the unmarshalling of the received events. After the developer has all the implementation pieces necessary, the input variables are specified along with the desired input filtering, if any.

The application logic of the condensation function is captured inside the calculator object. The calculator object extends the framework's calculator base class and provides implementation for its pure virtual methods. The calculator object is associated with the condensation function by selecting its location in the file system. The shared data structure (between the input trigger and calculator) is passed as an argument to the initialization method.

The next step is to indicate the triggering mechanism, which, as mentioned earlier, is application dependent. The last task is to specify the publisher name, variable name, datatype, and publishing interval for the variable produced by the condensation function and the desired output filter, if any. This information registers a new variable that is accessible through the normal subscription process.

7.3.2 Placing Condensation Function

After successfully defining the condensation function, a request is sent to the management plane with all the necessary information that the developer specified. The management plane forwards the request to the cloud where all the input variables are located. The cloud's leaf QoS broker first checks if the variables that are used as input to the condensation function are all from publishers connected to the same edge status router. If that's the case, the function is placed on the identified edge status router. Otherwise, a brute force algorithm tests whether or not the placement of the condensation function on a candidate status router complies with the QoS requirements of all the input variables. Once the leaf QoS broker has verified that the QoS requirements requested by the condensation function are satisfied, a loading command is sent to the status router. When the leaf QoS broker has received confirmation that the condensation function has loaded successfully, it establishes all the subscription paths that the condensation function requires. Finally, the variable that is produced by the condensation function is registered.

7.4 Examples of the Condensation Function Mechanism

This section presents three condensation functions that demonstrate the benefits outlined earlier. The functionality of all three condensation functions is, with respect to a monitoring and control system, tailored for the electric power grid. The examples are greatly simplified but the reader is still able to observe the benefits of migrating some of the application logic into the middleware layer.

7.4.1 Calculate the Total

The purpose of the *calculate the total* condensation function is to sum up values produced by a number of variables and publish the result of the summation as a new variable. The potential benefit of encapsulating this operation in a condensation function is the reduction of network usage. Consider the total electric usage within a geographical area. The sensors registering the power usage within the pre-defined area serve as the input variables for the function. At the specified publishing interval, the total usage is calculated and published. Interested parties for this kind of information include different companies operating the grid, power traders, city planners, researchers in the electric power disciplines, and environmental groups.

7.4.2 Threshold Total Plus an Alert

The purpose of this condensation function is to detect unusual and potentiality damaging situations by correlating values from different variables. If such a situation occurs, an alert is generated.

Consider the major transmission lines, each one of which has a sensor that generates an alert in the case that the line fails. The failure may be the result of an accident or sabotage. If the total usage exceeds some threshold value and an alert is received from the transmission line sensors, a new alert is generated. For this scenario, the condensation function uses as inputs the result of the previous condensation function (calculate the total) and the values from all the transmission line sensors going into the monitored area.

Electric grid participants may use this alert as an input into their operational control. For example, electric power producers may have made agreements with some of the area's largest electric power consumers that, under certain conditions, the latter will reduce their power consumption in exchange for a lower utility price. The consuming companies have their controllers shutdown some of their power hungry processes by embedding in the controller a subscription to the condensation function. When the alert occurs, the process is automatically terminated (without human intervention) and as a result the reaction time to an alert is improved. Other processes that might use this kind of alerts are processes where backup power is not an option, but they still need to be shutdown gracefully. If an alert is received, then that is a sign of potential power failure. This alert would give these processes some time to do a graceful shutdown.

In addition to the interested parties listed for the previous condensation function, FEMA (Federal Emergency Management Agency) and local FBI might be interested in such alerts.

7.4.3 Find the Largest Value

The purpose of this condensation function is to find the largest value from a set of input values and publish events containing the selected value. Using the scenario presented for the first condendation function, the *find the largest value* condensation function subscribes to the same input variables. The sub-area that currently consumes the largest amount of electric energy is the output of the function. The difference between the two condensation functions is that, in this case, the data structure of the resulting variable is different from the data structure of the incoming variables. Its purpose is to find the city with the largest consumption and then produce a variable that encapsulates that variable, see Figure 7.4.

```
struct VariableBase {     struct cityUsage {       struct largestUsage {
  short variableId;         VariableBase base;       VariableBase base;
  int created;              int data;                cityUsage source;
};                        };                       };
```

Figure 7.4: Data structures for the find the largest value condensation function

The condensation function receives variables of type `cityUsage` and produces a variable of type `largestUsage`. After receiving a value from all the input variables, the condensation function calls the calculator object. The calculator finds the input variable with the largest value and places that event into the `source` field in `largestUsage`. Variable `largestUsage` is then published.

CHAPTER 8

EXPERIMENTAL RESULTS

Experiments were conducted to show the performance of the GridStat framework with respect to scalability in two dimensions namely, load on a status router and the number of hops on a subscription path. The purpose of the experiments is not to quantify the performance of the framework architecture, but rather to show the feasibility (proof of concept) of the architecture. Consequently, the results are viewed as upper bounds with the implementation prototype done in Java. For a deployment of a system like the GridStat framework we envision that the status routers would be implemented into special purpose hardware, like the Intel® IXA Network Processor [72] for the high-end routers.

The metrics below are used in order to obtain quantitative assessment of the system's performance in each dimension:

- System load (CPU): For each machine involved in a experiment run the CPU usage was measured. The *top* program was used to observe the CPU usage. These were observations (very coarse) rather than precise measurements, because the purpose was to report the trend.
- End-to-end latency of the reference systems: The duration from the time an event is created by the publisher until the time it is received by the subscriber.
- Number of events published: Each publisher recorded the number of events it published.
- Publication duration: How long the publisher was publishing events from its variables.
- Number of subscribed events received: Each subscriber recorded the number of events that it received.
- Subscription duration: How long the subscriber was subscribing to events of variables.

- Number of events received: Each SR recorded the number of events that it received on each of its incoming communication links.
- Number of events filtered: Each SR recorded the number of events that got filtered by the filtering algorithm.
- Number of events dropped: Each SR recorded the number of events it had to drop due to full outgoing buffers (did not occur during the experiments).
- Number of events sent: Each SR recorded the number of events that it sent on each of its outgoing communication links.

The bookkeeping information concerning the events and the duration of publishing and subscribing were used to derive the throughput and the percentage of events that were dropped at various places.

For all experiments, the status variables are arranged into two groups: the *load variables* group and the *reference variables* group. A load variable is published by a *load publisher* and subscribed to by a *load subscriber* and likewise, a reference variable is published by a *reference publisher* and subscribed to by a *reference subscriber*. A load publisher and its respective load subscriber peer are referred to as a *load system*, whereas a reference publisher and its reference subscriber peer are referred to as a *reference system*. Subsequently, each experiment run consists of a certain number of *reference variables* (trv) and a certain number of *load variables* (tlv). The reason for making this distinction is to use the load systems to create different load patterns on the machines involved in the experiment topology, while a reference system records the end-to-end latency for reference variables that it publishes and subscribes to. As a result, the publisher and subscriber of a reference system are located on the same physical machine so that they can share the same system clock. On the other hand, the publisher and subscriber of a load system are located on different machines, since their role is only to create load patterns for the specific experiment topology.

Both the load and reference systems record the total number of events that are published and received by the subscribers in order to calculate the total throughput in the system and detect any event loss. For all experiments, all events from the load variables are published at the same interval (*load variable interval*). The same holds for all events from the reference variables (*reference variable interval*). The load variable interval and the reference variable interval are not necessarily equal.

The remainder of this section is organized as follows. Section 8.1 describes the hardware, software, variable settings used for the experiments, and the experiment procedure. Section 8.2 presents general observations and the predicted behavior of the experiments that are conducted. Finally, Sections 8.3 – 8.7 describe and analyze the experiments that were conducted.

8.1 Experiment Setting

The experiments presented in this dissertation were conducted using the following hardware specification, software specification, and options/arguments settings:

Hardware specification:

- 19 Dell PowerEdge 1750, Dual Xeon 2.8 GHz, 533MHz system bus, 1 GB of RAM, triple 1Gb network interface.
- 1 Dell PowerEdge 2650, Dual Xeon 2.8 GHz, 533MHz system bus, 2 GB of RAM, triple 1Gb network interface.
- 4 Nortel Networks BayStack 5510-24T switch.

Software specification:

- Fedora 2 Linux distribution (kernel version 2.6.10-1.9_FC smp) on the 19 PowerEdge 1750.
- Microsoft Windows XP on the PowerEdge 2650.

- Java 2 Platform Standard Edition 5.0 (version 1.5.02) from SUN.

Java compiler options:

- -g:none: No debug information is included in the bytecode.
- -source 5: The compiler must conform to the Java 5 specification or greater.
- -target 5: The JVM that is used to run the compiled application has to conform to Java 5 specification or greater.

Java Virtual Machine (JVM) arguments:

- -server: Use the server version of the JVM.
- -jar: The program is wrapped inside a jar file.
- -Xms512m -Xmx512m: The minimum and maximum heap size for the application. By setting them to the same value, the garbage collector does not try to resize the heap at runtime[1].
- -XX:+UseParNewGC: Uses the parallel garbage collector for the young generation heap[1].
- -XX:+UseConcMarkSweepGC: Uses the Concurrent mark/sweep garbage collector for the tenured generation heap[1].

Application arguments:

- Packing variable set to 4: Each packet can potentially contain 4 status events. By conducting a few test runs with the packing variable set to 4, 8, 12, 16, it was discovered that when it was set to 4 the end-to-end latency curve was at its "tightest." The optimal setting of the packing variable for different setting needs further investigation.

[1]Set to minimize the interruption by the garbage collector

- Socket send and receive buffers size set to 128k bytes[2]: The application requests that the OS should allocate buffers of this size for the sockets used.

Machines used for the different entities:

All topologies used the following machine assignment for the different entities.

- CORBA Naming Service: Placed on the PowerEdge 2650 running Windows XP.
- Leaf QoS broker: Placed on the PowerEdge 2650 running Windows XP.
- Status routers: Each of the status routers runs exclusively on one of the PowerEdge 1750.
- Publishers and subscribers of load variables: Each of these publishers and subscribers runs exclusively on one of the PowerEdge 1750.
- Publishers and subscribers of reference variables: Both publisher and its peer subscriber run on one of the PowerEdge 1750. As it was mentioned earlier, the reason that the publishers and subscribers of the reference system run on the same machine (closed loop) is that they both have access to the same local clock. Using the same clock allows for very accurate measurements without the need for clock synchronization.

The leaf QoS broker and the naming service run on a machine using Windows XP. For the experimental setup the naming service wouldn't work on the Linux machines (reasons unknown), and that's the reason it was placed on the PowerEdge 2650 along with the leaf QoS broker. No measurements were taken for the operations that the leaf QoS broker was performing, so it was placed on the PowerEdge 2650 along with the naming service. This also gave us the opportunity to use an advanced version of the GUI that provides visualization of the various state information about the SRs. The SRs, publishers, and subscribers run on the Linux machines that were booted

[2]This was the maximum that the OS would allocate. We would have preferred to get more since we experienced drop of events at high loads.

into console multiuser mode (init level 3). This was done in order to have as little interference as possible from other processes.

Experiment procedure:

- For each new topology, the subscription set is run for 60 minutes in order to initialize the JVM of all the applications. Then all the subscriptions are terminated, but the applications are still running.
- Each experiment run lasts for 50 minutes, and this time period is divided into two phases. The first phase lasts 20 minutes and its purpose is to initialize the JVM. The second phase lasts 30 minutes and during this period the end-to-end latency is recorded at the subscribers.
- Some of the instrumentation is performed over the full 50 minute run, while others are only done during the 30 minute recording phase. The reason is that for some of the instrumentation it was not possible to create a synchronized cut between the two phases. The number of events that were dropped is recorded over both phases. As a result, it is not possible to know what portion of the dropped events occurred during the initialization phase and which was dropped (if any) during the recording phase.

Interval setting for all the experiments:

- Publication interval for a reference variable: 50 ms
- Subscription interval for a reference variable: 50 ms
- Publication interval for load variables: 1 ms. Note: this means that a load publisher publishes *x* status events (1 from each of the load variables) and then it sleeps for 1 ms. Thus, the throughput is not *x* status events / ms, but

$$\frac{x}{1 + (time\,to\,publish\,x\,events)}\,events/ms \tag{8.1}$$

- Subscription interval for load variables: 1 ms (unless otherwise specified.)

8.2 General Predictions and Observations

This section presents predictions and observations related to the experiments. Firstly, two functions are specified to predict the performance of the framework. These functions are used to compare the expected behavior of the system to the actual results. Secondly, a "mini" experiment is conducted to estimate the lower bound of the end-to-end latency of a reference system. This latency will be compared against the actual latency measurements during the different experiment runs. It is also used to set some of the parameters in the specified functions. Finally, the system load (CPU usage) of the load system machines is presented with all the load patterns that are used for the experiments. The system load on the reference system machines is not presented due to the fact that the load on these machines is extremely low.

8.2.1 Expected Behavior

In order to verify the linearly scalability we present two simplified functions that predict the behavior of the latency as the load (number of load variables) and number of hops (number of status routers) vary. A simplification is made so that the latency that is incurred by the network is not considered as a separate term, but included in that of the SR. The assumption is that each SR and its communication link to the next hop can be considered as one entity. This is a reasonable assumption, because the experiments were conducted on a switched 1Gb network, so there are plenty of network resources for each SR without any contention occurring for these resources. The throughput for the maximum load incurred during the experiments for any of the physical network links was:

Packet sized: 38 bytes (MAC/IP/UDP header) + 24 bytes (GridStat packet) = 62 bytes

*Max throughput: 496 bits * 65 000 (events in/sec) * 65 000 (events out/sec) = 61.493 Mb/s*

Therefore the simplification is made that the delay incurred by the network should be fairly constant for each of the hops, since less than 10 % of the available network resources was used for each of the communication links.

Instead of a detailed investigation of how many operations are performed by a status router to route an event (including queuing theory), a "black box" approach is taken with respect to how the function is constructed. The following abbreviations and simplifications are used in the functions:

- *rd*: reference system delay (the delay incurred by the entities depicted in Figure 8.1.)
- *er*: events routed per ms (the throughput of a SR for a given load pattern.)
- *lbd*: lower bound event delay (the minimum delay of routing an event through an SR.)
- *k*: variable depending on the operating environment (resources of the machine). As the load increases, events spend more time in buffers and queues. The time that an event, on average, spends in these buffers and queues depends on the time it takes for the machine to route events under different loads. A more detailed explanation can be found in Appendix C.
- *s*: number of SRs on a path.

Function 8.2 predicts the mean delay of a reference system and a single status router located between the edge SRs with different load patterns.

$$Mean_{delay}(ev) = rd + lbd + \frac{er}{k}\, ms \tag{8.2}$$

Function 8.3 extends Function 8.2 so that it predicts the mean delay of a reference system and a network of multiple status routers, where each can have a different load pattern.

$$Mean_{delay}(ev) = rd + \sum_{i=1}^{s}(lbd + \frac{er_i}{k})\, ms \tag{8.3}$$

After each experiment the various arguments are set accordingly to test if the experimental results conform to the predicted. If they conform, then the hypothesis that the framework scales linearly, is verified at least within the test environment. What is meant with linear scalability for the experiments is twofolded. First, as the load on a SR is increased, the delay and the variance of the delay that is incurred by an event being forwarded through this SR increments according to a linear function. Second, as the number of hops that an event traverses increases, the end-to-end delay and the variance of this delay increases according to a linear function.

8.2.2 Reference System Lower Bound End-to-End Latency Experiment

The lower bound end-to-end latency experiment was conducted to find the lower bound end-to-end latency for a reference system. For all experiment topologies, a reference system consists of at least the following components: publisher, edge SR for the publisher, subscriber, and edge SR for the subscriber. Thus, the topology for this experiment only consisted of those components (see Figure 8.1). Pub p_0 only published one reference variable that was subscribed to by Sub s_0.

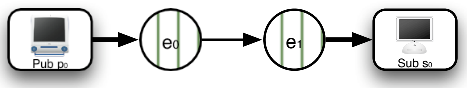


Figure 8.1: Topology for the lower bound exp.

Latency (ms)	Received (%)
0.2	0.634
0.3	14.765
0.4	52.265
0.5	30.069
0.6	2.225
0.7	0.012
0.8	0.007
0.9	0.002
1.0	0
1.1	0
1.2	0
[1.3, 1.9]	0.005
[2.0, 2.9]	0.012
[3.0, 3.3]	0.002

Table 8.1: End-to-end latency for the lower bound exp.

The result for the recording phase of the experiment is presented in Table 8.1 and depicted in Figure 8.2. There were no events dropped during the experiment run. From the result, it can be

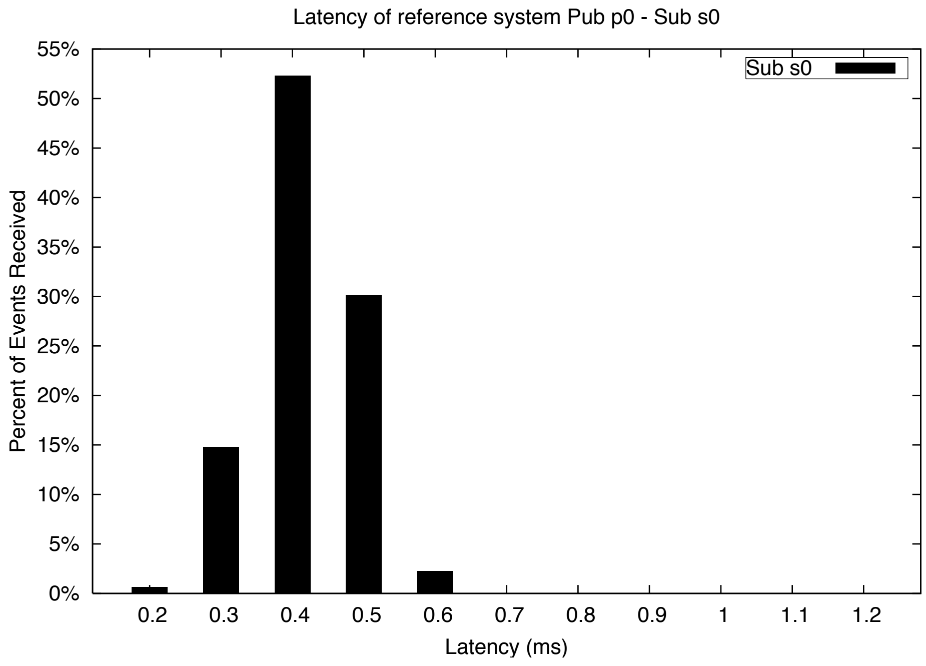


Figure 8.2: End-to-end latency for the lower bound exp.

seen that 99.959 % of the events were received with an end-to-end latency of less than 0.7 ms. The mean was 0.419 ms, with standard deviation of 0.079 ms. As it can be seen from the graphical representation of the result (Figure 8.2), the curve is bell-shaped and the reason for this shape will be explained in Section 8.2.3.

From this experiment we are able to set *rd* to 0.419 ms in Function 8.2 and subsequently in Function 8.3.

8.2.3 Explaining the Shape of the Latency Curve

The end-to-end latency curve for the reference variable is bell-shaped for all the experiments that were conducted. It would have been expected that for the lower bound experiment the end-to-end latency would be spike-shaped instead of bell-shaped, i.e. all events are received with the same delay. In this section, we will try to explain the shape of the curve by examining the implementation. In order to find the part of the framework that produces this shape of the latency curve a mini framework was developed.

This mini framework had very little functionality except to create events, forward events, receive events, and register their latency. The mini experiment consisted of three entities namely, a *mini data producer* (MDP), a *mini data forwarder* (MDF), and a *mini data consumer* (MDC). The MDP sends out 24-byte packets on a UDP socket until it is stopped. In each of these packets the first 8 bytes are used to store a timestamp. The MDF listens on a UDP socket for 24-byte packets, and when a packet is received it is sent out on another UDP socket. The MDC listens on a UDP socket for 24-byte packets. When a packet is received the timestamp is removed and the end-to-end latency is recorded. All the three entities are single threaded, compared to their counterparts in the GridStat framework.

The purpose of this mini framework was to determine if the shape of the latency curve was due to some delay incurred below the GridStat layer (OS and network), the lowest level of the GridStat data forwarding service(reading packets and writing packets to UDP sockets), or the upper layer of the GridStat forwarding service (routing, filtering, multicast, packing, queuing). The mini framework does not have the upper layer of the GridStat forwarding service. If the experiment from the mini framework does not exhibit the same shape of the latency curve, as was experienced by the GridStat framework, then this curve is due to the upper layer of the forwarding service.

Because this was discovered after the experiments were conducted, we did not have access to that testbed anymore. Another testbed was used, that also produced the same shape for the end-to-end latency of the reference system. The various settings for the experiment are as follows:

Hardware specification:

- 4 Ambient P4 2.4GHz HT, 200MHz system bus, 1GB of RAM, 100Mb network interface.
- 1 HP ProCurve Switch 2424M.

Software specification:

- Gentoo GNU/Linux distribution (kernel version 2.6.7-rc2) on all machines.

- Java 2 Platform Standard Edition 5.0 (version 1.5.02) from SUN.

Java compiler options:

- -g:none: No debug information is included in the bytecode.
- -source 5: The compiler must conform to the Java 5 specification or greater.
- -target 5: The JVM to be used to run the bytecode have to conform to the Java 5 specification or greater.

Java Virtual Machine (JVM) arguments:

- -server: Use the server version of the JVM.
- -jar: The program is wrapped inside a jar file.
- -Xms512m -Xmx512m: The minimum and maximum heap size for the application. By setting them to the same value, the garbage collector does not try to resize the heap at runtime.

Placement of the different component:

- Mini data forwarder: Each of the MDF runs exclusively on one machine.
- Mini data producer and mini data consumer: Both of them run on one of the machines in order to share the system clock.

Experiment procedure

- Each of the runs consists of one phase, which is a 30 minute recording phase.

Interval settings:

- The mini data producer creates and sends a packet every: 50 ms

Latency (ms)	Received (%)			
	0 MDF	1 MDF	2 MDF	3 MDF
0.0	99.863	0	0	0
0.1	0.113	99.875	79.23	0
0.2	0.008	0.085	20.648	98.37
0.3	0.003	0.015	0.068	1.525
0.4	0	0.003	0.028	0.04
0.5	0	0	0	0.028
[0.6, 10]	0.015	0.023	0.028	0.038

Table 8.2: End-to-end latency for the mini framework with different number of MDF

Four runs were conducted (see Table 8.2) to investigate if the mini framework implementation would also generate the bell-shaped end-to-end latency curve. In the first run, the MDP was connected directly to the MDC. As it can be seen from the results, 99.863 % of the packets was received with an end-to-end latency less than 0.1 ms. When inserting 1 MDF between the MDP and the MDC, then 99.875 % of the packets was received with end-to-end latency between $[0.1, 0.2\rangle$ms. With 2 MDF, (this is the same as the lower bound experiment presented in Section 8.2.2), 99.878 % of the packets was received with end-to-end latency between $[0.1, 0.3\rangle$ms. Finally, with 3 MDF (this is the same as the load scalability experiment presented in Section 8.3), 99.895 % of the packets was received with end-to-end latency between $[0.2, 0.4\rangle$ms.

The results of the stripped down version form a spike-shaped curve. However, for 2 and 3 MDF the packets are received in two time intervals. This is most likely an artifact of the time granularity used, i.e. for 2 MDF the events might have been received in the interval $[0.15, 0.25\rangle$ hence still a spike.

The conclusion from running these experiments is that the bell-shaped curve for the end-to-end latency of the framework implementation is an artifact of all or a subset of the following: the filtering algorithm, the event queuing/packing, the threading and synchronization model, or the behavior of the JVM when more work (filtering etc) is performed on each event routed. For the first two, namely the filtering algorithm and the queuing/packing, it is unlikely that they should create the shape since they both have a fixed number of steps to be executed for each event that is routed. The threading and synchronization mode are more likely candidates for causing the shape. This is the case because indeterminism is introduced, since each event is handled by two threads (the receiving thread and the sending thread). Both of these threads have to lock parts of the shared queue. For the low loads that were used in the lower bound experiment and also in the load scalability experiment with 0 load variables, it should not be the case that one of the two threads has to wait for the other (competition for shared resources). Therefore, we believe that the source of the shape comes from the thread context switch performed by the JVM and that this will create a bell-shaped curve of the time it takes for an event to be routed through a status router.

8.2.4 System Load (CPU) on Machines Composing Load Systems with Different Load Patterns

The system load (CPU usage) for the machines of the load systems is presented in Table 8.3. A load system consists of a publisher, the edge SR of the publisher, a subscriber and the edge SR for the subscriber. Each load publisher publishes x variables that are subscribed to by its load subscriber peer. There are two interesting points to observe from Table 8.3. Firstly, the edge SRs are consuming much more resources compared to the publishers and subscribers. Secondly, the edge SR of the subscriber consumes fewer resources than the edge SR of the publisher for the same number of load variables.

The reason, that the system load at the edge SRs is higher than that on the publishers and subscribers for the same number of load variables, has to do with the amount of work that they are each doing. The publisher and subscriber applications are single threaded and they are either creating status events that are sent out as a packet or receiving status events as packets and perform some bookkeeping. On the other hand, an edge SR is multithreaded and performs a fixed number of operations on each status event that it routes. Each communication link of an edge SR has two threads; one for receiving events and one for sending events. The receiving thread first looks up the routing table and then applies the filtering algorithm to determine if the status event should be dropped or placed into one or more of the outgoing communication link queues. The sending thread waits for events to be placed into its queue and then sends them to the next hop.

Total load variables	System load (%)			
	Load pub.	Load sub.	Edge SR of load pub.	Edge SR of load Sub.
5	0.1-0.2	1-2	1-7	1-5
10	0.1-0.2	1-3	1-10	2-8
20	0.1-0.5	3-5	10-13	5-10
40	0.1-0.5	5-7	18-21	10-15
60	0.8-1.6	7-9	24-28	12-17
80	7-10	9-11	24-34	17-22
100	7-10	11-12	24-34	17-22
120	7-10	11-12	24-34	19-22

Table 8.3: System load on machines composing a load system with different load patterns

It is our belief that the difference in system load for the publisher edge SR and the subscriber edge SR with respect to the same number of load variables is contributed to the packing of status events. Packing takes place when multiple status events are sent in one packet. The publisher never packs status events, and consequently the publisher edge SR receives each of the status events as a single packet. For each consecutive hop, status events that are destined for the same outgoing communication link are packed into one packet, provided that the receiving thread can do so without blocking the sending thread. As a result, the subscriber edge SR will receive fewer packets, each containing multiple status events, and this is most likely the reason why the load on subscriber edge SR is lower than that of a publisher edge SR.

8.3 Load Scalability of a Status Router Experiment

The purpose of this experiment is to investigate the delay of routing a status event through a status router under varying loads. The experiment topology is depicted in Figure 8.3 and consists of the following systems and variables:

- Three reference systems which consist of Pub p_0 – Sub s_0, Pub p_2 – Sub s_2, and Pub p_4 – Sub s_4.
- Two load systems which consist of Pub p_1 – Sub s_1 and Pub p_3 – Sub s_3.
- Total number of load variables (tlv) is 0, 10, 20, 40, 80, 120, 160, 200, 240 (each load system provides half of the tlv.)

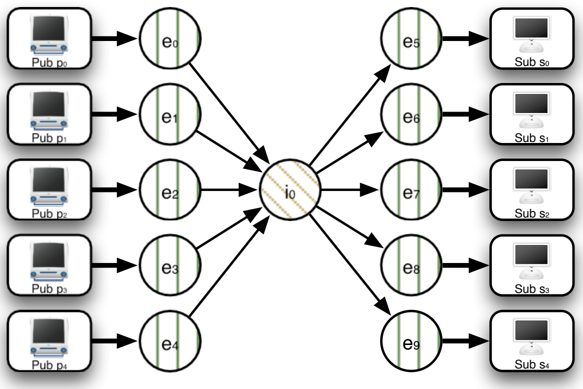


Figure 8.3: Topology for the load scalability exp.

The purpose of the reference systems is to have a very low load on the machines composing these systems and take the measurements from these three systems as the load on status router SR i_0 increases. The reason for intermixing the reference and load systems is to evaluate whether or not the placement of a reference system influences its performance.

8.3.1 Analysis of Results

Five metrics are used in this experiment: end-to-end latency, mean and standard deviation, throughput, system load, and event drop. Function 8.2 parameters are set to check how the results from the experiment compare to the predicted ones.

End-to-End Latency Analysis

The results from the three reference systems during the recording phase of the experiments are presented in Tables 8.4, 8.5, and 8.6 and depicted in Figure 8.4. As it can be observed from the results, the end-to-end latency is not affected much by the increasing load. In Table 8.7 and Figure 8.5, the mean and standard deviation for the different loads are presented. The mean is between 0.6 ms and 0.7 ms, while the standard deviation is between 0.11 ms and 0.15 ms.

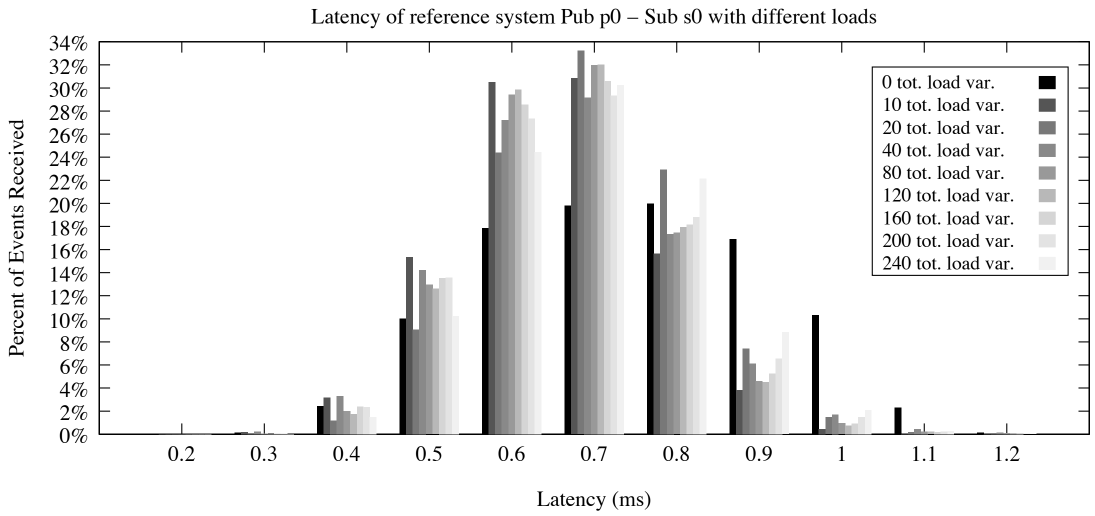
An unexpected observation is that the largest end-to-end latency and largest standard deviation occur in the absence of load variables, i.e. only the three reference systems are running. This

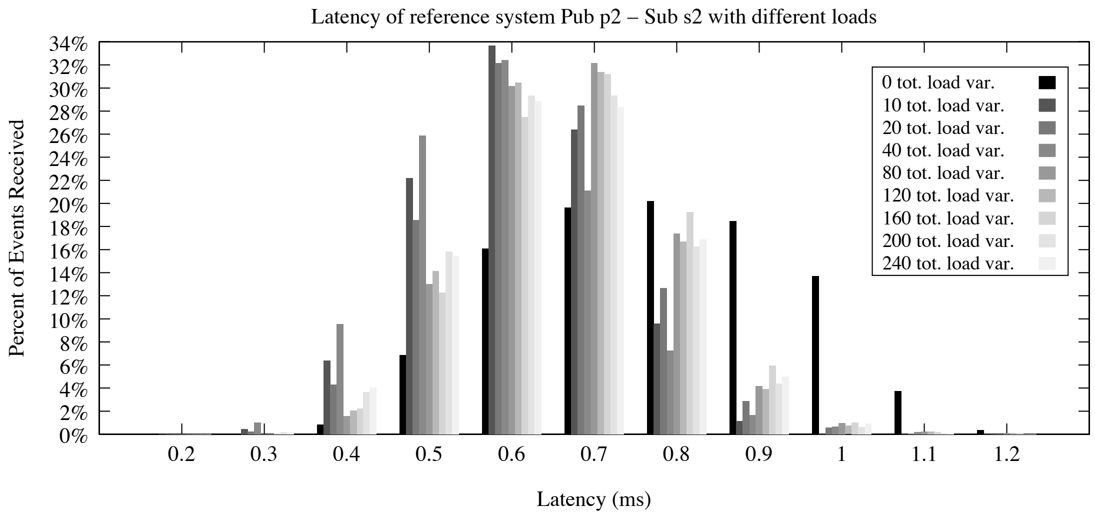
Latency (ms)	Received (%)								
	0 total load var.			10 total load var.			20 total load var.		
	Sub s_0	Sub s_2	Sub s_4	Sub s_0	Sub s_2	Sub s_4	Sub s_0	Sub s_2	Sub s_4
0.2	0	0	0	0	0	0	0	0	0
0.3	0.128	0.012	0.453	0.174	0.46	0.006	0.03	0.224	0.238
0.4	2.4251	0.835	5.135	3.162	6.382	1.276	1.198	4.333	4.972
0.5	10.024	6.859	13.982	15.359	22.207	11.451	9.066	18.549	20.532
0.6	17.876	16.109	19.085	30.506	33.683	29.9	24.404	32.159	33.487
0.7	19.804	19.624	20.069	30.846	26.424	34.429	33.227	28.471	27.522
0.8	19.998	20.21	19.381	15.641	9.609	18.806	22.928	12.662	10.956
0.9	16.926	18.481	14.241	3.818	1.145	3.81	7.428	2.883	1.912
1.0	10.332	13.728	6.458	0.011	0.043	0.283	1.476	0.569	0.289
1.1	2.317	3.736	1.134	0	0	0.006	0.177	0.073	0.031
1.2	0.16	0.377	0.055	0.003	0.003	0	0.028	0.006	0.006
[1.3,1.9]	0.009	0.012	0.003	0.011	0.02	0.0	0.008	0.017	0.011
[2.0,2.9]	0.0	0.012	0.003	0.023	0.023	0.026	0.025	0.045	0.034
[3.0,3.9]	0.003	0.006	0.0	0.003	0.003	0.006	0.003	0.008	0.008
[4.0,4.9]	0.0	0.0	0.0	0.003	0.0	0.003	0.0	0.0	0.003

Table 8.4: End-to-end latency for the load scalability exp. while varying tlv from 0 to 20

Latency (ms)	Received (%)								
	40 total load var.			80 total load var.			120 total load var.		
	Sub s_0	Sub s_2	Sub s_4	Sub s_0	Sub s_2	Sub s_4	Sub s_0	Sub s_2	Sub s_4
0.2	0	0.009	0.002	0	0	0	0	0	0
0.3	0.236	1.038	0.267	0.026	0	0.034	0.026	0.029	0.347
0.4	3.29	9.528	5.212	1.732	1.582	1.995	1.732	2.057	5.373
0.5	14.214	25.894	20.784	12.598	13.001	13.633	12.598	14.147	20.063
0.6	27.236	32.425	32.634	29.867	30.158	29.439	29.867	30.496	32.31
0.7	29.156	21.142	26.598	32.039	32.176	31.523	32.039	31.376	26.83
0.8	17.332	7.27	10.857	17.949	17.409	16.974	17.949	16.69	11.694
0.9	6.113	1.68	2.45	4.525	4.184	4.663	4.525	3.934	2.455
1.0	1.716	0.666	0.751	0.75	0.984	1.137	0.75	0.739	0.509
1.1	0.445	0.189	0.258	0.214	0.249	0.299	0.214	0.229	0.185
1.2	0.142	0.04	0.08	0.095	0.095	0.115	0.096	0.087	0.064
[1.3,1.9]	0.053	0.044	0.047	0.129	0.115	0.131	0.159	0.174	0.11
[2.0,2.9]	0.06	0.064	0.053	0.056	0.039	0.053	0.035	0.032	0.046
[3.0,3.9]	0.004	0.009	0.004	0.008	0.008	0.006	0.009	0.012	0.012
[4.0,4.9]	0.002	0.002	0.002	0.0	0.0	0.0	0.0	0.0	0.0

Table 8.5: End-to-end latency for the load scalability exp. while varying tlv from 40 to 120





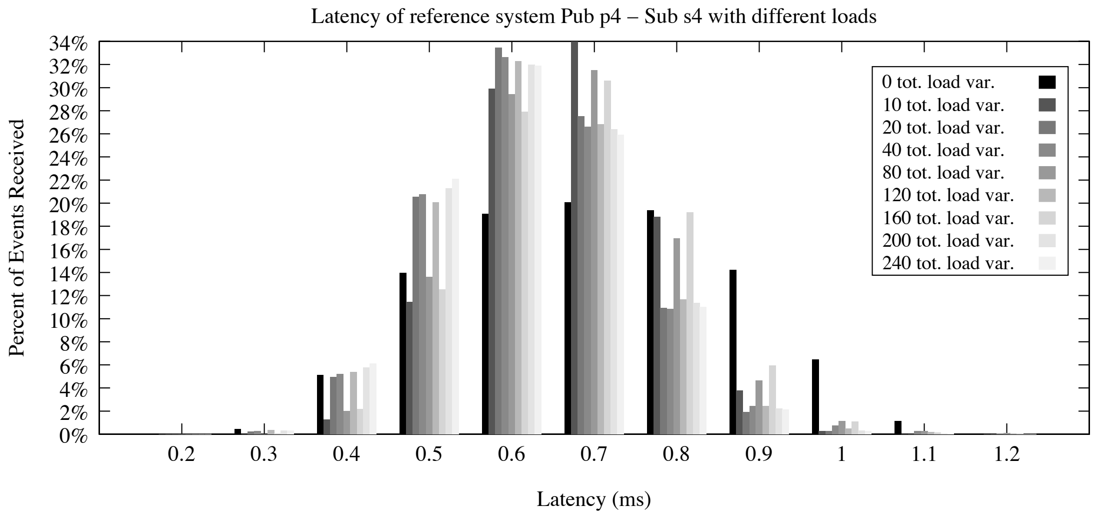


Figure 8.4: End-to-end latency for the load scalability exp.

Latency (ms)	Received (%)								
	160 total load var.			200 total load var.			240 total load var.		
	Sub s_0	Sub s_2	Sub s_4	Sub s_0	Sub s_2	Sub s_4	Sub s_0	Sub s_2	Sub s_4
0.2	0	0	0	0	0	0	0	0	0
0.3	0.062	0.076	0.073	0.069	0.218	0.33	0.025	0.203	0.315
0.4	2.378	2.227	2.192	2.358	3.641	5.764	1.507	4.054	6.139
0.5	13.554	12.23	12.548	13.58	15.849	21.293	10.235	15.446	22.09
0.6	28.556	27.502	27.932	27.35	29.331	31.982	24.466	28.855	31.903
0.7	30.622	31.211	30.599	29.347	29.377	26.418	30.245	28.365	25.913
0.8	18.178	19.256	19.22	18.83	16.262	11.394	22.159	16.857	11.009
0.9	5.248	5.971	5.97	6.572	4.406	2.243	8.829	5.019	2.128
1.0	0.925	1.029	1.096	1.48	0.631	0.327	2.104	0.906	0.27
1.1	0.171	0.188	0.138	0.224	0.109	0.063	0.265	0.121	0.062
1.2	0.082	0.082	0.053	0.034	0.023	0.04	0.034	0.028	0.028
[1.3,1.9]	0.132	0.121	0.107	0.075	0.089	0.0 86	0.079	0.084	0.09
[2.0,2.9]	0.076	0.039	0.053	0.075	0.06	0.049	0.062	0.056	0.048
[3.0,3.9]	0.014	0.008	0.014	0.006	0.003	0.003	0.003	0.003	0.003
[4.0,4.9]	0.003	0.0	0.003	0.0	0.003	0.009	0.0	0.003	0.0

Table 8.6: End-to-end latency for the load scalability exp. while varying tlv from 160 to 240

phenomenon is also observed in later experiments; whenever the load of the system is almost non-existing, it results in the largest end-to-end latency and standard deviation. We have a few educated guesses as to why this occurs. The first is that under heavier load the JVM at the SR gets optimized to perform its operations, i.e. the code that performs the operations are compiled from byte-code to native-code. However, when the load is very low this optimization does not take place, hence byte-code is used. The second reason involves the scheduling of the java process itself. When the only load that is incurred on SR i_0 originates from the reference variables, SR i_0 is scheduled to perform some processing 3 times every 50 ms. However, in the presence of numerous load variables, SR i_0 is scheduled to perform processing at a much higher rate. Due to this, the probability that SR i_0 is already scheduled to perform some processing or already runs is much greater compared to when the number of load variables is low. In this way, the reference variables might be "piggy backed" scheduled with the load variables. For future work, we intend to port the implementation of the status routers to C++. If the experiments are rerun using the C++ implementation, then we would be able to compare the two implementations. If the peculiar behavior is not observed for the C++ implementation then results observed here are most likely an artifact of the JVM runtime optimizations. On the other hand, if the same behavior is observed then a likely candidate is the

OS scheduler or some other artifacts that we have yet to discover.

Mean and Standard Deviation Analysis

Total load variables	Mean (ms)				Standard deviation (ms)		
	Sub s_0	Sub s_2	Sub s_4	Est. fun	Sub s_0	Sub s_2	Sub s_4
0	0.749	0.783	0.708	0.646	0.168	0.166	0.168
10	0.654	0.614	0.671	0.669	0.123	0.118	0.115
20	0.697	0.638	0.626	0.691	0.123	0.127	0.124
40	0.672	0.598	0.63	0.731	0.143	0.138	0.133
80	0.672	0.671	0.671	0.795	0.132	0.127	0.132
120	0.672	0.665	0.631	0.841	0.127	0.129	0.133
160	0.672	0.678	0.678	0.884	0.139	0.133	0.136
200	0.678	0.657	0.626	0.915	0.139	0.135	0.141
240	0.7	0.659	0.622	0.932	0.154	0.140	0.129

Table 8.7: Mean and standard deviation for the load scalability exp.

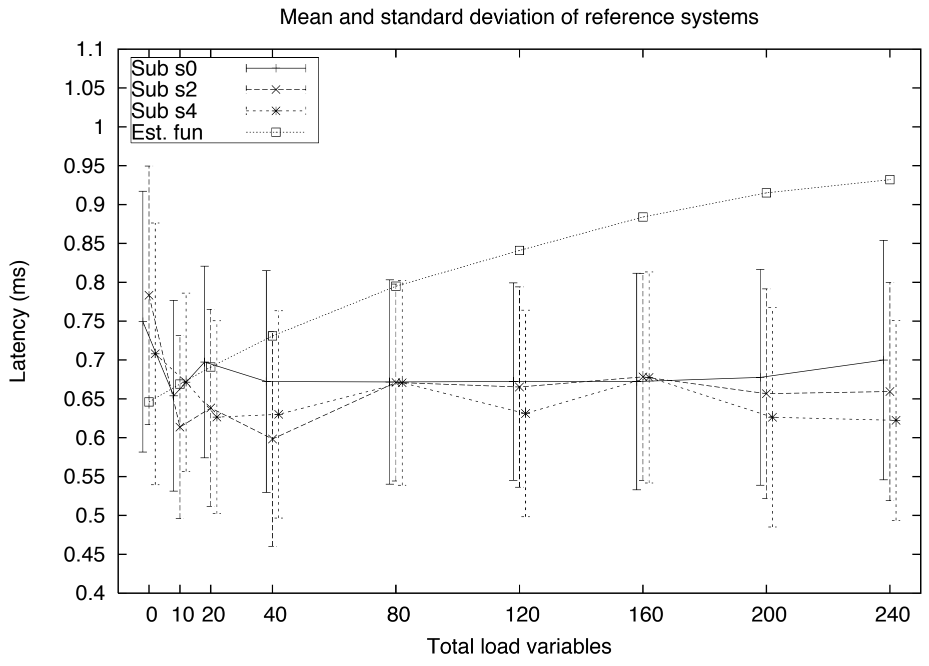


Figure 8.5: Mean and standard deviation for the load scalability exp.

The predicted performance of how the framework would scale with respect to load (Function 8.2) is now compared to the results that the experiment runs yielded. Due to the poor performance in the absence of load variables, the experiment run with 10 load variables will be used as the base case for setting the variables in Function 8.2. The variables are set as follows (explanation of the constants *k* are presented in Appendix C):

rd = 0.419 ms

lbd = Avg(0.654+0.614+0.671) - 0.419 = 0.227 ms

k = 210

Function 8.2 becomes as follows:

$$Mean_{delay}(ev) = 0.419 + 0.227 + \frac{er}{210}\ ms \tag{8.4}$$

The predicted and observed results are presented in Table 8.7 and plotted in Figure 8.5. As it can be seen, the predicted delay is much more conservative than the experimental results. The results from the experiment show that as the load increases the delay increases linearly extremely slowly, whereas the predicted linear increase has a much greater slope. Note, it might look like the estimated function produces a parabola, but this is not the case because the *x* axis represents the total load variables which is not the same as the throughput (see Function 8.1). The reason for this extremely slow increase might be that the reference systems doesn't get influenced much by the load system. This might happen because the machines that are used have dual processors with hyperthreading so four threads can execute at any given time. Also for this configuration each of the reference system streams will be received by a dedicated thread and sent out by a dedicated thread. Because of this the reference event streams will not share any of the queues with the load systems. Having realized this after the experiment a better configuration for this experiment would have been one with two SRs. For such a configuration the reference and load streams would share an outgoing queue at the first SR and share the receiving thread at the second SR.

Throughput and System Load Analysis

While the mean is not affected much by the increase of load variables, the throughput and the system load are. Table 8.8 shows the throughput (TP) at the publisher and subscriber and the

Total load variables	Throughput (event/ms)		System load (%)
	Pub $p_{(0-4)}$	Sub $s_{(0-4)}$	SR i_0
0	0.058	0.058	0.1-0.5
10	4.862	4.869	4-9
20	9.426	9.427	7-18
40	17.744	17.767	19-26
80	31.396	31.273	35-40
120	41.154	41.038	52-58
160	49.908	49.886	70-80
200	57.320	56.507	74-85
240	62.488	60.048	85-90

Table 8.8: Throughput and system load for the load scalability exp.

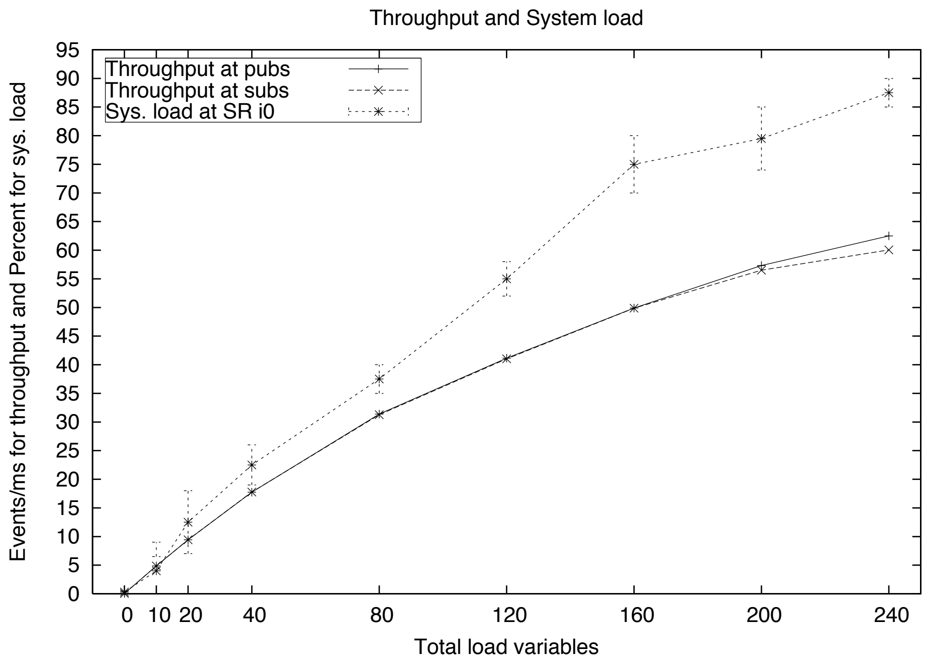


Figure 8.6: Throughput and system load for the load scalability exp.

system load at SR i_0 (depicted in Figure 8.6). The throughput at the publisher is measured over both phases of the experiment, whereas the throughput at the subscriber is only measured during the recording phase. As the number of load variables increases, the throughput does not increase linearly. The explanation of this is justified by Function 8.1. The difference in the publisher and subscriber throughput as the load is increased is due to the dropping of events. As the throughput increases the system load increases almost linearly up to 80 tlv. When tlv is greater than 80 then the system load increases faster than the throughput.

Event Drop Analysis

Total load variables	Event drop (%)			
	Pub $p_{(0-4)}$ - SR $e_{(0-4)}$	SR $e_{(0-4)}$ - SR i_0	SR i_0 - SR $e_{(5-9)}$	SR $e_{(5-9)}$ - Sub $s_{(0-4)}$
0	0	0	0	0
10	0	0	0	0
20	0	0	0	0
40	0	0	0	0
80	0	0	0	0
120	0	0.001	0	0
160	0	0.005	0	0
200	0.006	1.631	0	0
240	0.058	3.969	0	0

Table 8.9: Event drop for the load scalability exp.

Table 8.9 shows the events dropped at different path segments. The second column of Table 8.9 shows the percentage of events that are dropped between the publishers and the edge SR of the publishers. The third column presents the percentage of events that are dropped between the edge SR of the publishers and status router SR i_0. This is followed by the column of the events dropped between SR i_0 and the edge SRs of the subscribers. Finally, the last column shows the events that are dropped between the edge SRs of the subscribers and the subscribers.

The bottleneck, which causes events to be dropped, is the socket buffers (one for sending and one for receiving) that are allocated by the OS. It is possible for the application layer to suggest the size of these buffers, but the maximum was 128 kB for the testbed used. So, as the load increases, these buffers are not large enough for the reader thread to remove events from the buffers before it fills up; and as a result events are dropped. If given the opportunity to be able to change the configuration of the OS so that it allocates larger buffers for the socket it opens, then it would be possible to increase the number of load variables to observe the behavior of the SR as its processing resources are exhausted. But in the setting that we had available for the experiments, the resource that was exhausted first was the socket buffers.

As the load increases the bottleneck (with respect to event drop) is revealed to be, at SR i_0. As the system load on SR i_0 is increased above 75%, the socket buffers are not large enough and an unacceptable large portion of the events are dropped. During all the runs no events from the reference systems were dropped, only events from the load systems. This is of course due to the fact that it is the socket buffers that are exhausted at the SR and since the load incurred by the reference systems are so low its socket buffers are not exhausted.

8.4 Hop Scalability Experiment

The second experiment follows from the first in that the load on the status router stays stable (four different load patterns) while the number of status routers that the events traverse varies. What we would like to see from these experiment runs is for the framework to scale linearly with respect to the load on a status router and with respect to the number of hops a status event traverses. The purpose of this experiment is to verify that the results from the previous experiment scale linearly as status routers are added on the subscription path. The topology for this experiment is depicted in Figure 8.7 and consists of the following systems and variables:

- One reference system which consists of Pub p_0 – Sub s_0.
- Two load systems which consist of Pub p_1 – Sub s_1 and Pub p_2 – Sub s_2.
- Total number of load variables (tlv) is 40, 80, 120, 160 (each load system provides half of the tlv.)
- *s* status routers on the path from the publishers to the subscribers, where *s* is varied from 1 to 7.

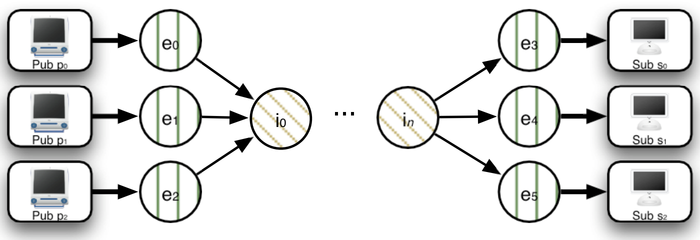


Figure 8.7: Topology for the hop scalability exp.

Similar to the load scalability experiment, the load systems in this experiment are used to create four different load patterns. The highest load pattern was chosen so that the event drop would be relatively low and the lowest load was chosen as to have a low system load (CPU usage). The two intermediate loads were then chosen to be between the high and low load patterns. Using four load patterns allows the investigation of the behavior of the framework under varying loads as SRs are added. Measurements of the end-to-end latency is taken from the reference system, while other measurements (throughput, system load, event drop) are obtained at the status routers, load systems, and the reference system.

8.4.1 *Analysis of Results*

Similar to the load scalability experiment, the metrics used are: end-to-end latency, mean and standard deviation, throughput, system load, and event drop. For this experiment, Function 8.3 is used as the prediction function for what is expected to be observed.

End-to-End Latency Analysis

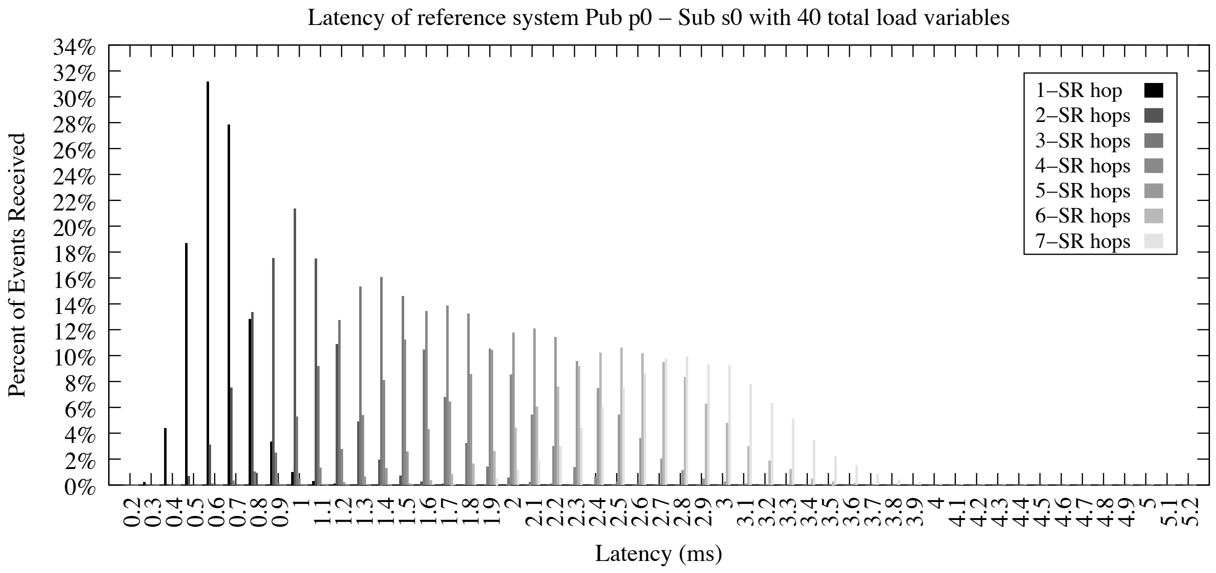
The results from the reference system during the recording phase are presented in Tables 8.10 and 8.11 and depicted in Figures 8.8 and 8.9. As it can be observed from the figures, the end-to-end latency curve changes to a wider and wider bell-shaped curve as hops are added. For 160 tlv the tail of the curve starts to get long for 6 and 7 SR hops, but still over 99 % of the events are received with a end-to-end latency of less than 7 ms.

Latency(ms)	Received (%)													
	40 tlv							80 tlv						
	1 SR	*2 SR*	*3 SR*	*4 SR*	*5 SR*	*6 SR*	*7 SR*	*1 SR*	*2 SR*	*3 SR*	*4 SR*	*5 SR*	*6 SR*	*7 SR*
0.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0.3	0.202	0.003	0	0	0	0	0	0.257	0	0	0	0	0	0
0.4	4.389	0.055	0	0	0	0	0	4.163	0.006	0	0	0	0	0
0.5	18.68	0.688	0.009	0	0	0	0	17.067	0.09	0	0	0	0	0
0.6	31.163	3.114	0.107	0	0	0	0	29.359	0.666	0.015	0	0	0	0
0.7	27.848	7.514	0.327	0.009	0	0	0	28.589	2.596	0.081	0.003	0	0	0
0.8	12.812	13.348	1.034	0.026	0	0	0	14.756	6.592	0.346	0.009	0	0	0
0.9	3.364	17.52	2.479	0.156	0	0	0	4.42	12.143	0.982	0.061	0	0	0
1	0.982	21.347	5.294	0.477	0.029	0	0	0.96	18.399	2.494	0.246	0	0	0
1.1	0.309	17.476	9.198	1.339	0.058	0	0	0.234	21.059	4.844	0.66	0.014	0	0
1.2	0.081	10.885	12.752	2.754	0.228	0	0	0.09	17.492	8.453	1.41	0.07	0	0
1.3	0.032	4.897	15.303	5.391	0.639	0.009	0	0.04	11.21	11.842	2.611	0.211	0	0
1.4	0.012	1.966	16.072	8.11	1.304	0.052	0.003	0.014	5.615	14.995	4.724	0.461	0.009	0
1.5	0.006	0.726	14.604	11.217	2.576	0.139	0.003	0.012	2.335	16.331	7.123	0.818	0.032	0
1.6	0.006	0.275	10.443	13.432	4.324	0.361	0.03	0.003	0.921	14.696	9.957	1.69	0.106	0
1.7	0	0.084	6.776	13.863	6.429	0.853	0.117	0.009	0.397	11.02	12.31	2.874	0.253	0.006
1.8	0.003	0.029	3.213	13.244	8.556	1.651	0.263	0.003	0.212	6.734	13.789	4.71	0.607	0.006
1.9	0.006	0.02	1.393	10.534	10.392	2.611	0.539	0.003	0.078	3.708	13.135	6.27	1.114	0.043
2	0.009	0.012	0.581	8.501	11.773	4.429	1.146	0	0.07	1.783	12.177	8.851	1.913	0.151
2.1	0.023	0.009	0.24	5.441	12.091	6.03	1.996	0.006	0.023	0.804	8.73	10.8	2.906	0.304
2.2	0.026	0.006	0.084	3.005	11.415	7.606	2.974	0	0.014	0.34	5.879	12.076	4.492	0.684
2.3	0.012	0.009	0.02	1.383	9.579	9.182	4.393	0.003	0.014	0.174	3.462	11.817	5.88	1.13
2.4	0.017	0.006	0.02	0.625	7.495	10.24	6.012	0.003	0.017	0.105	1.766	11.25	7.481	1.846
2.5	0	0.006	0.006	0.255	5.431	10.604	7.556	0	0.006	0.067	0.897	9.416	9.196	2.743
2.6	0.006	0	0	0.09	3.602	10.191	8.588	0.003	0.003	0.038	0.434	7.021	10.077	4.018
2.7	0.003	0.003	0.012	0.04	2.018	9.5	9.761	0	0.009	0.035	0.182	4.802	10.661	5.412
2.8	0	0	0.003	0.012	1.136	8.346	9.911	0	0.003	0.023	0.122	3.008	10.443	7.003
2.9	0.003	0	0.009	0.012	0.483	6.265	9.315	0	0.003	0.017	0.075	1.743	9.066	8.045
3	0	0	0.006	0.023	0.251	4.776	9.252	0	0.003	0.035	0.052	0.827	8.002	9.578
3.1	0.003	0	0	0.009	0.087	3.009	7.804	0	0.006	0.003	0.023	0.492	6.122	9.769
3.2	0	0	0.009	0.017	0.026	1.862	6.302	0.003	0.003	0.009	0.032	0.239	4.377	9.519
3.3	0	0.003	0.003	0.003	0.02	1.208	5.117	0	0	0.006	0.026	0.132	2.868	9.008
3.4	0.003	0	0.003	0.017	0.012	0.486	3.453	0	0.003	0.006	0.02	0.11	1.827	8.212
3.5	0	0	0	0.009	0.014	0.278	2.226	0	0	0.006	0.026	0.056	1.108	6.795
3.6	0	0	0	0.003	0.003	0.147	1.493	0	0.003	0	0.009	0.039	0.581	5.137
3.7	0	0	0.003	0	0.006	0.049	0.826	0	0	0.003	0.012	0.045	0.307	3.74
3.8	0.003	0	0	0	0.006	0.035	0.401	0	0	0	0.017	0.034	0.215	2.703
3.9	0	0	0	0	0.012	0.023	0.215	0	0	0.003	0.003	0.031	0.071	1.548
4	0	0	0	0	0	0.017	0.123	0	0	0	0.003	0.034	0.071	1.068
4.1	0	0.003	0	0	0	0.012	0.057	0	0.003	0	0.003	0.011	0.027	0.576
4.2	0	0	0	0	0.006	0.006	0.03	0	0	0.003	0.003	0.011	0.053	0.324
4.3	0	0	0	0	0	0.003	0.03	0	0	0	0.006	0.006	0.029	0.159
4.4	0	0	0	0.003	0	0.003	0.027	0	0	0	0	0.008	0.032	0.116
4.5	0	0	0	0.003	0	0.009	0.009	0	0	0	0	0.008	0.012	0.077
4.6	0	0	0	0	0	0.006	0.006	0	0	0	0.003	0.003	0.006	0.062
4.7	0	0	0	0	0	0	0.015	0	0	0	0	0	0.015	0.043
4.8	0	0	0	0	0	0	0.006	0	0	0	0	0	0.003	0.028
4.9	0	0	0	0	0	0	0.006	0	0	0	0	0	0.009	0.023
5	0	0	0	0	0	0.003	0.009	0	0	0	0	0	0	0.034
5.1	0	0	0	0	0	0	0.009	0	0	0	0	0	0.012	0.014
5.2	0	0	0	0	0	0	0.003	0	0	0	0	0	0	0.011
[5.3, 5.9]	0	0	0	0	0	0.003	0	0	0	0	0	0.008	0.006	0.043
[6.0, 6.9]	0	0	0	0	0	0	0.003	0	0	0	0.003	0.003	0.009	0.002
[7.0, 7.9]	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[8.0, 8.9]	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[9.0, 9.9]	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[10.0, 29.9]	0	0	0	0	0	0	0	0	0.003	0	0	0	0.003	0.003

Table 8.10: End-to-end latency for the hop scalability exp. with 40 and 80 total load variables

Latency(ms)	Received (%)													
	120 tlv							160 tlv						
	1 SR	*2 SR*	*3 SR*	*4 SR*	*5 SR*	*6 SR*	*7 SR*	*1 SR*	*2 SR*	*3 SR*	*4 SR*	*5 SR*	*6 SR*	*7 SR*
0.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0.3	0.062	0	0	0	0	0	0	0.02	0	0	0	0	0	0
0.4	2.11	0	0	0	0	0	0	1.432	0	0	0	0	0	0
0.5	12.296	0.023	0	0	0	0	0	9.715	0.017	0	0	0	0	0
0.6	28.019	0.325	0.006	0	0	0	0	25.224	0.266	0.003	0	0	0	0
0.7	31.426	1.392	0.046	0	0	0	0	31.695	1.222	0.014	0	0	0	0
0.8	19.084	4.399	0.162	0	0	0	0	21.995	4.342	0.061	0	0	0	0
0.9	5.522	9.456	0.561	0.006	0	0	0	7.932	9.403	0.223	0	0	0	0
1	0.943	17.518	1.537	0.017	0	0	0	1.538	16.259	0.733	0.006	0	0	0
1.1	0.24	20.571	3.54	0.106	0	0	0	0.21	18.96	1.859	0.032	0	0	0
1.2	0.13	18.034	6.522	0.305	0.006	0	0	0.075	16.331	4.454	0.136	0.003	0	0
1.3	0.034	12.445	10.302	0.859	0.02	0	0	0.035	11.596	7.916	0.263	0.009	0	0
1.4	0.034	7.481	14.122	1.744	0.055	0	0	0.029	7.165	11.826	0.66	0.026	0	0
1.5	0.011	3.985	16.11	3.468	0.237	0	0	0.032	4.304	13.841	1.448	0.061	0	0
1.6	0.028	2.101	14.556	6.109	0.6	0.003	0	0.017	2.811	13.714	3.216	0.151	0	0
1.7	0.011	1.055	11.467	8.928	1.365	0.017	0	0.009	1.754	11.249	5.48	0.456	0	0
1.8	0.003	0.466	8.031	11.505	2.716	0.081	0	0.003	1.089	8.501	8.213	0.934	0.026	0
1.9	0.008	0.223	4.924	13.617	4.505	0.33	0.003	0.003	0.878	5.769	10.851	1.503	0.044	0
2	0.008	0.141	3.384	13.847	7.157	0.729	0	0.006	0.578	4.333	12.145	3.213	0.169	0.003
2.1	0.003	0.079	1.719	12.088	8.889	1.328	0.019	0.003	0.373	2.905	11.76	5.07	0.37	0.026
2.2	0.011	0.064	1.046	8.994	10.865	2.672	0.056	0.006	0.332	2.117	10.089	7.122	0.65	0.044
2.3	0.003	0.041	0.679	6.844	11.873	4.333	0.158	0.006	0.243	1.599	7.356	9.403	1.18	0.082
2.4	0.008	0.018	0.353	4.399	11.526	6.094	0.378	0.009	0.251	1.228	5.637	10.523	1.972	0.172
2.5	0	0.018	0.168	2.592	10.889	7.442	1.034	0	0.194	1.005	4.346	10.7	3.143	0.395
2.6	0	0.015	0.116	1.612	8.681	9.108	1.912	0.003	0.13	0.666	3.04	9.415	4.612	0.64
2.7	0	0.021	0.11	0.942	6.58	9.851	3.254	0	0.124	0.588	2.383	7.874	5.934	1.254
2.8	0	0.026	0.064	0.626	4.817	10.111	4.722	0	0.104	0.507	1.772	6.286	6.98	1.947
2.9	0	0.018	0.072	0.316	3.03	9.77	6.098	0	0.098	0.461	1.39	4.765	7.726	3.099
3	0	0.015	0.055	0.241	1.986	9.426	7.834	0.003	0.101	0.446	1.158	3.802	8.142	4.648
3.1	0	0.012	0.04	0.152	1.29	7.664	8.889	0	0.092	0.295	0.932	2.687	7.359	5.586
3.2	0	0.006	0.026	0.129	0.791	5.929	9.277	0	0.09	0.339	0.741	2.255	6.464	6.671
3.3	0	0	0.023	0.092	0.534	4.613	9.403	0	0.058	0.345	0.628	1.709	5.255	7.612
3.4	0	0.009	0.026	0.055	0.329	3.092	9.398	0	0.061	0.203	0.573	1.361	4.262	7.574
3.5	0	0.012	0.026	0.037	0.254	2.097	8.619	0	0.043	0.197	0.547	1.175	3.443	7.364
3.6	0	0.003	0.023	0.04	0.139	1.49	7.148	0	0.052	0.235	0.472	0.94	2.878	6.411
3.7	0	0	0.006	0.04	0.144	0.894	5.967	0	0.058	0.226	0.478	0.81	2.435	5.426
3.8	0.003	0	0.017	0.04	0.084	0.616	4.596	0	0.029	0.174	0.382	0.749	2.013	4.651
3.9	0	0	0.014	0.026	0.087	0.425	3.19	0.003	0.043	0.2	0.397	0.676	1.835	3.4
4	0	0	0.023	0.02	0.058	0.35	2.389	0	0.035	0.171	0.347	0.667	1.547	3.081
4.1	0	0	0.012	0.011	0.04	0.194	1.529	0	0.043	0.156	0.304	0.546	1.521	2.473
4.2	0	0	0.012	0.006	0.029	0.165	0.943	0	0.04	0.162	0.313	0.572	1.325	2.111
4.3	0	0	0	0.011	0.046	0.139	0.67	0	0	0	0.266	0.456	1.311	1.897
4.4	0	0	0	0.017	0.052	0.104	0.431	0	0	0	0.281	0.467	1.323	1.754
4.5	0	0	0	0.02	0.032	0.11	0.372	0	0	0	0.229	0.435	1.352	1.453
4.6	0	0	0	0.009	0.014	0.067	0.236	0	0	0	0.208	0.342	1.296	1.353
4.7	0	0	0	0.009	0.023	0.093	0.193	0	0	0	0.185	0.374	1.317	1.286
4.8	0	0	0	0.014	0.032	0.084	0.166	0	0	0	0.159	0.316	1.445	1.228
4.9	0	0	0	0.006	0.014	0.058	0.115	0	0	0	0.179	0.241	1.241	1.143
5	0	0	0	0.006	0.032	0.067	0.096	0	0	0	0.136	0.299	1.32	1.228
5.1	0	0	0	0.009	0.014	0.049	0.08	0	0	0	0.185	0.258	1.186	1.052
5.2	0	0	0	0.009	0.014	0.058	0.043	0	0	0	0.122	0.18	1.18	1.219
[5.3, 5.9]	0	0.006	0.006	0.017	0.072	0.223	0.402	0	0.046	0.226	0.382	0.862	4.346	7.951
[6.0, 6.9]	0	0	0.003	0.04	0.049	0.09	0.276	0	0.017	0.023	0.093	0.183	1.075	3.201
[7.0, 7.9]	0	0.003	0	0.006	0.017	0.029	0.075	0	0.02	0.017	0.017	0.044	0.186	0.433
[8.0, 8.9]	0	0	0	0.003	0	0.003	0.011	0	0.009	0.003	0.003	0.015	0.017	0.058
[9.0, 9.9]	0	0.012	0.003	0.003	0.003	0	0.003	0	0	0.003	0.009	0.009	0.02	0.015
[10.0, 29.9]	0	0.003	0.006	0.006	0.012	0.006	0.016	0	0.052	0.014	0.049	0.055	0.099	0.058

Table 8.11: End-to-end latency for the hop scalability exp. with 120 and 160 total load variables



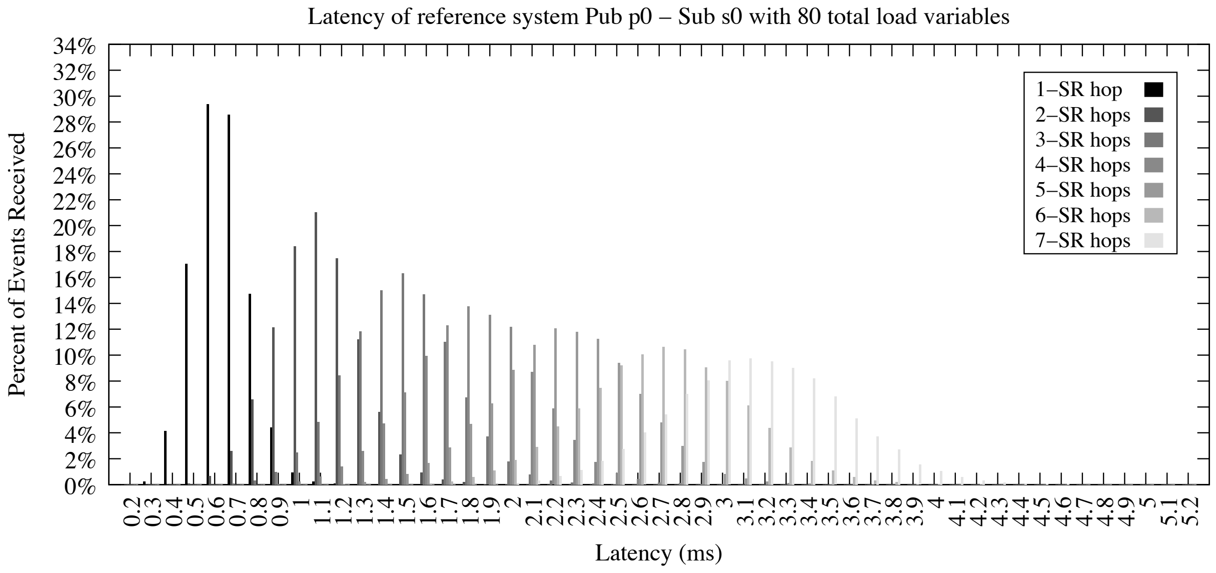
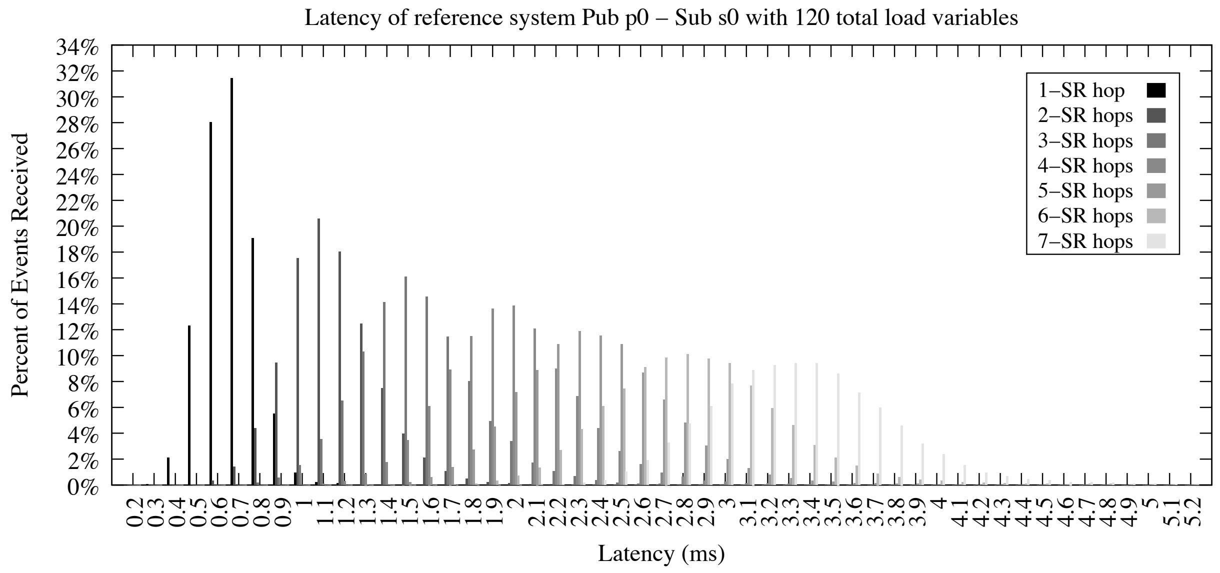


Figure 8.8: End-to-end latency for the hop scalability exp. with 40 and 80 total load variables



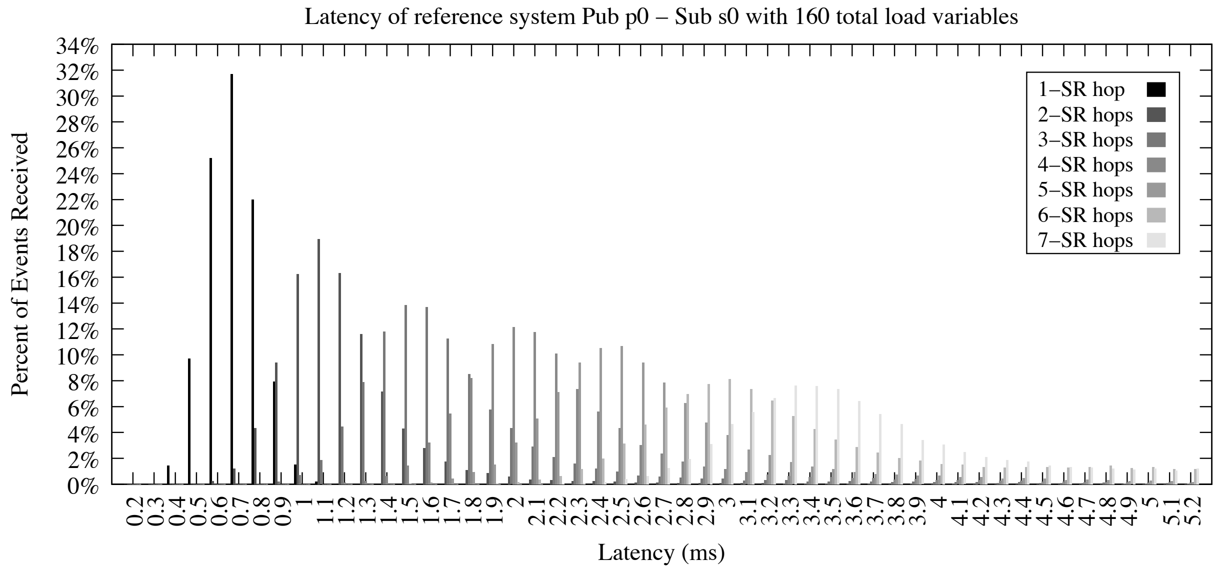


Figure 8.9: End-to-end latency for the hop scalability exp. with 120 and 160 total load variables

Mean and Standard Deviation Analysis

The mean and standard deviation for the different hops with the four load patterns are presented in Tables 8.12 and 8.13 and depicted in Figure 8.10. The mean increases almost linearly, with an average of 0.36 ms per hop for 40 tlv, 0.416 ms per hop for 80 tlv, 0.447 ms per hop for 120 tlv, and 0.532 per hop for 160 tlv (see Table 8.14).

The standard deviation also increases linearly as the number of hops increases. The largest difference occurs when the second SR is added. The reason for this is most likely due to the packing of status events. In the case of two or more SRs, all the events that are received by the first SR are all going out on the same communication link to the second SR. Therefore when there are more than two SRs, then packing occurs at the first hop which then creates a larger standard deviation. As more SRs are added the standard deviation increases linearly.

Total load variables	Mean (ms)							
	1 SR		2 SR		3 SR		4 SR	
	Sub s_0	Est. fun	Sub s_0	Est. fun	Sub s_0	Est. fun	Sub s_0	Est. fun
40	0.644	0.73	0.986	1.043	1.371	1.353	1.706	1.666
80	0.652	0.795	1.101	1.17	1.489	1.546	1.831	1.922
120	0.677	0.841	1.154	1.262	1.559	1.686	1.999	2.106
160	0.696	0.883	1.235	1.343	1.754	1.809	2.273	2.273

Total load variables	Mean (ms)					
	5 SR		6 SR		7 SR	
	Sub s_0	Est. fun	Sub s_0	Est. fun	Sub s_0	Est. fun
40	2.083	1.976	2.518	2.288	2.823	2.593
80	2.263	2.299	2.708	2.673	3.149	3.051
120	2.397	2.53	2.867	2.954	3.361	3.375
160	2.73	2.738	3.476	3.196	3.89	3.66

Table 8.12: Mean for the hop scalability exp.

Total load variables	Standard deviation (ms)						
	1 SR	2 SR	3 SR	4 SR	5 SR	6 SR	7 SR
40	0.139	0.197	0.246	0.283	0.327	0.366	0.395
80	0.133	0.217	0.259	0.302	0.343	0.388	0.416
120	0.129	0.254	0.326	0.381	0.427	0.48	0.561
160	0.13	0.641	0.616	0.725	0.788	1.035	0.958

Table 8.13: Standard deviation for the hop scalability exp.

The predicted performance of the framework with respect to the number of hops (Function 8.3)

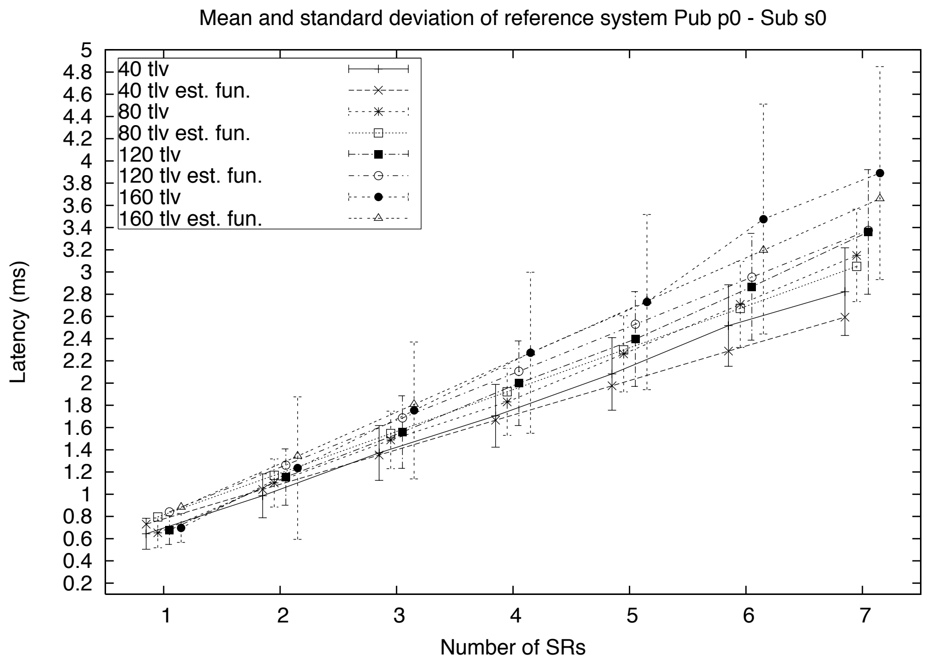


Figure 8.10: Mean and standard deviation for the hop scalability exp.

SR	Mean (ms)				Standard deviation (ms)			
	40 tlv	80 tlv	120 tlv	160 tlv	40 tlv	80 tlv	120 tlv	160 tlv
1-2	0.342	0.449	0.477	0.539	0.058	0.084	0.125	0.484
2-3	0.365	0.388	0.405	0.519	0.049	0.042	0.072	-0.025
3-4	0.335	0.342	0.44	0.519	0.037	0.043	0.055	0.109
4-5	0.377	0.432	0.398	0.457	0.044	0.041	0.046	0.063
5-6	0.435	0.445	0.47	0.746	0.039	0.0 45	0.053	0.247
6-7	0.305	0.441	0.494	0.414	0.029	0.0 28	0.081	-0.077
Avg	0.360	0.416	0.447	0.532	0.043	0.047	0.072	0.134

Table 8.14: Difference of the mean and standard dev. between the hops for the hop scalability exp.

is now compared to the results that the experiment runs yielded. The parameters that were used in Function 8.4, are also used in Function 8.3 and give the function below:

$$Mean_{eed}(ev) = 0.419 + \sum_{i=0}^{n}(0.227 + \frac{er_i}{210})\ ms \tag{8.5}$$

The predicted results are filled in Table 8.12 and plotted in Figure 8.10 along with the results from the experiment runs. According to Figure 8.10, the predicted and the experimental results agree. Taking the absolute value of the difference between the value produced by Function 8.5 and the value from the experiment runs (see Table 8.15) and summing up these differences from all the runs, yields 2.905 ms. That means 28 runs (4 (patterns) * 7 (number of SRs)) are off by 2.905 ms which in turn means that each of the run is off by an average of 0.104 ms.

Total load variables	Difference (ms)							
	1 SR	2 SR	3 SR	4 SR	5 SR	6 SR	7 SR	Sum
40	0.087	0.057	0.018	0.04	0.107	0.23	0.229	0.768
80	0.143	0.069	0.058	0.091	0.036	0.035	0.098	0.53
120	0.164	0.108	0.127	0.108	0.133	0.086	0.014	0.74
160	0.186	0.108	0.055	0	0.008	0.28	0.23	0.867
Sum	0.58	0.342	0.258	0.239	0.284	0.631	0.571	2.905

Table 8.15: Difference from predicted value and experimental value for the hop scalability exp.

Throughput and System Load Analysis

The throughput for this experiment is shown in Table 8.16. The throughput at the publishers was measured during both phases of the experiment runs, whereas at the subscriber the throughput was measured during the recording phase. The small difference between the two throughput measurements is the result of the difference in the measurements (one phase versus two phases) and also due to some event loss.

The system load is presented in Table 8.17. The system load depends on two factors: the throughput (number of subscriptions) and the number of incoming and outgoing communication links that the SR has. For s=1, the largest resource usage is observed because there is only 1 SR with three incoming and three outgoing communication links. For each communication link, the

SR needs an extra thread, and this increases thread context switching. When s > 2, there are 3 “types” of SRs: the first SR (SR i_0) which has three incoming communication links, the last SR (SR i_n) which has three outgoing communication links and the intermediate SRs (SR $i_{(1-(n-1))}$) which have only one incoming and one outgoing communication link. There are small differences between the first and last SRs, because they either receive events from three communication links or send events on three communication links. For the intermediate SRs the load is quite a lot smaller compared to the end-points (first and the last SRs.) The experiment setup might have contributed to this difference. The machines that the SRs were running on had dual CPUs with hyperthreading, so four threads could execute simultaneously. For the intermediate SRs this might have meant that they didn’t need to do much thread switching (2 threads for comm. links and 1 thread for command module), while for the end-points this most likely is not the case (4 threads for comm. links and 1 thread for command module.) When s=1, the SR i_0 has 7 threads resulting in highest usage of system resources.

Total load variables	Throughput (event/ms) at Pub $p_{(0-2)}$						
	1 SR	2 SR	3 SR	4 SR	5 SR	6 SR	7 SR
40	17.722	17.805	17.624	17.764	17.767	17.835	17.705
80	31.331	31.24	31.103	31.248	31.812	31.192	31.391
120	40.929	41.018	41.135	41.07	41.129	41.1	41.02
160	49.722	49.499	49.729	49.837	49.659	49.748	49.781

Total load variables	Throughput (event/ms) at Sub $s_{(0-2)}$						
	1 SR	2 SR	3 SR	4 SR	5 SR	6 SR	7 SR
40	17.728	17.845	17.703	17.797	17.737	17.746	17.562
80	31.278	31.155	31.227	31.263	31.309	31.207	31.295
120	40.892	40.889	41.011	40.906	40.988	41.038	41.001
160	49.68	49.391	49.607	49.673	49.707	49.537	49.568

Table 8.16: Throughput for the hop scalability exp.

Total load variables	System load (%)			
	SR i_0 (s = 1)	SR i_0 (s $>$ 1)	SR $i_{(1-(n-1))}$ (s $>$ 1)	SR i_n (s $>$ 1)
40	19-28	14-22	11-16	13-20
80	35-43	30-38	21-27	26-33
120	48-54	40-46	28-33	40-47
160	64-70	49-55	33-38	45-55

Table 8.17: System load on the SRs for the hop scalability exp.

Event Drop Analysis

The event drop for the hop scalability experiment is presented in Table 8.18. The event drop at the SRs increases as the number of SR increases. The reason for this is that the socket buffers get overloaded and therefore events are lost. The overload can occur when the JVM optimizes itself or when the operating system is performing some tasks itself. The only way for this not to occur would be to reconfigured the OS in order to allow for larger buffer space for the sockets. It is important to keep in mind that the event drop presented is the total over both phases, because it was not feasible to only measure it during the recording phase. It might be the case that most of the events dropped occurred during the first phase of the experiment since that's when the JVMs are initialized. Unfortunately, there is no way of knowing for certain.

Total load variables	Event drop (%) from pub $p_{(0-2)}$ to SR $e_{(0-2)}$						
	1 SR	*2 SR*	*3 SR*	*4 SR*	*5 SR*	*6 SR*	*7 SR*
40	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0
120	0	0	0	0	0	0	0
160	0	0	0	0	0	0	0

Total load variables	Event drop (%) from SR $e_{(0-2)}$ to SR $e_{(3-5)}$						
	1 SR	*2 SR*	*3 SR*	*4 SR*	*5 SR*	*6 SR*	*7 SR*
40	0	0	0	0	0	0	0
80	0	0.003	0	0.001	0	0.001	0.002
120	0.001	0.006	0.018	0.022	0.033	0.059	0.056
160	0.013	0.05	0.157	0.25	0.308	0.868	0.82

Total load variables	Event drop (%) from SR $e_{(3-5)}$ to Sub $s_{(0-2)}$						
	1 SR	*2 SR*	*3 SR*	*4 SR*	*5 SR*	*6 SR*	*7 SR*
40	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0
120	0	0	0	0	0	0	0
160	0	0	0	0	0	0	0

Table 8.18: Event drop for the hop scalability exp.

8.5 Multicast Mechanism Experiment

The purpose of this experiment is to investigate the performance of the multicast mechanism. This experiment topology is similar to the one used in the hop scalability experiment, with the difference that each of the SRs has a branching factor of 2 instead of 1. The topology for this experiment is depicted in Figure 8.11 and consists of the following systems and variables:

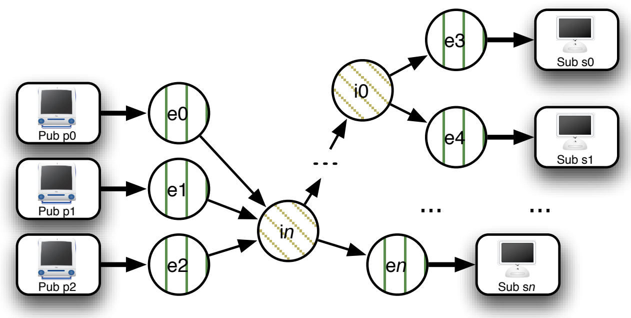


Figure 8.11: Topology for the multicast mechanism exp.

- One reference system which consists of Pub p_0 – Sub s_0.
- Two load publishers which are Pub p_1 and Pub p_2.
- *s* load subscribers, where each subscribes to Pub p_1's and Pub p_2's status variables.
- Total number of load variables (tlv) is 20, 40, 80, 120 (each load system provides half of the tlv.)
- *s* status routers on the path from the reference publisher to the reference subscriber, where *s* is from 1 to 4.

This experiment has a different load pattern compared to the previous experiments. The difference is that the load pattern of 20 total load variables is added and the pattern of 160 total load variables is removed. The reason for this change is due to the cost of providing two output streams from one input stream at each of the status routers. This cost overloads the SRs for a load pattern of 160 variables and therefore a lower pattern replaces it. The reason for the high cost of the splitting is explained later in this section. The measurements taken for this experiment are the same as the ones obtained for the previous experiments.

8.5.1 *Analysis of Results*

The results of end-to-end latency are presented and analyzed firstly. The end-to-end latency is analyzed with respect to the mean and standard deviation as the number of hops increases under the different load patterns. After the latency is analyzed, the throughput, system load, and event drop are examined. In the system load discussion, the reason for the extra cost of splitting the streams is explained based on how the SRs are implemented and the current limitation of Java.

End-to-End Latency Analysis

The results from the reference system during the recording phase of the experiment runs are presented in Tables 8.19 and 8.20 and depicted in Figures 8.12 and 8.13. As it can be seen from the results, the curve of the end-to-end latency changes to a wider and wider bell-shaped curve as hops are added. For $s = 1$ and $s = 2$ the latency curve stays relative consistent over the different load patterns. For $s = 3$ and $s = 4$ and 80 total load variables, the tail of the curves becomes very long and this is due to the fact that the system load becomes high. Also for $s = 3$ and $s = 4$ and 120 total load variables the latency curve becomes very flat. This is consistent with the system load (Table 8.25) during these experiment runs. The system load is very high (75 % - 80 %) for these runs and therefore the end-to-end latency curve becomes very flat.

Mean and Standard Deviation Analysis

In Table 8.21 and Figure 8.14, the mean and standard deviation for the different hops with the four load patterns are presented. For 20 and 40 total load variables (tlv) the mean end-to-end latency increments almost linearly as SRs are added to the path (see Table 8.23). This is not the case for 80 and 120 tlv; the increment is stable as long as s is less than 3, but when s is 3 or larger, the SRs are overloaded and the end-to-end latency increments are at a much steeper slope.

Throughput Analysis

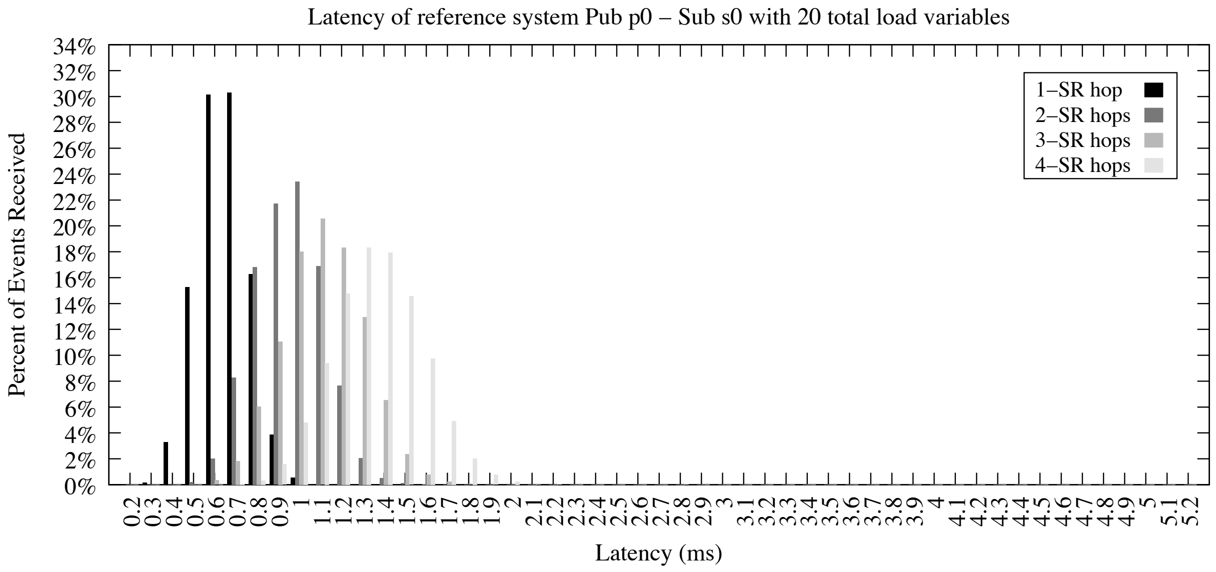
The throughput for this experiment is shown in Table 8.24. For 20 and 40 tlv, the throughput is very close to be the exact multiple of the number of load subscribers and the throughput at the

Latency (ms)	Received (%)							
	20 total load var.				40 total load var.			
	1 SR	*2 SR*	*3 SR*	*4 SR*	*1 SR*	*2 SR*	*3 SR*	*4 SR*
0.2	0	0	0	0	0	0	0	0
0.3	0.155	0	0	0	0.224	0	0	0
0.4	3.304	0	0	0	3.474	0.008	0	0
0.5	15.28	0.228	0.003	0	14.687	0.137	0	0
0.6	30.138	2.033	0.344	0.006	27.798	1.223	0.006	0
0.7	30.31	8.296	1.838	0.046	29.481	4.691	0.17	0.003
0.8	16.278	16.831	6.048	0.332	17.24	10.697	0.962	0.036
0.9	3.883	21.709	11.059	1.582	5.148	17.499	3.082	0.215
1	0.54	23.418	18.016	4.797	1.433	23.734	7.876	1.043
1.1	0.045	16.884	20.551	9.411	0.364	21.461	12.701	2.742
1.2	0.011	7.683	18.325	14.768	0.087	13.012	17.263	5.665
1.3	0.006	2.076	12.944	18.327	0.017	5.294	18.728	9.629
1.4	0.003	0.506	6.549	17.923	0.017	1.418	15.532	13.646
1.5	0.003	0.116	2.377	14.564	0	0.349	10.575	15.244
1.6	0	0.046	0.781	9.744	0.003	0.131	6.013	14.661
1.7	0	0.006	0.251	4.923	0.003	0.045	2.927	12.282
1.8	0.003	0.012	0.079	2.018	0.003	0.034	1.544	8.706
1.9	0	0	0.026	0.768	0	0.025	0.678	5.406
2	0.006	0.017	0.023	0.264	0.003	0.02	0.357	3.902
2.1	0.006	0.012	0.02	0.106	0.006	0.014	0.152	2.204
2.2	0.008	0.006	0.015	0.032	0.006	0.008	0.079	1.473
2.3	0.008	0.012	0.026	0.017	0.003	0.011	0.067	0.781
2.4	0.003	0.017	0.017	0.023	0	0.008	0.058	0.46
2.5	0	0.012	0.026	0.02	0	0.011	0.041	0.234
2.6	0	0.006	0.02	0.011	0	0.008	0.029	0.176
2.7	0	0.006	0.023	0.014	0	0.006	0.047	0.137
2.8	0	0.003	0.015	0.034	0	0.006	0.038	0.07
2.9	0	0	0.006	0.02	0.003	0	0.035	0.078
3	0.003	0.006	0.003	0.009	0	0.006	0.029	0.059
3.1	0.003	0	0.009	0.023	0	0.008	0.02	0.064
3.2	0.003	0	0.009	0.02	0	0.003	0.032	0.042
3.3	0.003	0.003	0.012	0.009	0	0	0.029	0.05
3.4	0	0.009	0.009	0.011	0	0.003	0.041	0.045
3.5	0	0	0	0.003	0	0.008	0.026	0.061
3.6	0	0.003	0.003	0.009	0	0.003	0.026	0.059
3.7	0	0.003	0.003	0.009	0	0.008	0.023	0.059
3.8	0	0.006	0.009	0.006	0	0.006	0.038	0.047
3.9	0	0	0.003	0.011	0	0.003	0.02	0.031
4	0	0	0.003	0.011	0	0.008	0.015	0.028
4.1	0	0.006	0.003	0.009	0	0.006	0.026	0.033
4.2	0	0	0.003	0.006	0	0.003	0.02	0.042
4.3	0	0	0	0.014	0	0.003	0.015	0.025
4.4	0	0	0	0.006	0	0.014	0.026	0.042
4.5	0	0	0	0.003	0	0.003	0.023	0.039
4.6	0	0	0	0.006	0	0.008	0.012	0.025
4.7	0	0	0	0.009	0	0.006	0.018	0.017
4.8	0	0	0	0.009	0	0	0.015	0.022
4.9	0	0	0	0.003	0	0.008	0.012	0.039
5	0	0	0	0.02	0	0	0.015	0.036
5.1	0	0	0	0.006	0	0.008	0.009	0.028
5.2	0	0	0	0.006	0	0.006	0.02	0.033
[5.3, 5.9]	0	0.006	0.026	0.029	0	0.014	0.088	0.179
[6.0, 6.9]	0	0	0.035	0.006	0	0.017	0.126	0.078
[7.0, 7.9]	0	0	0.038	0	0	0	0.161	0.017
[8.0, 8.9]	0	0	0.038	0	0	0	0.114	0.006
[9.0, 9.9]	0	0	0.064	0	0	0	0.035	0
[10.0, 29.9]	0	0	0.3	0	0	0	0.003	0

Table 8.19: End-to-end latency for the multicast exp. with 20 and 40 total load variables

Latency (ms)	Received (%)							
	80 total load var.				120 total load var.			
	1 SR	*2 SR*	*3 SR*	*4 SR*	*1 SR*	*2 SR*	*3 SR*	*4 SR*
0.2	0	0	0	0	0	0	0	0
0.3	0.017	0	0	0	0.098	0	0	0
0.4	1.442	0.008	0	0	2.288	0	0	0
0.5	11.195	0.106	0	0	13.389	0.017	0	0
0.6	25.862	0.742	0.006	0	26.147	0.228	0	0
0.7	30.822	3.156	0.075	0	29.545	1.264	0	0
0.8	20.916	8.634	0.328	0.006	19.486	5.534	0.003	0
0.9	7.158	16.318	1.142	0.009	6.895	12.705	0.074	0
1	1.707	24.35	3.354	0.101	1.521	23.062	0.407	0
1.1	0.459	22.79	6.268	0.358	0.301	24.581	1.393	0.004
1.2	0.211	14.011	10.246	0.874	0.162	17.083	2.528	0.054
1.3	0.078	5.699	12.376	1.97	0.067	8.093	3.729	0.192
1.4	0.026	1.84	11.9	3.345	0.032	3.328	4.055	0.36
1.5	0.009	0.717	9.523	4.933	0.026	1.346	3.815	0.617
1.6	0.012	0.434	6.726	5.828	0.012	0.518	2.995	0.855
1.7	0.003	0.23	4.638	6.067	0.006	0.445	2.481	0.951
1.8	0.012	0.185	3.285	5.589	0.003	0.283	2.08	1.104
1.9	0.012	0.118	2.462	4.784	0	0.218	1.69	0.939
2	0.006	0.064	2.034	4.157	0	0.114	1.699	0.882
2.1	0.014	0.045	1.766	3.509	0.006	0.152	1.554	0.817
2.2	0.02	0.022	1.431	2.874	0.006	0.1	1.325	0.725
2.3	0.006	0.045	1.16	2.372	0	0.069	1.105	0.782
2.4	0.006	0.036	0.892	2.223	0	0.052	1.194	0.705
2.5	0.006	0.022	0.847	2.086	0	0.052	1.123	0.675
2.6	0	0.02	0.829	1.77	0	0.066	1.049	0.686
2.7	0	0.017	0.672	1.564	0.003	0.035	1.203	0.659
2.8	0	0.02	0.597	1.277	0	0.014	1.096	0.602
2.9	0	0.014	0.582	1.241	0	0.024	0.977	0.817
3	0	0.017	0.69	1.235	0	0.003	0.989	0.598
3.1	0.003	0.006	0.551	1.158	0	0.045	1.055	0.61
3.2	0	0.008	0.485	1.015	0.006	0.014	0.989	0.594
3.3	0	0.008	0.473	0.979	0	0.014	0.998	0.644
3.4	0	0.011	0.509	0.979	0	0.007	0.969	0.732
3.5	0	0.006	0.536	0.904	0	0.01	0.98	0.594
3.6	0	0.008	0.473	0.922	0	0.003	0.992	0.648
3.7	0	0	0.437	0.833	0	0.007	1.013	0.663
3.8	0	0.006	0.392	0.919	0	0.017	1.01	0.598
3.9	0	0.011	0.452	0.883	0	0.007	0.853	0.636
4	0	0.006	0.41	0.847	0	0.014	0.927	0.801
4.1	0	0.008	0.458	0.794	0.003	0.003	1.055	0.663
4.2	0	0.006	0.404	0.871	0	0.007	1.073	0.709
4.3	0	0	0	0.806	0	0.007	1.075	0.771
4.4	0	0	0	0.85	0	0.007	1.013	0.736
4.5	0	0	0	0.773	0	0.017	1.254	0.909
4.6	0	0	0	0.856	0	0.014	1.093	0.748
4.7	0	0	0	0.842	0	0.017	1.304	0.748
4.8	0	0	0	0.85	0	0.017	1.334	0.782
4.9	0	0	0	0.842	0	0.017	1.364	0.782
5	0	0	0	1.071	0	0.003	1.628	0.97
5.1	0	0	0	0.886	0	0.014	1.836	0.985
5.2	0	0	0	1.056	0	0.01	2.062	1.131
[5.3, 5.9]	0	0.025	2.552	5.926	0	0.073	18.94	12.372
[6.0, 6.9]	0	0.076	2.1	6.383	0	0.09	10.413	17.701
[7.0, 7.9]	0	0.07	1.51	4.425	0	0.104	3.431	11.609
[8.0, 8.9]	0	0.006	0.443	2.357	0	0.041	1.527	10.577
[9.0, 9.9]	0	0.003	0.223	1.558	0	0.01	0.591	9.607
[10.0, 29.9]	0	0.003	0.172	1.241	0	0.024	0.654	8.657

Table 8.20: End-to-end latency for the multicast exp. with 80 and 120 total load variables



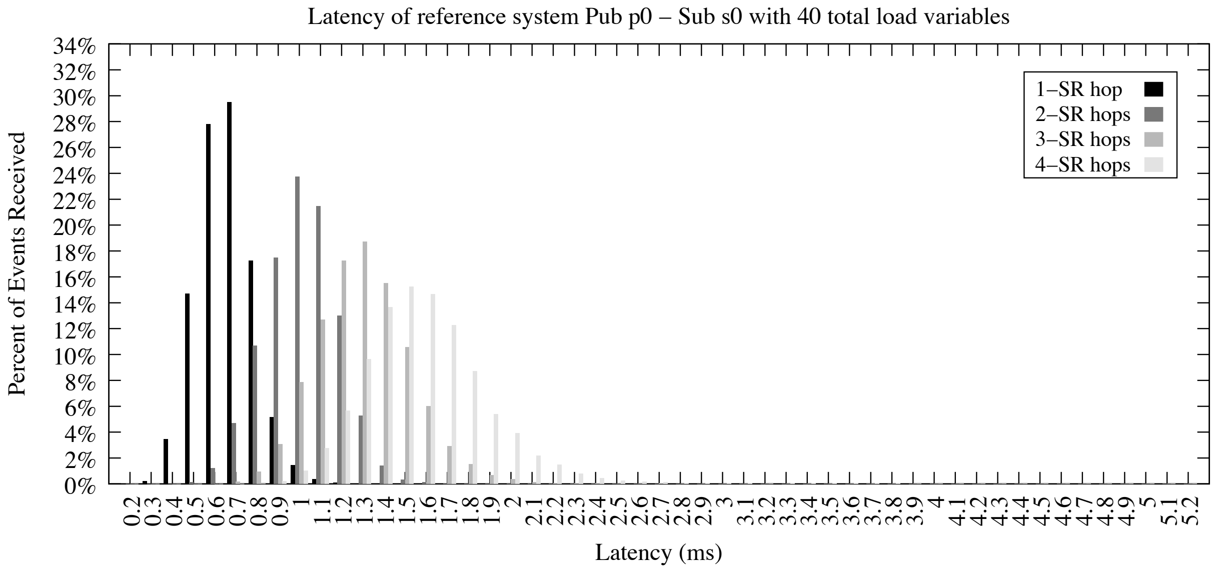
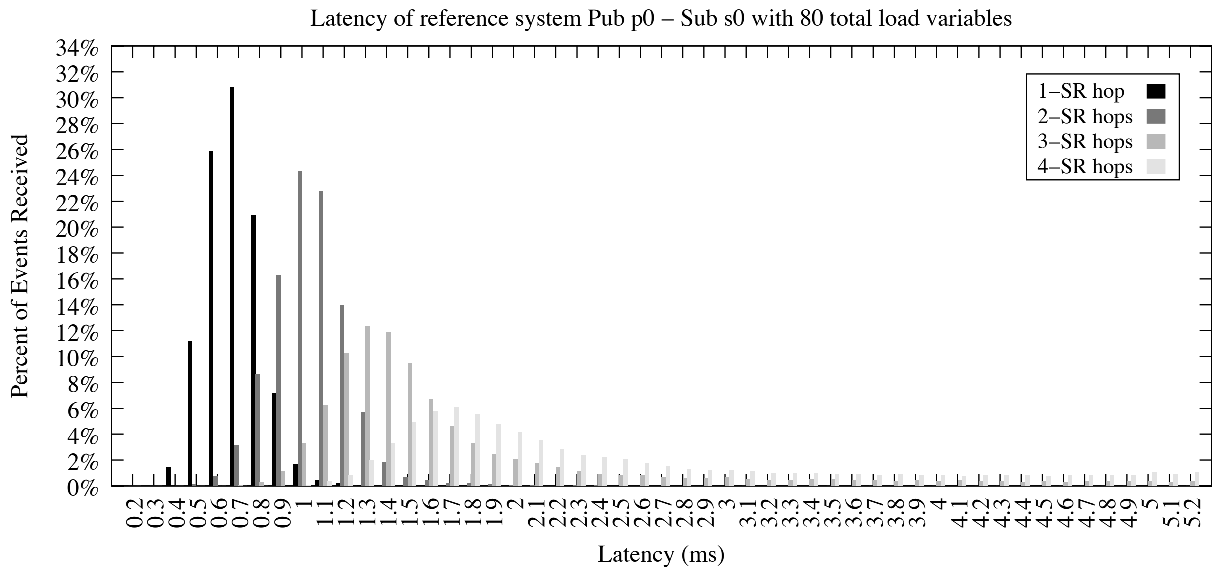


Figure 8.12: End-to-end latency for the multicast exp. with 20 and 40 total load variables

Total load variables	Mean (ms)							
	1 SR		2 SR		3 SR		4 SR	
	Sub s_0	Est. fun	Sub s_0	Est. fun	Sub s_0	Est. fun	Sub s_0	Est. fun
20	0.656	0.695	0.955	1.02	1.168	1.345	1.366	1.669
40	0.666	0.739	1.02	1.152	1.347	1.565	1.616	1.978
80	0.693	0.809	1.06	1.362	2.114	1.915	3.516	2.468
120	0.681	0.86	1.124	1.516	4.042	2.172	6.476	2.828

Table 8.21: Mean for the multicast exp.



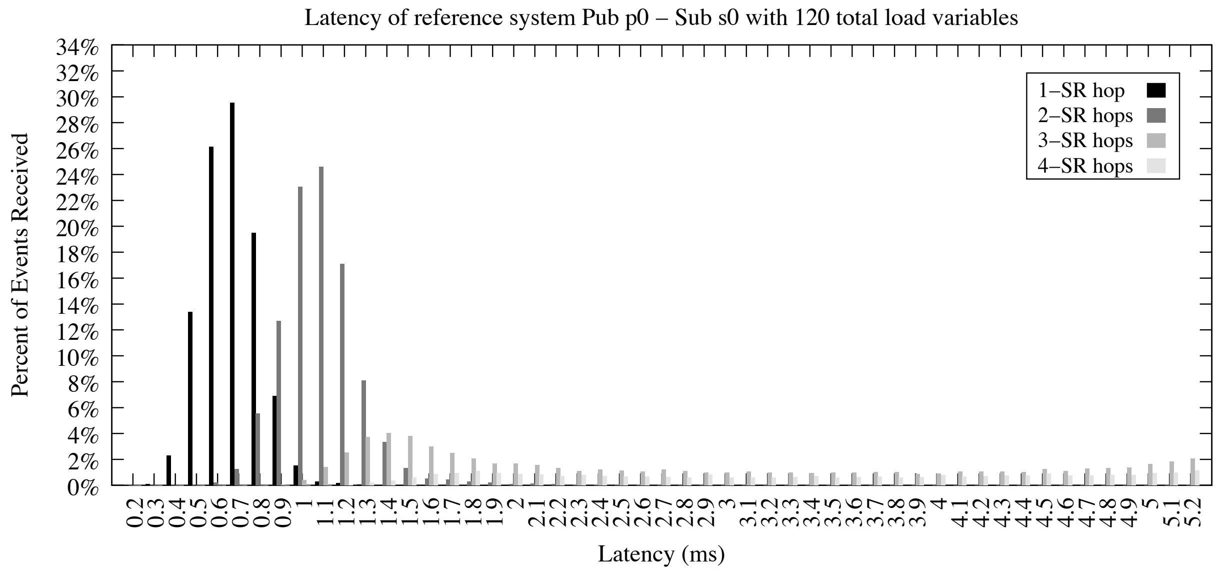


Figure 8.13: End-to-end latency for the multicast exp. with 80 and 120 total load variables

Total load variables	Standard deviation (ms)			
	1 SR	2 SR	3 SR	4 SR
20	0.125	0.187	0.767	0.262
40	0.133	0.224	0.53	0.431
80	0.136	0.366	1.55	2.319
120	0.136	0.48	2.127	2.758

Table 8.22: Standard deviation for the multicast exp.

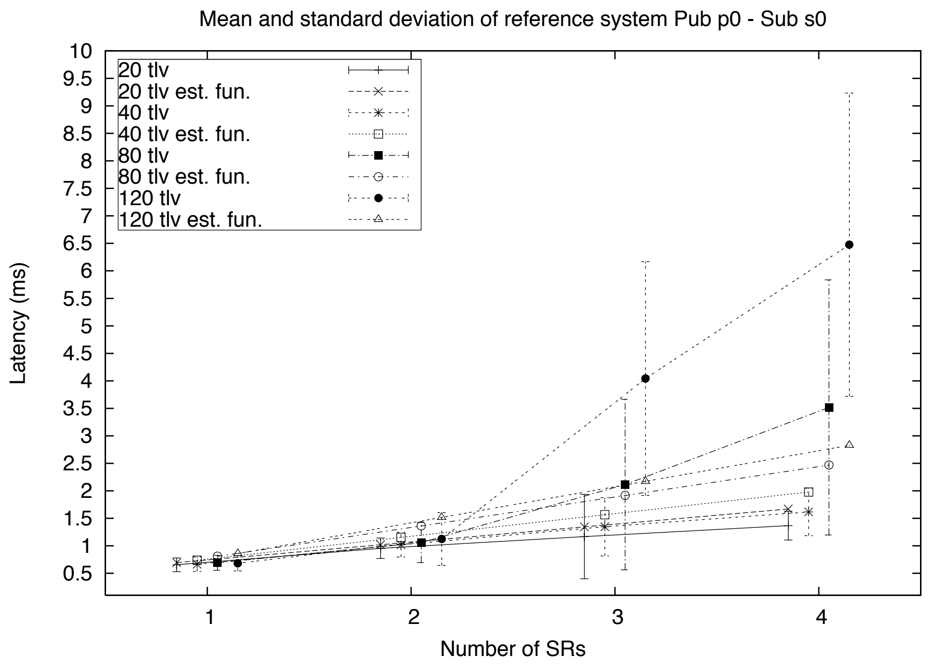


Figure 8.14: Mean and standard deviation for the multicast exp.

SR	Mean (ms)				Standard deviation (ms)			
	20 tlv	40 tlv	80 tlv	120 tlv	20 tlv	40 tlv	80 tlv	120 tlv
1-2	0.299	0.354	0.367	0.443	0.062	0.091	0.23	0.344
2-3	0.213	0.327	0.99	2.918	0.058	0.306	1.184	1.647
3-4	0.198	0.269	1.402	2.434	-0.505	-0.099	0.769	0.631
Avg	0.237	0.317	0.92	1.932	-0.128	0.099	0.728	0.874

Table 8.23: Difference of the mean and standard derivation between the hops for the multicast exp.

publisher. The exact calculation for the throughput is presented in Function 8.6. The reason for subtracting the 0.019 has to do with the fact that this is the throughput for the reference variable that is not multicasted.

$$TP_{sub} = (TP_{pub} - 0.019) * s + 0.019\, event/ms \tag{8.6}$$

This function doesn't work for 80 and 120 tlv because of the event drop (see Table 8.26) during these experiment runs. The reason for the high event drop is explained below in the event drop analysis.

Total load variables	Throughput (event/ms)							
	Pub $p_{(0-2)}$				Sub $s_{(0-n)}$			
	1 SR	2 SR	3 SR	4 SR	1 SR	2 SR	3 SR	4 SR
20	9.381	9.386	9.425	9.392	9.389	18.783	28.262	37.373
40	17.818	17.845	17.708	17.565	17.838	35.391	53.136	70.132
80	31.144	31.345	31.308	31.423	31.271	62.707	93.251	122.962
120	41.217	41.151	40.905	41.079	41.137	81.817	118.009	153.001

Table 8.24: Throughput for the multicast exp.

System Load Analysis

The system load is presented in Table 8.25. When there is only one SR in this experiment, the system load is consistent with the system load at SR i_0 when $s > 1$ for the hop scalability experiment (Section 8.4). In this case, the two topologies are almost the same; the only difference is that in this experiment the SR i_0 also has a communication link to edge SR e_3. The load of the reference system is very low, so it will not influence the system load on SR i_0 much. The load on the SR i_0 when $s > 1$ in this experiment can be compared to the SR $i_{(1-(n-1))}$ in the hop scalability experiment (Section 8.4) when $s > 1$. This is true because in both topologies these SRs have one incoming and one outgoing communication link. Again SR i_0 (when $s > 1$) in this experiment also has an additional outgoing communication link to the reference subscriber.

The system load on SR i_n when $s > 1$ in this experiment is a little higher than the system load on SR $i_{(1-(n-1))}$ (when $s > 1$) in the previous experiment. The reason for this is that SR i_n has three incoming communication links while SR $i_{(1-(n-1))}$ has only one. If these SRs $i_{(1-n)}$ in this experiment are compared to their counterparts in the hop scalability experiment, then the system load on these SRs is more than twice as much as in the hop scalability experiment. Even though the SRs in this experiment have to send out the event streams from the load variables on two outgoing communication links, they only receive one copy of each of the event streams. The extra work of forwarding the load events to two outgoing communication links should not account for such a large increase in the system load.

The reason for the large increase in the system load is attributed to the implementation of the multicast in the SR. The SR uses buffers of type ByteBuffer. The ByteBuffer type has methods for allocating the memory directly from the heap so that the OS can read and write to it directly without going through the JVM first. The SR allocates a pool of these buffers of size 24, so that each element of a status event can be stored in one of these buffers. These buffers are recycled as events are forwarded on the outgoing communication links in order to minimize the usage of the garbage collector. The problem occurs when the event is to be routed to multiple outgoing links. In this case, the ByteBuffer must be duplicated. The data itself is not duplicated but the wrapper (pointers) that holds the raw data (status event) has to be duplicated. For each of the outgoing communication links that the events are to be sent on, has to get its own wrapper of the ByteBuffer. After each event has been sent, the ByteBuffer is recycled back to the buffer pool, but it is not possible to have a pool of wrappers. As a result, they are collected by the garbage collector and this is what causes the high system load. If the SR were to be implemented in another language where direct access to memory is allowed, then this overhead could have been avoided.

Total load variables	System load (%)			
	SR i_0 (s = 1)	SR i_0 (s > 1)	SR $i_{(1-(n-1))}$ (s > 1)	SR i_n (s > 1)
20	6-15	15-27	16-24	6-9
40	15-25	32-41	33-42	12-16
80	30-37	55-67	55-63	22-26
120	40-46	75-80	66-71	26-29

Table 8.25: System load on the SRs for the multicast exp.

Event Drop Analysis

The event drop for the multicast experiment is presented in Table 8.26. The event drop is calculated as follows:

$$Event\ drop = \frac{\sum_{j=0}^{n}(Event\ drop(SR_{ij}))}{\sum_{j=0}^{2}(Events\ sent\ from\ SR_{ej})} * 100 \tag{8.7}$$

The reason that the subpath from the SRs connection to an edge SR and then to a subscriber is not included in Function 8.7 is that very few (close to 0) events were dropped on those path segments. Consequently, the event drop occurs at the SRs that perform multicast.

In the case that s is 2 some events are dropped for the 80 and 120 tlv. The system load on SR i_1 is between 75-80% and its socket buffers are overrun. It is important to keep in mind that for this configuration only SR i_1 multicasts the events to two outgoing communication links while SR i_0 only forwards the load variables to one outgoing communication link. When one more SR is added, then the event loss increments drastically. At first it might look that this drastic increment indicates that there is something wrong in this configuration. Why should it make such a big difference if two SRs are performing multicast compared to one? The reason is that when s is 3 there is one distinction between SR i_2 and SR i_1; SR i_2 has a dedicated incoming communication link for each of the streams published by each of the load publishers. On the contrary, SR i_1 has only one incoming communication link that has to be shared for all the event streams received by SR i_2. The reason for the drastic increase in the event drop is due to the fact that SR i_2 has twice the buffer space than SR i_1. As one more SR is added, s becomes 4, the event drop increments consistently with that of the increment from 2 to 3 SRs. This is again due to the fact that now there are two SRs with only one incoming communication link that is performing multicast.

Total load variables	Event drop (%)											
	Pub $p_{(0-2)}$ - SR $e_{(0-2)}$				SR $e_{(0-2)}$ - SR $e_{(3-n)}$				SR $e_{(3-n)}$ - Sub $s_{(0-n)}$			
	1 SR	2 SR	3 SR	4 SR	1 SR	2 SR	3 SR	4 SR	1 SR	2 SR	3 SR	4 SR
20	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0.022	0.005	0	0	0	0
80	0	0	0	0	0	0.022	1.108	3.489	0	0	0	0
120	0	0	0	0	0	0.374	5.349	10.98	0	0	0	0

Table 8.26: Event drop for the multicast exp.

8.6 Multicast Rate-filtering Routing Mechanism Experiment

The multicast rate-filtering routing experiment uses the same topology as the multicast experiment (see Figure 8.11), with the configuration of four SRs. Events are filtered as close to their source as possible in order to save resources further down the paths. This experiment consists of the following systems and variables:

- One reference system which consists of Pub p_0 – Sub s_0.
- Two load publishers which are Pub p_1 and Pub p_2.
- Four load subscribers, where each subscribes to Pub p_1's and Pub p_2's status variables at an interval of 1ms and 2 ms (filtering occurs.)
- Total number of load variables (tlv) is 40 and 80 (each load system provides half of the tlv.)
- Four status routers on the path from the reference publisher to the reference subscriber.

The multicast rate-filtering routing experiment is performed in the following way: for the first run, SR i_0 performs filtering, for the second run both SR i_0 and SR i_1 perform filtering and so on. All load events have to be forwarded to SR i_3, until all the load subscribers are subscribing to an interval of 2ms. When this occurs, the filtering of the load events is performed at the edge SR e_1 and SR e_2.

8.6.1 Analysis of Results

The experiment is analyzed the same way as the previous experiments. First the end-to-end latency for the reference system is presented, then the mean and standard deviation. The throughput, system load, and event drop discussions follow.

End-to-End Latency Analysis

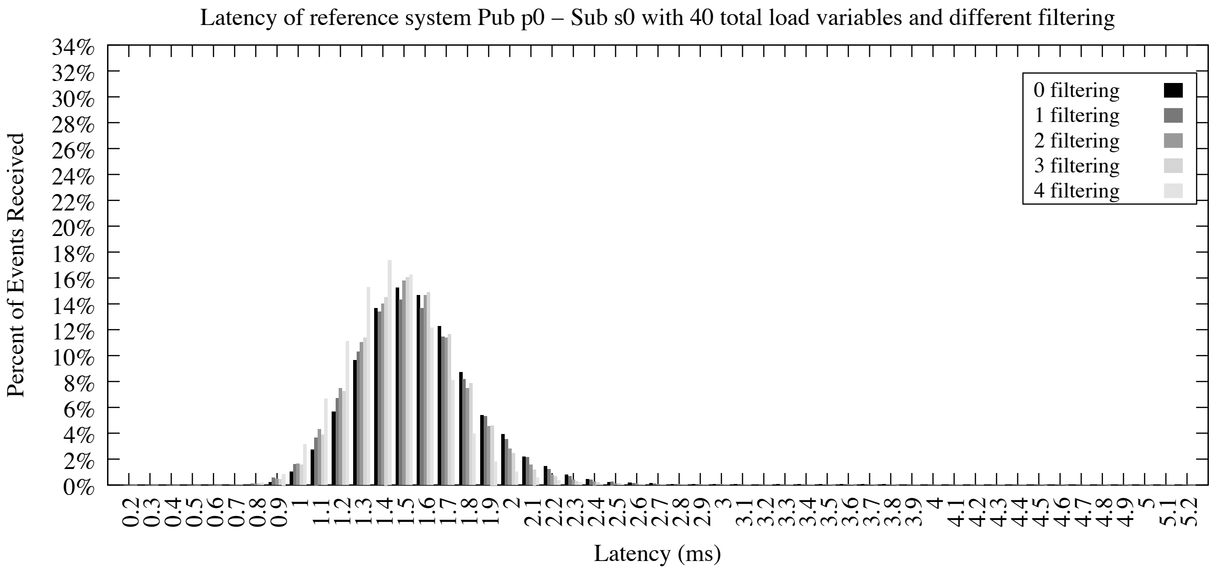
The results from the reference system during the recording phase of the experiment are presented in Table 8.27 and depicted in Figure 8.15. As it can be seen from the figure, when there are 40 tlv the difference in the end-to-end latency among the different filtering runs is small. The curve is slightly better when all the load subscribers subscribe to an interval of 2 ms. The results are more interesting for 80 tlv, where there is a very little difference in the end-to-end latency when none of the load subscribers is filtering and when only SR i_0 is filtering. The reason is explained in the system load (see Section 8.6.1). An improvement can be seen from 1 filtering to 2 filterings and again from 2 filterings to 3 filterings. A greater improvement can be observed from 3 filterings to 4 filterings. This occurs because when all the load subscribers are filtering then the filtering is performed at the edge SR of the load publisher. Therefore only half of the events are forwarded to the SRs. This can also be seen as the throughput is reduced to almost half.

Mean and Standard Deviation Analysis

In Table 8.28 and Figure 8.16, the mean and standard deviation for the different filtering runs with the two load patterns are presented. For 40 tlv there is a small difference in the mean for the different runs, but a slight improvement is observed as more of the load subscribers are subscribing to a higher interval. As far as the standard deviation is concerned, in the case of 40 tlv there is an exception when only Sub s_4 is the only one that is subscribing to an interval of 2ms. This can also be seen from Table 8.27. The tail for the end-to-end latency curve is larger than the other runs. The reason for this might be that some of the JVMs are optimizing themselves during the recording phase. This results in a very small percentage of the events to be excessively delayed, which again

Latency (ms)	Received (%)									
	40 total load var.					80 total load var.				
	0 filter	*1 filter*	*2 filter*	*3 filter*	*4 filter*	*0 filter*	*1 filter*	*2 filter*	*3 filter*	*4 filter*
0.2	0	0	0	0	0	0	0	0	0	0
0.3	0	0	0	0	0	0	0	0	0	0
0.4	0	0	0	0	0	0	0	0	0	0
0.5	0	0	0	0	0	0	0	0	0	0
0.6	0	0	0	0	0	0	0	0	0	0
0.7	0.003	0.018	0.014	0.012	0.023	0	0.003	0	0.006	0.031
0.8	0.036	0.114	0.086	0.131	0.181	0.006	0.003	0.029	0.023	0.183
0.9	0.215	0.562	0.469	0.433	0.853	0.009	0.053	0.076	0.125	0.827
1	1.043	1.594	1.654	1.565	3.136	0.101	0.166	0.255	0.286	2.419
1.1	2.742	3.671	4.307	3.877	6.67	0.358	0.494	0.76	0.927	4.95
1.2	5.665	6.684	7.467	7.251	11.132	0.874	1.333	1.576	2.004	8.409
1.3	9.629	10.305	11.039	11.381	15.292	1.97	2.601	2.932	3.815	11.178
1.4	13.646	13.379	14.009	14.53	17.369	3.345	3.822	5.087	6.157	13.325
1.5	15.244	14.335	15.778	16.043	16.253	4.933	5.249	7.021	8.989	13.448
1.6	14.661	13.651	14.653	14.909	12.151	5.828	6.036	7.954	10.812	11.049
1.7	12.282	11.472	11.376	11.663	8.101	6.067	6.068	8.277	11.85	8.372
1.8	8.706	8.172	7.487	7.856	3.96	5.589	5.619	7.502	10.479	5.157
1.9	5.406	5.309	4.519	4.604	1.813	4.784	4.744	6.372	7.82	3.467
2	3.902	3.522	2.799	2.452	1.031	4.157	4.289	5.321	6.417	2.491
2.1	2.204	2.156	1.577	1.195	0.566	3.509	3.624	4.165	4.194	1.91
2.2	1.473	1.231	0.801	0.678	0.36	2.874	3.095	3.393	3.29	1.412
2.3	0.781	0.69	0.412	0.302	0.218	2.372	2.557	2.738	2.342	1.257
2.4	0.46	0.401	0.278	0.218	0.11	2.223	2.208	2.254	1.852	0.945
2.5	0.234	0.269	0.106	0.125	0.062	2.086	1.942	1.94	1.47	0.91
2.6	0.176	0.132	0.103	0.052	0.051	1.77	1.708	1.652	1	0.618
2.7	0.137	0.108	0.06	0.067	0.02	1.564	1.54	1.535	0.755	0.57
2.8	0.07	0.067	0.054	0.041	0.031	1.277	1.368	1.306	0.717	0.455
2.9	0.078	0.067	0.046	0.029	0.023	1.241	1.141	1.18	0.642	0.404
3	0.059	0.079	0.054	0.038	0.031	1.235	1.25	1.063	0.592	0.315
3.1	0.064	0.059	0.034	0.02	0.025	1.158	1.034	0.974	0.548	0.275
3.2	0.042	0.023	0.029	0.023	0.025	1.015	1.034	0.839	0.443	0.243
3.3	0.05	0.023	0.04	0.015	0.011	0.979	1.126	0.842	0.446	0.235
3.4	0.045	0.038	0.04	0.035	0.017	0.979	1.049	0.807	0.513	0.261
3.5	0.061	0.041	0.037	0.023	0.006	0.904	0.863	0.748	0.42	0.183
3.6	0.059	0.053	0.046	0.015	0.003	0.922	0.848	0.687	0.362	0.178
3.7	0.059	0.044	0.031	0.023	0.006	0.833	0.857	0.687	0.344	0.152
3.8	0.047	0.038	0.02	0.009	0.003	0.919	0.836	0.613	0.391	0.169
3.9	0.031	0.029	0.02	0.026	0.034	0.883	0.91	0.616	0.347	0.126
4	0.028	0.029	0.017	0.032	0.014	0.847	0.854	0.625	0.347	0.157
4.1	0.033	0.041	0.02	0.026	0.014	0.794	0.869	0.622	0.33	0.129
4.2	0.042	0.032	0.031	0.015	0.014	0.871	0.842	0.652	0.265	0.106
4.3	0.025	0.029	0.026	0.023	0.008	0.806	0.828	0.564	0.28	0.086
4.4	0.042	0.029	0.023	0.012	0.017	0.85	0.745	0.561	0.35	0.115
4.5	0.039	0.032	0.014	0.009	0.006	0.773	0.822	0.522	0.303	0.089
4.6	0.025	0.026	0.023	0.015	0.003	0.856	0.786	0.613	0.268	0.126
4.7	0.017	0.02	0.026	0.009	0.011	0.842	0.807	0.569	0.283	0.115
4.8	0.022	0.026	0.02	0.015	0.006	0.85	0.878	0.578	0.265	0.117
4.9	0.039	0.032	0.011	0.02	0.006	0.842	0.848	0.625	0.227	0.083
5	0.036	0.018	0.017	0.026	0.003	1.071	0.907	0.74	0.195	0.123
5.1	0.028	0.029	0.014	0.017	0.008	0.886	0.86	0.681	0.254	0.092
5.2	0.033	0.029	0.014	0	0.006	1.056	0.904	0.631	0.219	0.115
[5.3, 5.9]	0.179	0.269	0.169	0.09	0.048	5.926	5.778	4.05	1.668	0.762
[6.0, 6.9]	0.078	0.345	0.072	0.044	0.054	6.383	5.843	3.657	2.097	0.687
[7.0, 7.9]	0.017	0.161	0.04	0.006	0.042	4.425	3.866	1.679	0.933	0.45
[8.0, 8.9]	0.006	0.091	0.014	0	0.014	2.357	1.88	0.643	0.577	0.318
[9.0, 9.9]	0	0.053	0	0	0.014	1.558	1.026	0.414	0.359	0.195
[10.0, 29.9]	0	0.371	0	0.003	0.144	1.241	1.185	0.367	0.397	0.212

Table 8.27: End-to-end latency for the multicast rate-filtering routing exp. with 40 and 80 tlv



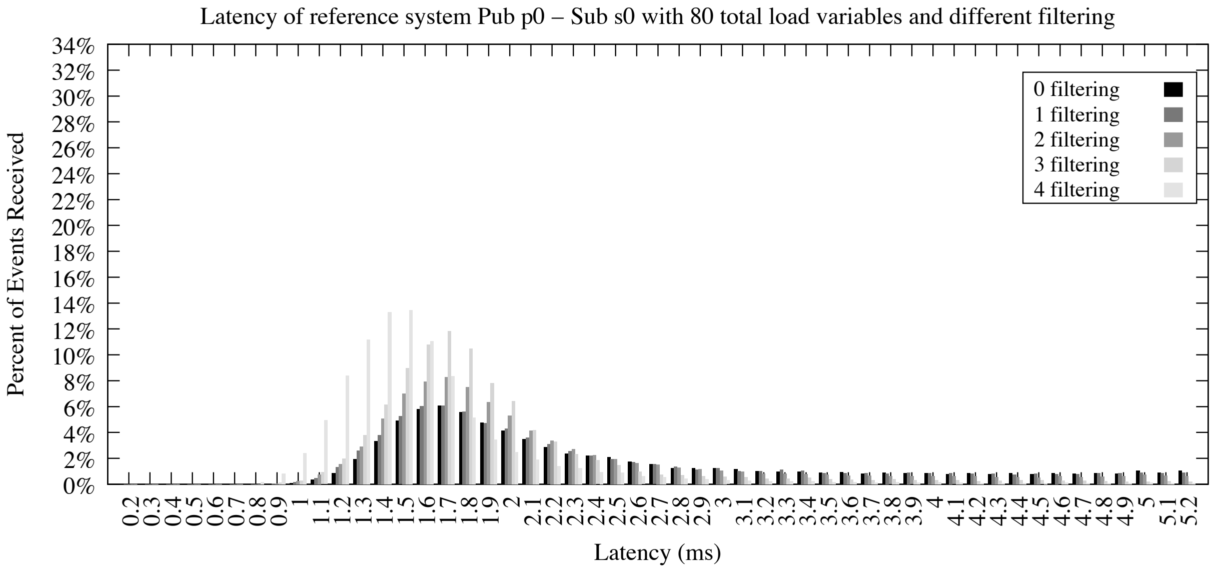


Figure 8.15: End-to-end latency for the multicast rate-filtering routing exp. with 40 and 80 tlv

affects the standard deviation.

For 80 tlv the mean improves quite a lot as the number of load subscribers that subscribe to the interval of 2 ms increases. This happens because the work that each of the SRs needs to do is reduced, as only half of the events need to be forwarded on one of the communication links while all the events are forwarded on the other communication link. This can also be seen as the system load is reduced (see Table 8.31) as the load subscribers start to subscribe at an interval of 2 ms. In the case of 80 tlv there is no exception in the standard deviation; it improves as the number subscribers that subscribe to an interval of 2 ms increases.

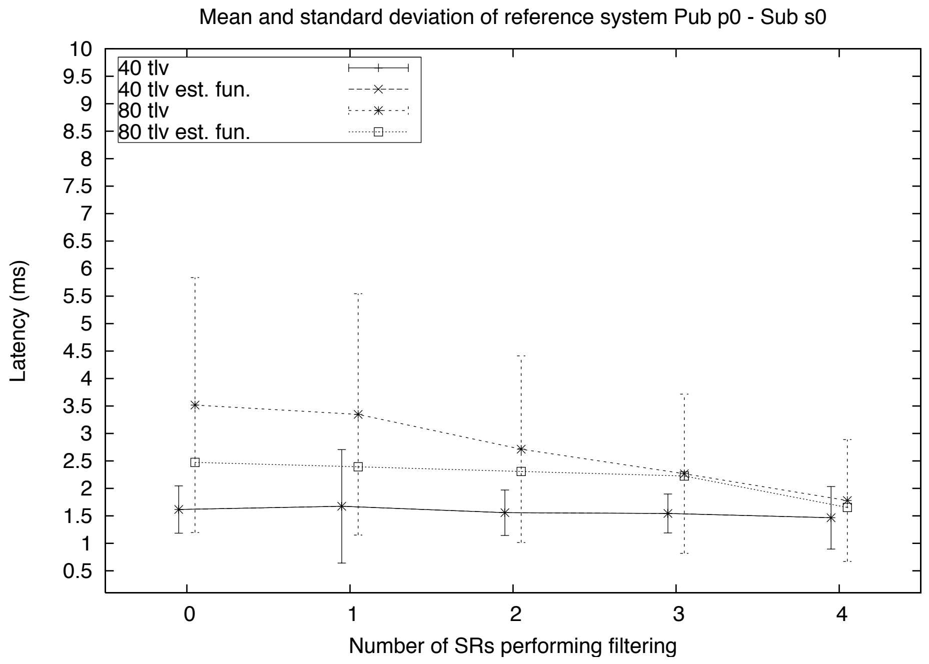


Figure 8.16: Mean and standard deviation for the multicast rate-filtering routing exp.

Total load variables	Mean (ms)									
	0 filter		1 filter		2 filter		3 filter		4 filter	
	Sub s_0	Est. fun	Sub s_0	Est. fun	Sub s_0	Est. fun	Sub s_0	Est. fun	Sub s_0	Est. fun
40	1.616	1.968	1.674	1.925	1.558	1.883	1.544	1.834	1.466	1.511
80	3.516	2.473	3.347	2.392	2.713	2.308	2.267	2.226	1.78	1.652

Table 8.28: Mean for the multicast rate-filtering routing exp.

Total load variables	Standard deviation (ms)				
	0 filter	1 filter	2 filter	3 filter	4 filter
40	0.431	1.033	0.414	0.355	0.569
80	2.319	2.197	1.698	1.448	1.109

Table 8.29: Standard deviation for the multicast rate-filtering routing exp.

Throughput Analysis

The throughput for this experiment is shown in Table 8.30. Function 8.8 presents a prediction of the expected throughput. The reason that the throughput from the results from the experiment runs does not match with the expected throughput for 80 tlv has do to with the event drop (see Section 8.5.1.)

Total load variables	Filter	Throughput (event/ms)		
		Pub $p_{(0-2)}$	Sub $s_{(0-4)}$	Est. Sub $s_{(0-4)}$
40	0	17.565	70.132	70.203
	1	17.641	61.847	61.696
	2	17.778	53.69	53.296
	3	17.684	44.565	44.172
	4	17.653	34.876	35.287
80	0	31.423	122.962	125.635
	1	31.456	107.822	110.049
	2	31.368	92.605	94.066
	3	31.36	78.067	78.372
	4	31.21	61.9	62.401

Table 8.30: Throughput for the multicast rate-filtering routing exp.

$$TP_{sub} = (TP_{pub} - 0.019) * (4 - filt) + \frac{TP_{pub} - 0.019}{2} * filt + 0.019\, event/ms \qquad (8.8)$$

System Load Analysis

The system load is presented in Table 8.31 for the four status routers. The entries in bold indicate that only half of the events are forwarded on one of the outgoing communication links, and the entries that are bold and emphasized receive load variables at an interval of 2 ms and forward all the received events on both outgoing communication links. As it can be seen from the table, when a SR has to forward only half of the events to both outgoing communication links then the system load is 7-10 % less for 40 tlv and 10-15 % less for 80 tlv compared to when all events are forwarded to both outgoing communication links. When all the load subscribers subscribe to an interval of 2 ms, then only half of the events are forwarded from the edge SR of the load publishers. For 40 tlv the system resources are reduced to about half compared to when all events are forwarded.

Total load variables	Filter	System load (%)			
		SR i_3	SR i_2	SR i_1	SR i_0
40	0	32-42	32-42	35-43	13-17
	1	**28-35**	30-40	32-45	13-16
	2	**23-33**	**23-34**	30-40	11-14
	3	**24-33**	**19-25**	**22-30**	11-15
	4	***14-30***	***12-22***	***12-23***	7-12
80	0	55-65	55-65	56-65	22-25
	1	**45-55**	52-65	57-65	20-25
	2	**45-55**	**38-50**	53-65	20-25
	3	**43-55**	**38-50**	**43-50**	21-25
	4	***24-42***	***24-38***	***26-37***	8-15

Table 8.31: System load on the SRs for the multicast rate-filtering routing exp.

Event Drop Analysis

The event drop for the multicast rate-filtering routing experiment is presented in Table 8.32. The event drop is calculated according to Function 8.7.

According to the results, as the system load decreases (due to forwarding only half of the events to both outgoing communication links) the event drop also decreases. This is because duplication of the holder to the event buffers (see explanation in Section 8.5.1) is reduced to half as only half the events are forwarded on both outgoing communication links. As the duplication is reduced so is the execution of the garbage collector. There is one anomaly to this. For 40 tlv when one SR is filtering then the event drop is higher than when no filtering is performed. The cause of this might have been an interruption by the general purpose operating system that was performing some tasks or that one of the JVMs was optimizing itself.

8.7 Condensation Function Mechanism Experiment

This experiment intends to justify the resource conservation that occurs when using condensation functions compared to having the logic of these functions at the application layer. For each of the

Total load variables	Filter	Event drop (%)		
		Pub $p_{(0-2)}$ - SR $e_{(0-2)}$	SR $e_{(0-2)}$ - SR $e_{(3-7)}$	SR $e_{(3-7)}$ - Sub $s_{(0-4)}$
40	0	0	0.005	0
	1	0	0.114	0
	2	0	0.001	0
	3	0	0.001	0
	4	0	0.044	0
80	0	0	3.489	0
	1	0	3.259	0
	2	0	1.982	0
	3	0	1.104	0
	4	0	0.679	0

Table 8.32: Event drop for the multicast rate-filtering routing exp.

experiment runs, the end-to-end latency of other event streams in the system is measured. Since the claim is that by moving the application layer into the middleware layer as a condensation function saves resources in the overall system, then the latency of the other streams should be reduced. The topology for this experiment is depicted in Figure 8.17 and the setting of the variables is as follows:

- Three reference system which consists of Pub p_0 – Sub s_0, Pub p_1 – Sub s_1, and Pub p_1 – Sub s_2.
- Two load publishers which are Pub p_2 and Pub p_3.
- Two load subscribers which are Sub s_3 and Sub s_4 that subscribe to all variables published by both Pub p_2 and Pub p_3.
- Total number of load variables (tlv) is 40, 80, 120 (half provided by each of the load publishers).

The different reference systems are present in order to study the effect of the condensation function at different places in the network. Reference system Pub p_0 – Sub s_0 checks the effect on SR i_1 with and without the condensation function. It is important to keep in mind what was observed in the load scalability experiment (see Section 8.4). When the load is very low the end-to-end latency is worse than in the case of a medium load. As the condensation function is placed on SR i_0 the load on SR i_1 will be very low as it can be observed by looking at the throughput.

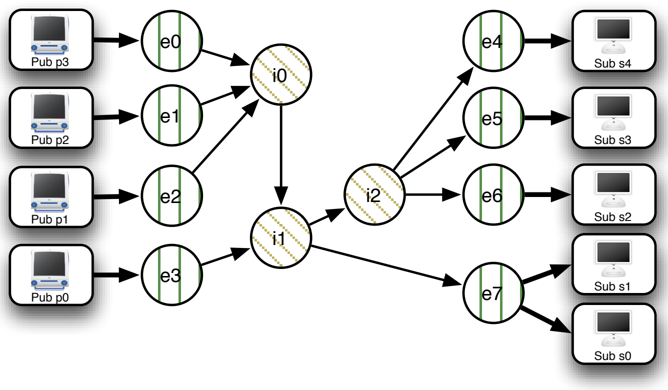


Figure 8.17: Topology for the condensation function exp.

So, for this reference system it might be observed that the end-to-end latency will be worse with the condensation function compared to without it. Of course, the system load will be drastically reduced, which will allow for many more event streams to be allocated and thus enhance the scalability.

The two other reference systems both use the reference publisher Pub p_1. Their first purpose is to test the effect of placing a condensation function on a SR, namely SR i_0 in this case. What is the cost of performing the condensation of multiple events streams into a new event stream, compared to routing all the event streams? It might be that the placement of the condensation function will save resources in the overall network, but what about the SR that the condensation function is loaded on? The reference system Pub p_1 – Sub s_1 will be used to test the effect of a condensation function placed on a single SR.

The second reference system involving Pub p_1, namely Pub p_1 – Sub s_2, tests the resource savages in the overall network with the use of a condensation function. Since both the load subscribers (Sub s_3 and Sub s_4) subscribe to all the load variables published by Pub p_2 and Pub p_3, multicast will take place. The place where the multicast will be performed, i.e. where the multicast tree splits, is at SR i_2. As observed and explained in the multicast experiment (see Section 8.5.1) the place where the multicast takes place is expensive. The reference system Pub p_1 – Sub s_2 will test how the condensation function affects the network resource usage also further down the multicast tree.

8.7.1 Condensation Function Average

The first condensation function takes as its input all the load variables that are published by Pub p_2 and Pub p_3 and publishes a new variable that is the average of the input variables. The triggering mechanism that is used is the interval, where every 1 ms the current values of the variables are used to calculate the current average. The condensation function is placed on SR i_0 and the load subscribers Sub s_3 and Sub s_4 subscribe to the variable produced by the condensation function instead of the load variables.

As is was done for the other experiments, the end-to-end latency, mean and standard deviation, throughput, system load and event drop are presented and analyzed.

Analyzing of End-to-End Latency

The results from the reference systems during the recording phase of the experiments are presented in Tables 8.33, 8.34, 8.35, and 8.36 and depicted in Figures 8.18, 8.19, and 8.20. According to the results there is not much difference for the end-to-end latency for reference system Pub p_0 – Sub s_0. As it was observed in the load scalability experiment, the end-to-end latency did not get affected much by the load on the system and this can explain why there is not any improvement in the end-to-end latency dimension by placing a condensation function into the middleware layer.

In the reference system Pub p_1 – Sub s_1 an improvement in the end-to-end latency is observed, and this is a result of introducing a condensation function. As this event stream also traverses the SR where the condensation function is loaded, it looks as if it has no negative effect on the end-to-end latency of other event streams. The resources saved by not forwarding all the event streams that become condensed into one is greater than the resources needed to perform the work necessary by the condensation function.

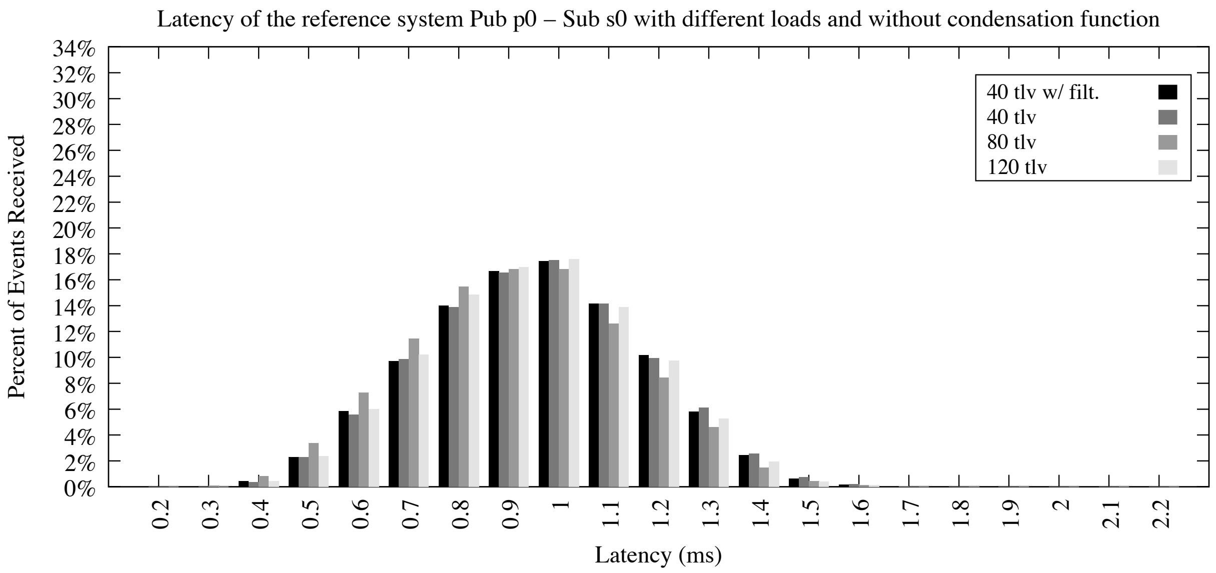
The real advantage of the condensation function can be observed from reference system Pub

p_1 – Sub s_2, as this event stream traverses a SR which is a splitting point of the load variables multicast tree. Since the cost of the splitting of the multicast streams is expensive (see Section 8.5.1) the benefit of the condensation function is shown further down the path. When there is no condensation function at the higher load patterns, then this reference system stream almost collapses with respect to end-to-end latency. When the condensation function is used then system wide improvements can be observed.

The only anomaly that can be seen with respect to end-to-end latency is that for the run with 120 tlv and condensation function, the end-to-end latency is shifted 0.1 ms to the right for reference systems containing reference publisher Pub p_1. This might be due to the extra load on SR i_0 (more load variables that are condensed by the condensation function.)

Latency (ms)	Received (%)					
	Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
0.2	0	0	0	0	0	0
0.3	0.017	0	0	0	0	0
0.4	0.428	0	0.009	0.174	0	0
0.5	2.311	0	0.124	1.959	0.012	0.078
0.6	5.849	0.029	0.613	6.767	0.336	0.99
0.7	9.722	0.376	2.07	10.609	2.407	4.245
0.8	14.009	1.781	4.853	12.426	7.16	9.21
0.9	16.662	4.919	8.238	14.144	12.825	12.662
1	17.443	10.011	13.057	16.635	18.131	15.674
1.1	14.163	13.854	15.335	13.331	19.309	16.603
1.2	10.202	17.008	16.054	9.385	17.338	15.055
1.3	5.803	17.075	14.811	7.441	12.877	10.946
1.4	2.47	14.224	11.065	5.156	6.671	7.286
1.5	0.628	9.727	7.261	1.762	2.196	4.386
1.6	0.182	5.24	3.998	0.197	0.44	2.028
1.7	0.029	2.652	1.59	0	0.162	0.613
1.8	0.012	1.385	0.624	0	0.043	0.148
1.9	0	0.674	0.171	0	0.023	0.035
2	0	0.387	0.043	0	0.006	0.006
2.1	0.006	0.13	0.012	0.003	0.003	0.012
2.2	0	0.069	0.009	0	0	0.006
[2.3, 2.9]	0.052	0.038	0.072	0.009	0.009	0.014
[3.0, 3.9]	0.012	0.023	0.055	0.003	0.009	0.012
[4.0, 4.9]	0	0.003	0.04	0	0.003	0.003
[5.0, 5.9]	0	0	0.026	0	0	0.003
[6.0, 6.9]	0	0	0.026	0	0	0.003
[7.0, 7.9]	0	0	0.014	0	0	0.003
[8.0, 8.9]	0	0	0	0	0	0.003
[9.0, 9.9]	0	0	0.017	0	0	0.006
[10.0, 29.9]	0	0	0.208	0	0	0.014

Table 8.33: End-to-end latency for the cond. function average exp. with 40 tlv and filtering



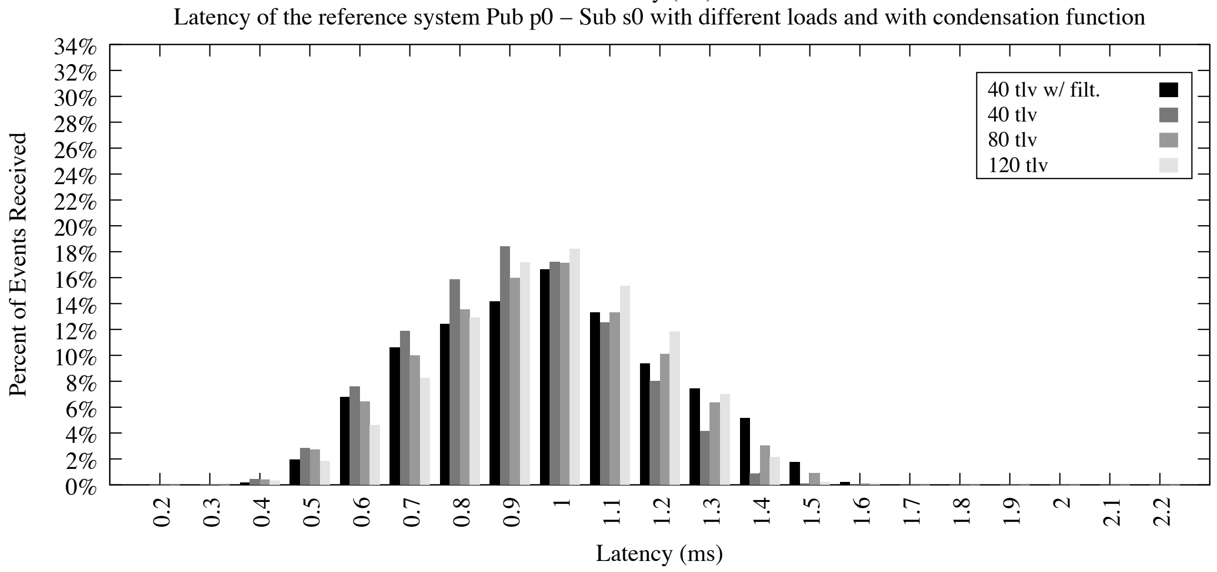
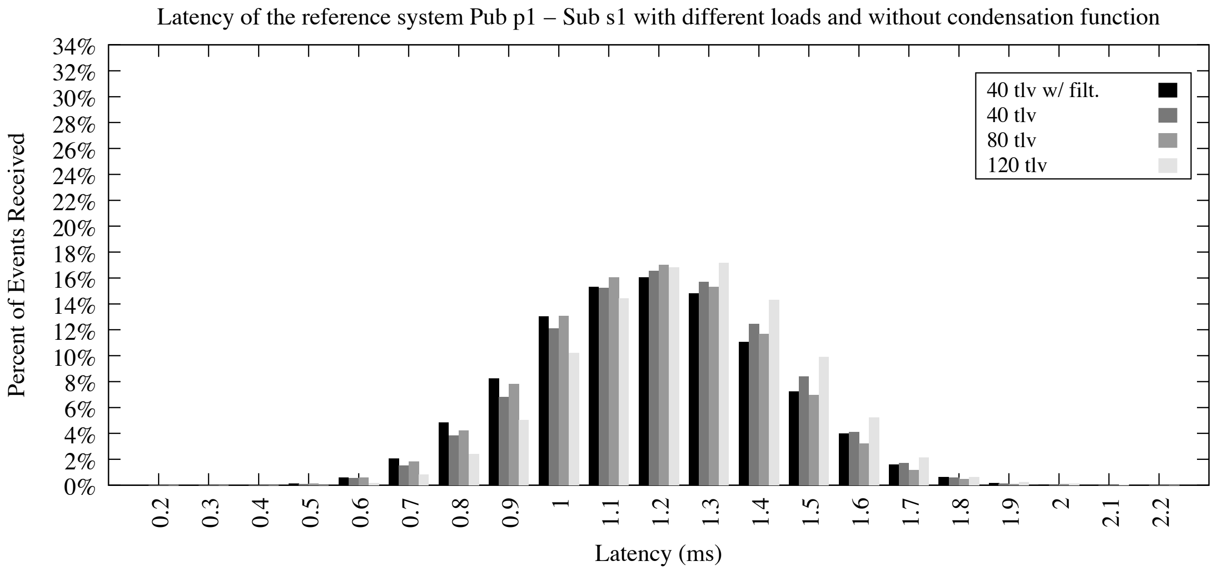


Figure 8.18: End-to-end latency for the cond. function average exp. for Sub s_0



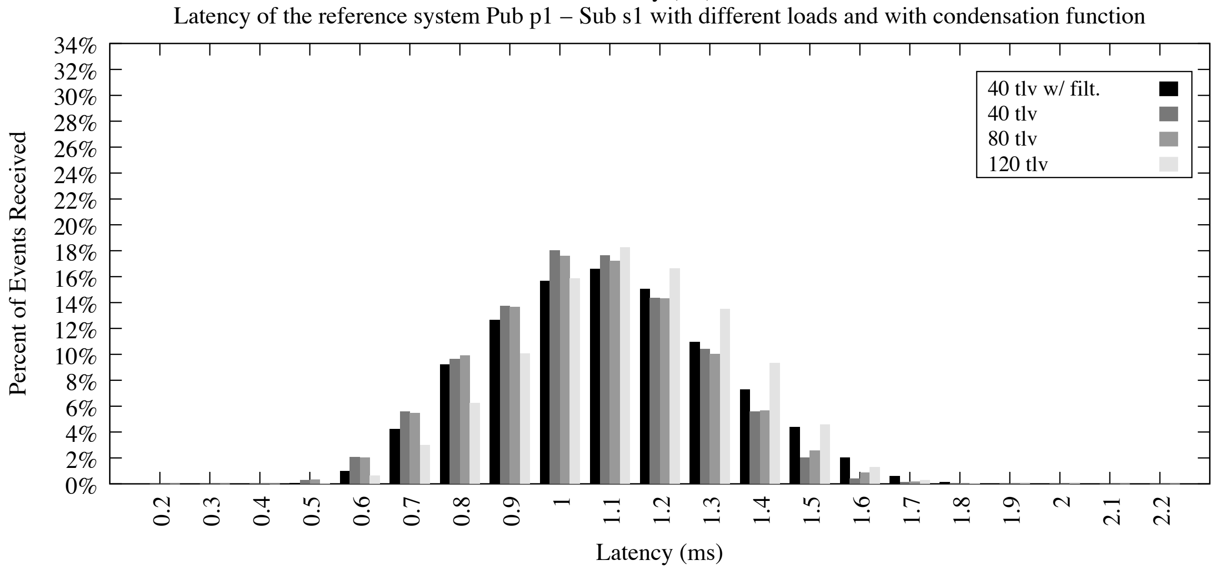
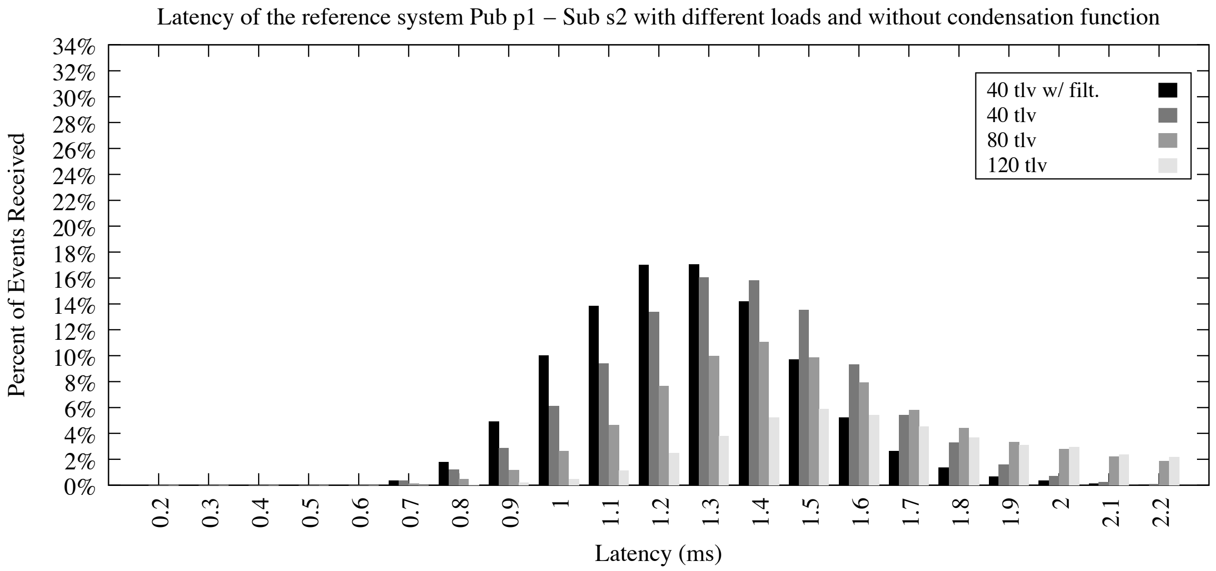


Figure 8.19: End-to-end latency for the cond. function average exp. for Sub s_1



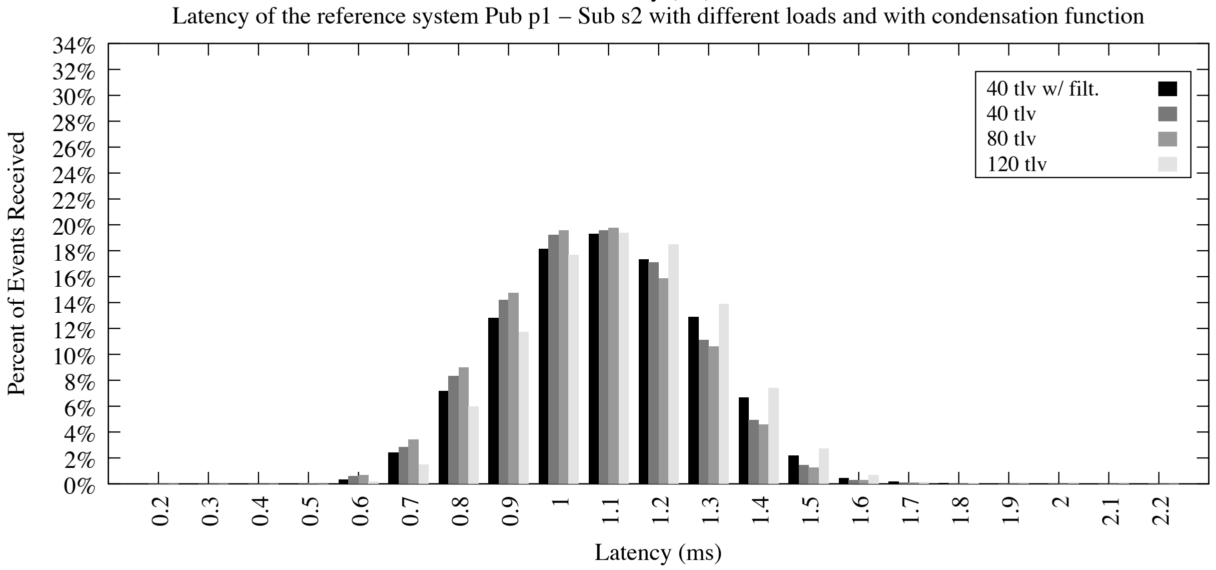


Figure 8.20: End-to-end latency for the cond. function average exp. for Sub s_2

Latency (ms)	Received (%)					
	Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
0.2	0	0	0	0	0	0
0.3	0.014	0	0	0.02	0	0
0.4	0.378	0.003	0	0.448	0.017	0
0.5	2.298	0.092	0	2.858	0.289	0.067
0.6	5.603	0.563	0.04	7.6	2.071	0.593
0.7	9.859	1.512	0.363	11.9	5.583	2.827
0.8	13.879	3.835	1.197	15.863	9.638	8.341
0.9	16.572	6.807	2.867	18.421	13.737	14.211
1	17.501	12.107	6.118	17.214	18.013	19.233
1.1	14.142	15.226	9.418	12.539	17.625	19.578
1.2	9.946	16.554	13.384	7.997	14.342	17.116
1.3	6.136	15.705	16.041	4.166	10.411	11.112
1.4	2.581	12.468	15.807	0.859	5.589	4.941
1.5	0.744	8.423	13.554	0.084	2.039	1.438
1.6	0.208	4.126	9.349	0.006	0.402	0.312
1.7	0.055	1.728	5.423	0.003	0.121	0.09
1.8	0.026	0.583	3.297	0	0.035	0.032
1.9	0.014	0.141	1.624	0	0.02	0.006
2	0.003	0.038	0.73	0	0.012	0.017
2.1	0.003	0.014	0.239	0	0.006	0.006
2.2	0	0.014	0.104	0	0.003	0.009
[2.3, 2.9]	0.029	0.046	0.138	0.012	0.04	0
[3.0, 3.9]	0.009	0.014	0.107	0.006	0.003	0.006
[4.0, 4.9]	0	0	0.072	0.003	0.003	0
[5.0, 5.9]	0	0	0.11	0	0	0.014
[6.0, 6.9]	0	0	0.014	0	0	0.049
[7.0, 7.9]	0	0	0.003	0	0	0.003
[8.0, 8.9]	0	0	0	0	0	0
[9.0, 9.9]	0	0	0	0	0	0
[10.0, 29.9]	0	0	0	0	0	0

Table 8.34: End-to-end latency for the cond. function average exp. with 40 tlv

Latency (ms)	Received (%)					
	Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
0.2	0	0	0	0	0	0
0.3	0.081	0	0	0.009	0	0
0.4	0.827	0.009	0	0.414	0.026	0
0.5	3.361	0.124	0.003	2.717	0.348	0.043
0.6	7.269	0.622	0.026	6.415	2.034	0.669
0.7	11.477	1.813	0.14	9.978	5.451	3.433
0.8	15.452	4.214	0.479	13.54	9.928	9.001
0.9	16.823	7.84	1.192	15.969	13.649	14.731
1	16.834	13.086	2.643	17.132	17.578	19.583
1.1	12.626	16.036	4.662	13.298	17.224	19.751
1.2	8.452	17.005	7.665	10.107	14.335	15.869
1.3	4.608	15.336	9.99	6.345	10.015	10.608
1.4	1.472	11.686	11.05	3.018	5.648	4.565
1.5	0.454	6.99	9.855	0.921	2.585	1.243
1.6	0.15	3.23	7.942	0.099	0.858	0.275
1.7	0.035	1.177	5.798	0.002	0.185	0.081
1.8	0.023	0.466	4.443	0.002	0.055	0.02
1.9	0.006	0.104	3.321	0	0.026	0.017
2	0.003	0.081	2.807	0	0.003	0.006
2.1	0	0.04	2.208	0.002	0.006	0.003
2.2	0	0.026	1.896	0	0.006	0.006
[2.3, 2.9]	0.043	0.098	6.785	0.024	0.029	0.023
[3.0, 3.9]	0	0.014	5.047	0.01	0.012	0.064
[4.0, 4.9]	0.003	0	4.2	0	0	0.009
[5.0, 5.9]	0	0	3.876	0	0	0
[6.0, 6.9]	0	0.003	2.392	0	0	0
[7.0, 7.9]	0	0	1.288	0	0	0
[8.0, 8.9]	0	0	0.082	0	0	0
[9.0, 9.9]	0	0	0.018	0	0	0
[10.0, 29.9]	0	0	0.19	0	0	0

Table 8.35: End-to-end latency for the cond. function average exp. with 80 tlv

Latency (ms)	Received (%)					
	Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
0.2	0	0	0	0	0	0
0.3	0.006	0	0	0.003	0	0
0.4	0.429	0	0	0.344	0	0
0.5	2.392	0.039	0	1.824	0.043	0.003
0.6	6.009	0.162	0.003	4.623	0.66	0.191
0.7	10.235	0.821	0.009	8.241	2.981	1.494
0.8	14.841	2.394	0.044	12.928	6.246	5.965
0.9	16.966	5.034	0.199	17.169	10.072	11.731
1	17.605	10.208	0.471	18.231	15.875	17.673
1.1	13.876	14.442	1.141	15.363	18.269	19.358
1.2	9.743	16.823	2.49	11.843	16.619	18.478
1.3	5.269	17.183	3.818	7.02	13.499	13.908
1.4	1.966	14.304	5.236	2.131	9.346	7.421
1.5	0.423	9.902	5.916	0.223	4.573	2.709
1.6	0.12	5.231	5.445	0.012	1.282	0.666
1.7	0.021	2.139	4.532	0	0.313	0.139
1.8	0.015	0.638	3.687	0	0.067	0.049
1.9	0.009	0.24	3.114	0	0.026	0.02
2	0	0.138	2.964	0	0.038	0.014
2.1	0.009	0.066	2.391	0	0.006	0.009
2.2	0.003	0.036	2.172	0	0.006	0.006
[2.3, 2.9]	0.042	0.135	9.653	0.035	0.067	0.035
[3.0, 3.9]	0.021	0.039	9.425	0.009	0.012	0.009
[4.0, 4.9]	0	0.009	13.574	0.003	0.003	0.003
[5.0, 5.9]	0	0.006	17.301	0	0	0
[6.0, 6.9]	0	0.003	3.79	0	0	0.003
[7.0, 7.9]	0	0.003	1.923	0	0	0
[8.0, 8.9]	0	0	0.259	0	0	0
[9.0, 9.9]	0	0	0.031	0	0	0
[10.0, 29.9]	0	0.006	0.411	0	0	0.116

Table 8.36: End-to-end latency for the cond. function average exp. with 120 tlv

Mean and Standard Deviation Analysis

In Table 8.37 and Figure 8.21, the mean and standard deviation for the four load patterns with and without condensation function are presented.

For reference subscriber Sub s_0 there is little difference for the different load patterns as well as the presence of a condensation function or not. The reason for this can be related back to the load scalability experiment (see Section 8.3), when only one SR is on the path between the publisher and subscriber. In this case the end-to-end latency is not affected much by the load on that SR. For reference subscriber Sub s_1 again there is not much difference with respect to the mean and standard deviation for the different load patterns. Although when comparing Sub s_1 without and with condensation function an improvement of about 0.2 ms for the end-to-end latency can be observed for the different load patterns, while the standard derivation remains the same. The real improvement by using a condensation function can be observed at Sub s_2. As the load gets larger, the cost of the multicasting starts to exceed the resources available at SR i_2. This can be observed as the mean and standard derivation get larger. The standard deviation is one order of magnitude larger for 120 tlv without condensation function compared to when the condensation function is used and the mean is three times larger than when the condensation function is present.

The one anomaly occurs for Sub s_2 for 40 tlv with filtering, where the standard deviation is relatively large compared to the run for 40 tlv. The reason for the relatively large standard derivation is due to a larger tail than the non filtering run. The reason for the tail is manyfold, but most likely it's due to a combination of one of the JVMs performing some optimization or some artifact of running on a general purpose OS incurring an extra delay for a very few of the events.

Total load variables	Mean (ms)						Standard deviation (ms)					
	Without cond. fun.			With cond. fun.			Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
40 w/ filt	0.956	1.189	1.305	0.977	1.101	1.102	0.224	0.241	0.684	0.239	0.231	0.245
40	0.959	1.21	1.368	0.912	1.055	1.079	0.223	0.234	0.311	0.207	0.215	0.233
80	0.92	1.189	2.152	0.956	1.06	1.067	0.224	0.235	1.486	0.229	0.221	0.2
120	0.946	1.254	3.299	0.973	1.13	1.119	0.22	0.269	1.984	0.212	0.218	0.192

Table 8.37: Mean and standard deviation for the cond. function average exp.

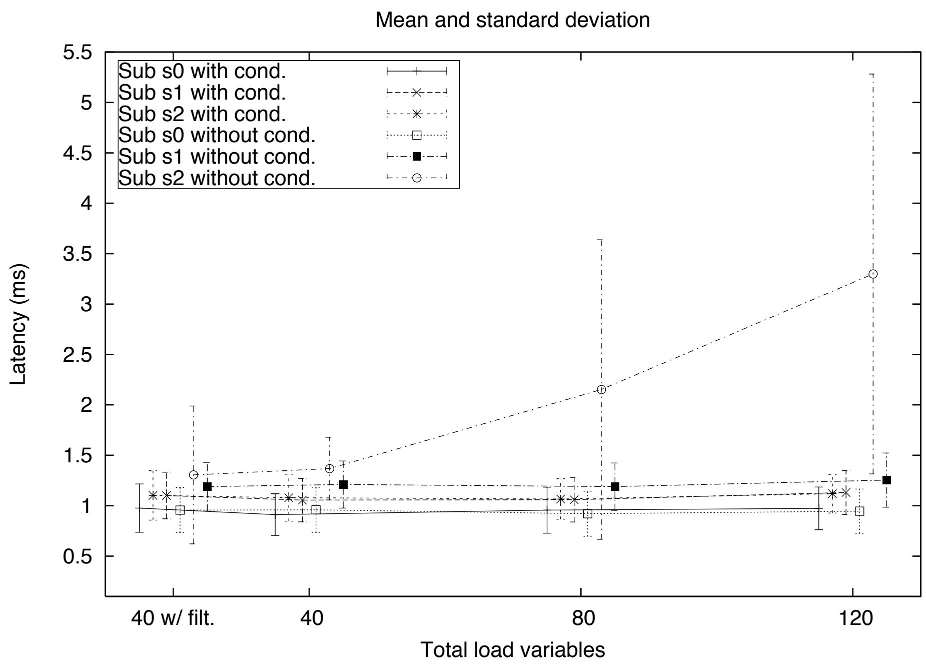


Figure 8.21: Mean and standard deviation for the cond. function average exp.

Throughput Analysis

The throughput for this experiment is shown in Table 8.38. When the condensation function is present, the throughput should stay the same for the different load patterns. This is not the case for 40 tlv with filtering, because in this case the condensation function is set to produce an event stream with an interval of 2 ms. As the load increases the throughput increases slightly when there is a condensation function; this should not happen. The reason for this is the triggering code for the condensation function. The triggering code performs two tasks: setting the value of the events received and triggering the calculator to produce a new event. In between performing the two tasks, the thread sleeps and relies on the JVM to wake it up to create a new event to the outgoing event stream or it gets signaled that a new event from its incoming streams has arrived. An improved future design of this triggering mechanism would be to split up the two tasks into two treads and create a more accurate output event stream. Although the throughput was not as accurate as predicted, it does not invalidate the importance of the condensation function, which is saving system resources.

Total load variables	Throughput (event/ms)			
	Without cond. fun.		With cond. fun.	
	Pub $p_{(0-3)}$	Sub $s_{(0-4)}$	Pub $p_{(0-3)}$	Sub $s_{(0-4)}$
40, w/ filt.	17.794	17.817	17.793	0.92
40	17.943	35.787	17.819	1.875
80	31.726	62.91	31.067	1.961
120	41.251	78.708	40.953	2.021

Table 8.38: Throughput for the cond. function average exp.

System Load Analysis

The system load is presented in Table 8.39 for the different load patterns without and with condensation functions for the different SRs present in the topology. At SR i_0 the system load is almost the same with or without the condensation function. The resources that the condensation function uses are saved by only sending out one event streams. The functionality of this condensation function is fairly simple, so it will depend on the functionality of the condensation function if this will be the case for a condensation function that has a more complex logic. For the remainder two SRs, namely SR i_1 and SR i_2, the system resources are saved by moving the application logic closer to the source of the necessary event streams.

Total load variables	System load (%)					
	Without cond. fun.			With cond. fun.		
	SR i_0	SR i_1	SR i_2	SR i_0	SR i_1	SR i_2
40, w/ filt.	5-15	3-10	15-23	9-15	1-3	1-3
40	17-23	10-15	30-40	18-23	1-4	2-5
80	30-33	20-25	55-65	32-38	1-5	2-5
120	42-50	25-30	70-72	42-46	1-7	2-5

Table 8.39: System load on the SRs for the cond. function 1 exp.

Event Drop Analysis

The event drop for this experiment is presented in Table 8.40. The event drop when there is no condensation function occurs at SR i_2. Again the reason for this was explained in the multicast experiment (see Section 8.5.1). It is a little peculiar that no events were dropped for 40 tlv, while some were dropped for 40 tlv with filtering, which only has about half of the throughput. For 80 and 120 tlv a large percentage of the events is dropped when there is no condensation function. This can be explained by the high system load on SR i_2 for these load patterns.

When the condensation function is used, no events are dropped. This is due to the reduction of the event streams and as it is seen in Table 8.39 the bottleneck (with respect to system load) SR i_2 is no longer a bottleneck.

Total load variables	Event drop (%)					
	Without cond. fun.			With cond. fun.		
	Pub $p_{(0-3)}$ - SR $e_{(0-3)}$	SR $e_{(0-3)}$ - SR $e_{(4-7)}$	SR $e_{(4-7)}$ - Sub $s_{(0-4)}$	Pub $p_{(0-3)}$ - SR $e_{(0-3)}$	SR $e_{(0-3)}$ SR $e_{(4-7)}$	SR $e_{(4-7)}$ - Sub $s_{(0-4)}$
40, w/ filt	0	0.135	0	0	0	0
40	0	0	0	0	0	0
80	0	1.167	0	0	0	0
120	0	4.505	0	0	0	0

Table 8.40: Event drop for the cond. function 1 exp.

8.7.2 *Condensation Function Alert*

The second condensation function takes as its input all the load variables that are published by Pub p_2 and Pub p_3 and publishes a new variable that is the average of the input variables. The difference between this condensation function and condensation function average is the triggering mechanism. In addition to the variables published for condensation function average, one more publisher is added (Pub p_4) which publishes an alert variable every 100 ms. Condensation function alert will use the alert received triggering mechanism, so that whenever a new event from one of the subscribed alert variables is received, the condensation function is triggered. When the condensation function is triggered, then it calculates the current average of all the non alert variables. This average is then used to publish an event stream of the current average whenever an event from an alert variable is received. Condensation function alert will be placed on SR i_0 and the load subscribers Sub s_3 and Sub s_4 will subscribe to the variable produced by condensation function alert instead of the load variables and the alert variable.

Similar to the other experiments the end-to-end latency, mean and standard deviation, throughput, system load and event drop are presented and analyzed.

End-to-End Latency Analysis

The results from the reference systems during the recording phase of the experiment are presented in Tables 8.41, 8.42, and 8.43 and depicted in Figures 8.22, 8.23, and 8.24. First, the end-to-end latency is analyzed when no condensation function is used and then when the condensation function is used. After that, the improvement, with respect to end-to-end latency, of using the condensation function for the different instrumentation points in the topology is presented.

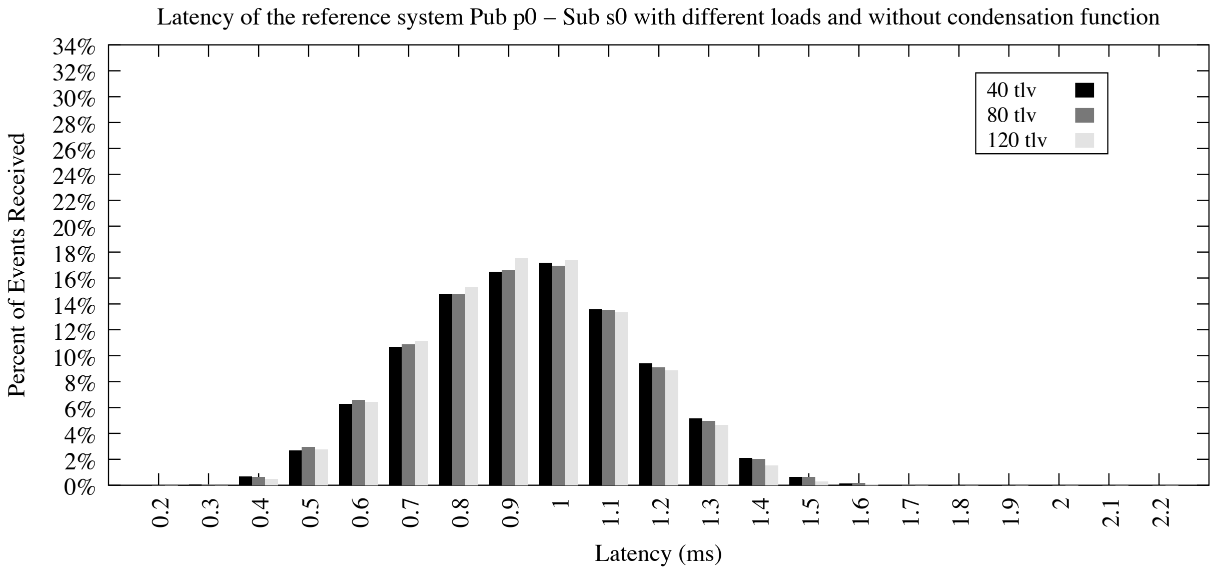
According to the experimental results, there is not much difference in the end-to-end latency for Sub s_0 for the different load patterns. This was also observed in the load scalability experiment (see Section 8.3.) For Sub s_1, as the number of load variables increases the end-to-end latency also increases slightly. The difference between Sub s_0 and Sub s_1 is that Sub s_1 has one more SR hop. This additional hop causes the different load patterns to exhibit different end-to-end latency, something that was also seen in the hop scalability experiment. For Sub s_2, as the load increases the end-to-end latency increases and for 80 and 120 tlv it starts to break down (i.e. the end-to-end latency is very large).

Mean and Standard Deviation Analysis

In Table 8.44 and Figure 8.25, the mean and standard deviation for the three load patterns with and without condensation function are presented. There is very little difference for the different load patterns when the condensation function is present with respect to the mean and the standard deviation. For Sub s_1 and Sub s_2 there is little difference between the different load patterns when there is no condensation function. However, for Sub s_3 there is a large difference for both the mean and the standard deviation compared to condensation function 1. In that case there was an anomaly with the standard derivation for Sub s_2 when there was no condensation function.

Throughput Analysis

The throughput for this experiment is shown in Table 8.45. Compared to the condensation function average experiment (see Section 8.7.1) the throughput is the same for all the load patterns in the



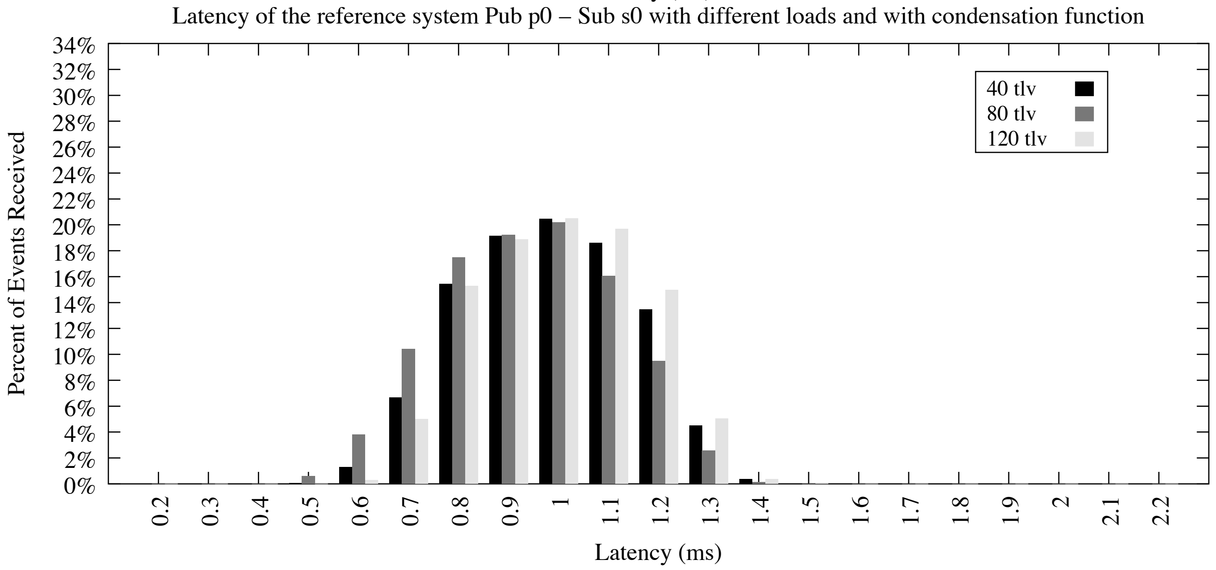
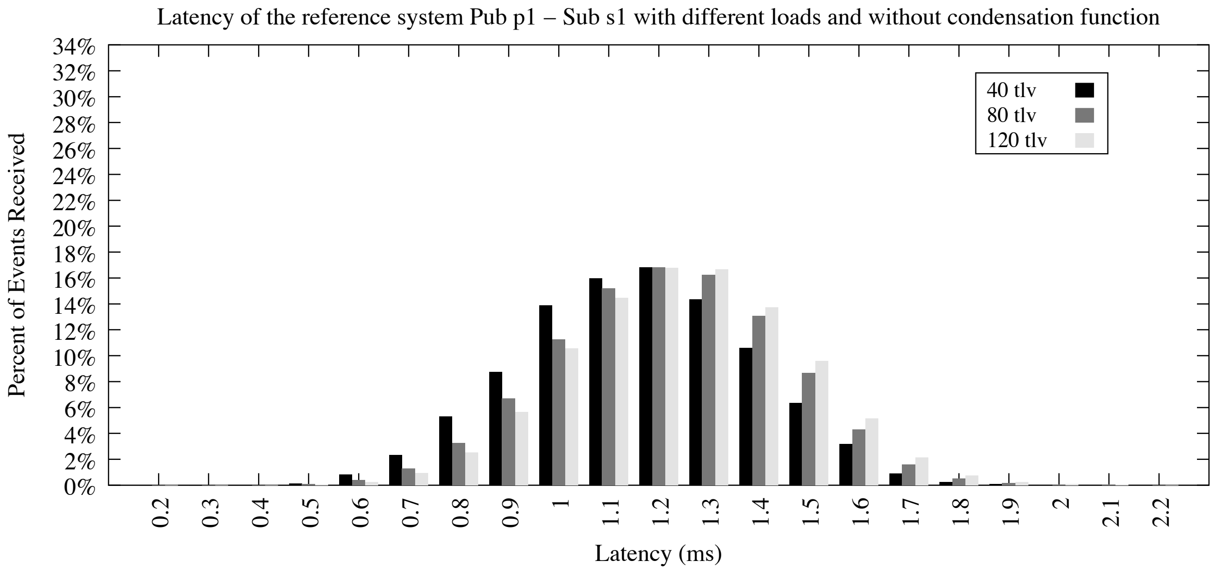


Figure 8.22: End-to-end latency for the cond. function alert exp. for Sub s_0



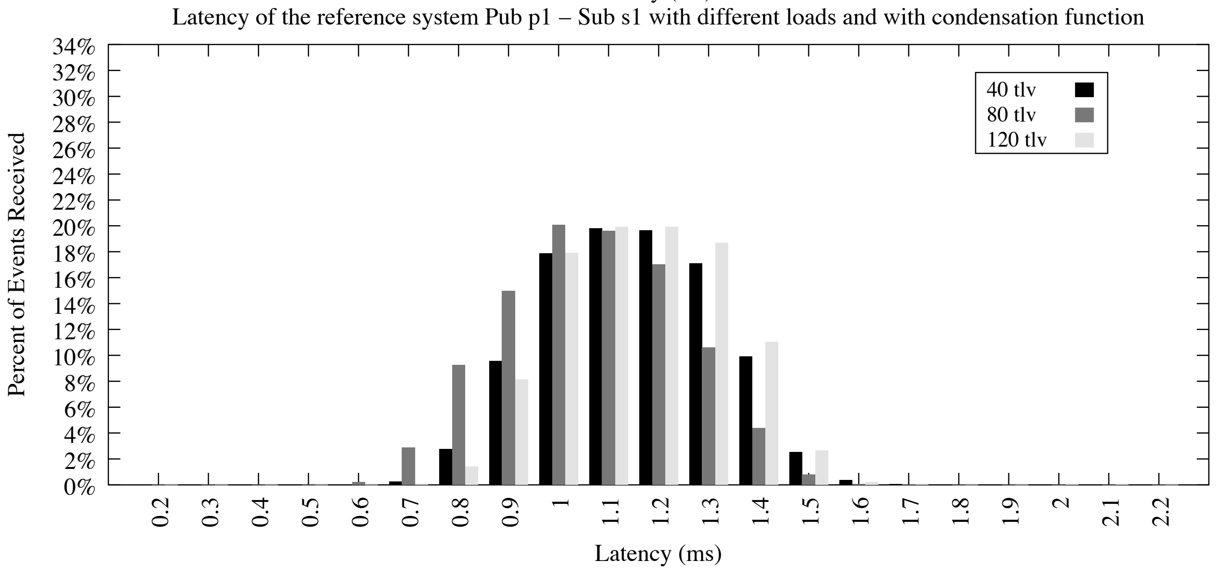


Figure 8.23: End-to-end latency for the cond. function alert exp. for Sub s_1

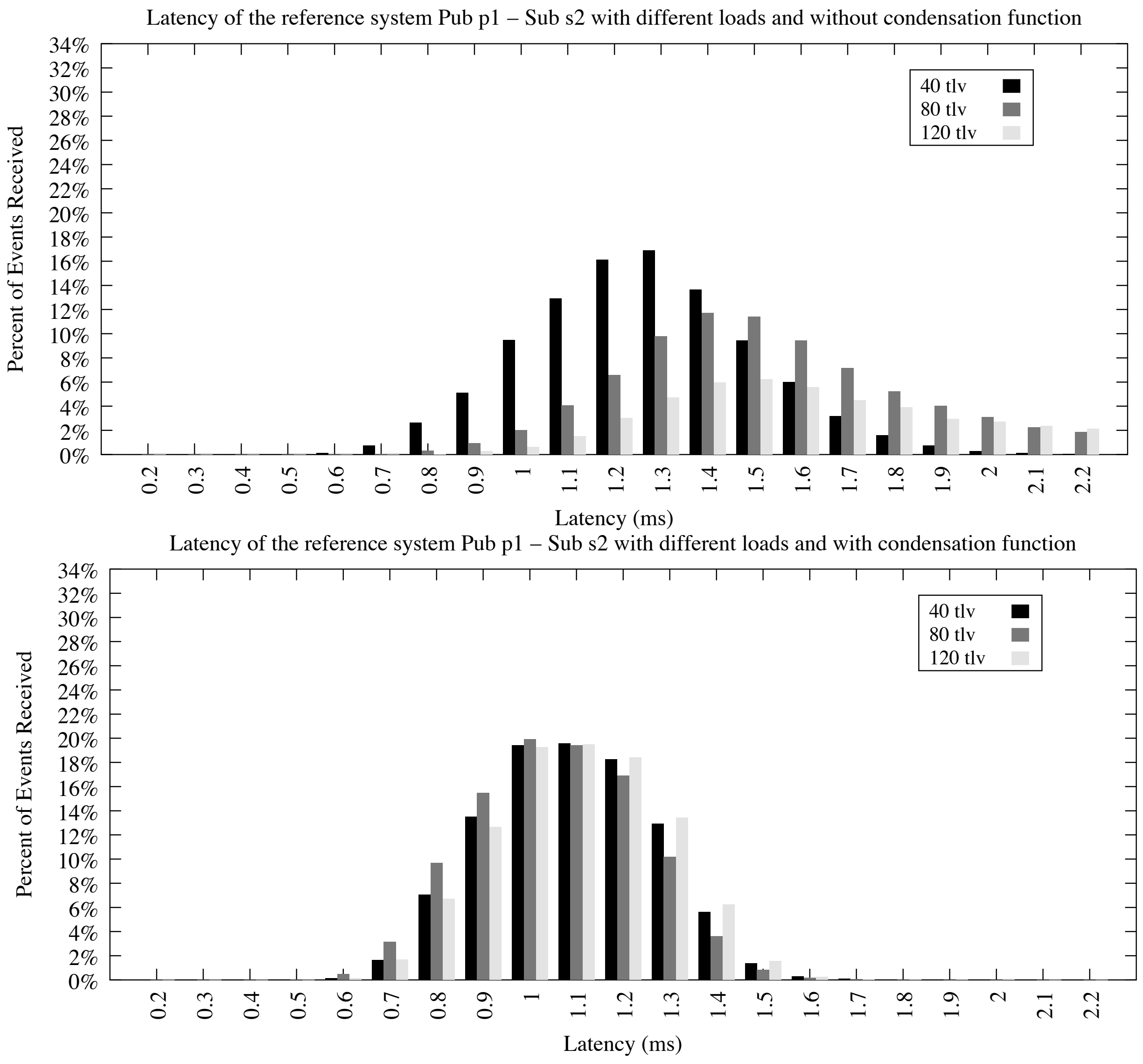


Figure 8.24: End-to-end latency for the cond. function alert exp. for Sub s_2

Latency (ms)	Received (%)					
	Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
0.2	0	0	0	0	0	0
0.3	0.067	0	0.003	0	0	0
0.4	0.677	0	0.014	0	0	0
0.5	2.694	0.014	0.15	0.058	0.003	0
0.6	6.28	0.145	0.819	1.282	0.122	0.012
0.7	10.685	0.756	2.329	6.652	1.627	0.252
0.8	14.763	2.636	5.311	15.445	7.042	2.784
0.9	16.47	5.131	8.762	19.133	13.5	9.577
1	17.191	9.496	13.914	20.461	19.416	17.851
1.1	13.562	12.923	15.982	18.619	19.585	19.815
1.2	9.415	16.122	16.809	13.467	18.277	19.632
1.3	5.152	16.904	14.359	4.491	12.932	17.106
1.4	2.118	13.645	10.616	0.377	5.609	9.934
1.5	0.648	9.453	6.355	0.003	1.378	2.538
1.6	0.136	5.994	3.173	0.003	0.302	0.357
1.7	0.041	3.178	0.911	0	0.11	0.055
1.8	0.017	1.605	0.26	0	0.029	0.023
1.9	0	0.739	0.107	0	0.015	0.003
2	0.003	0.307	0.032	0	0	0.003
2.1	0.009	0.133	0.012	0.003	0.003	0.003
2.2	0	0.061	0.012	0	0	0
[2.3, 2.9]	0.046	0.046	0.136	0.006	0.044	0.044
[3.0, 3.9]	0.026	0.017	0.084	0	0.009	0.009
[4.0, 4.9]	0	0.006	0.087	0	0.003	0
[5.0, 5.9]	0	0	0.055	0	0	0
[6.0, 6.9]	0	0	0.041	0	0	0
[7.0, 7.9]	0	0	0.026	0	0	0
[8.0, 8.9]	0	0	0.017	0	0	0
[9.0, 9.9]	0	0	0.052	0	0	0
[10.0, 29.9]	0	0	0.261	0	0	0

Table 8.41: End-to-end latency for the cond. function alert exp. with 40 tlv

Latency (ms)	Received (%)					
	Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
0.2	0	0	0	0	0	0
0.3	0.043	0	0	0	0	0
0.4	0.657	0	0	0.003	0	0
0.5	2.946	0	0.081	0.581	0.012	0
0.6	6.574	0.009	0.405	3.811	0.477	0.212
0.7	10.865	0.076	1.299	10.427	3.162	2.87
0.8	14.754	0.321	3.259	17.483	9.683	9.259
0.9	16.609	0.941	6.697	19.239	15.49	14.985
1	16.945	2.022	11.261	20.203	19.905	20.077
1.1	13.522	4.062	15.215	16.038	19.418	19.611
1.2	9.1	6.605	16.809	9.493	16.911	17.033
1.3	4.977	9.784	16.262	2.582	10.189	10.609
1.4	2.017	11.742	13.082	0.127	3.621	4.391
1.5	0.66	11.4	8.662	0	0.822	0.799
1.6	0.179	9.451	4.33	0	0.177	0.056
1.7	0.046	7.157	1.621	0	0.05	0.018
1.8	0.02	5.228	0.521	0	0.012	0.006
1.9	0.009	4.03	0.174	0	0.006	0.003
2	0.006	3.121	0.072	0	0.006	0.006
2.1	0.003	2.256	0.064	0	0	0.006
2.2	0.006	1.891	0.035	0.003	0.003	0.003
[2.3, 2.9]	0.052	0.096	6.862	0.003	0.053	0.047
[3.0, 3.9]	0.009	0.049	4.249	0.009	0.006	0.006
[4.0, 4.9]	0	0.006	3.145	0	0	0
[5.0, 5.9]	0	0	2.42	0	0	0
[6.0, 6.9]	0	0	1.797	0	0	0
[7.0, 7.9]	0	0	0.97	0	0	0
[8.0, 8.9]	0	0	0.058	0	0	0.003
[9.0, 9.9]	0	0	0.003	0	0	0
[10.0, 29.9]	0	0	0.397	0	0	0

Table 8.42: End-to-end latency for the cond. function alert exp. with 80 tlv

Latency (ms)	Received (%)					
	Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
0.2	0	0	0	0	0	0
0.3	0.02	0	0	0	0	0
0.4	0.473	0	0.003	0	0	0
0.5	2.741	0	0.053	0	0.003	0
0.6	6.446	0.003	0.239	0.275	0.138	0
0.7	11.134	0.012	0.943	4.994	1.686	0.03
0.8	15.332	0.078	2.544	15.286	6.69	1.423
0.9	17.521	0.273	5.679	18.863	12.669	8.148
1	17.369	0.648	10.556	20.516	19.249	17.895
1.1	13.36	1.551	14.473	19.688	19.497	19.922
1.2	8.86	3.026	16.795	14.963	18.421	19.925
1.3	4.674	4.719	16.657	5.027	13.438	18.666
1.4	1.511	5.968	13.719	0.374	6.245	11.044
1.5	0.307	6.229	9.599	0.009	1.566	2.651
1.6	0.101	5.576	5.144	0	0.263	0.194
1.7	0.025	4.498	2.153	0	0.06	0.024
1.8	0.02	3.929	0.771	0	0.015	0.006
1.9	0.006	2.956	0.262	0	0.009	0
2	0	2.709	0.11	0	0.012	0.009
2.1	0.008	2.381	0.07	0	0.006	0.015
2.2	0	2.132	0.02	0	0.006	0.003
[2.3, 2.9]	0.073	0.152	9.095	0.006	0.027	0.021
[3.0, 3.9]	0.02	0.028	9.063	0	0.015	0.006
[4.0, 4.9]	0	0.017	11.686	0	0.003	0
[5.0, 5.9]	0	0.008	16.341	0	0	0
[6.0, 6.9]	0	0	3.894	0	0	0
[7.0, 7.9]	0	0	2.332	0	0	0
[8.0, 8.9]	0	0	0.436	0	0	0
[9.0, 9.9]	0	0	0.012	0	0	0
[10.0, 29.9]	0	0.003	0.453	0	0	0

Table 8.43: End-to-end latency for the cond. function alert exp. with 120 tlv

Total load variables	Mean (ms)						Standard deviation (ms)					
	Without cond. fun.			With cond. fun.			Without cond. fun.			With cond. fun.		
	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2	Sub s_0	Sub s_1	Sub s_2
40	0.941	1.167	1.317	0.985	1.151	1.095	0.226	0.235	0.711	0.168	0.175	0.183
80	0.937	1.222	2.054	0.94	1.066	1.059	0.226	0.237	1.479	0.175	0.193	0.189
120	0.931	1.247	3.212	1.001	1.165	1.101	0.219	0.248	1.92	0.162	0.168	0.183

Table 8.44: Mean and standard deviation for the cond. function alert exp.

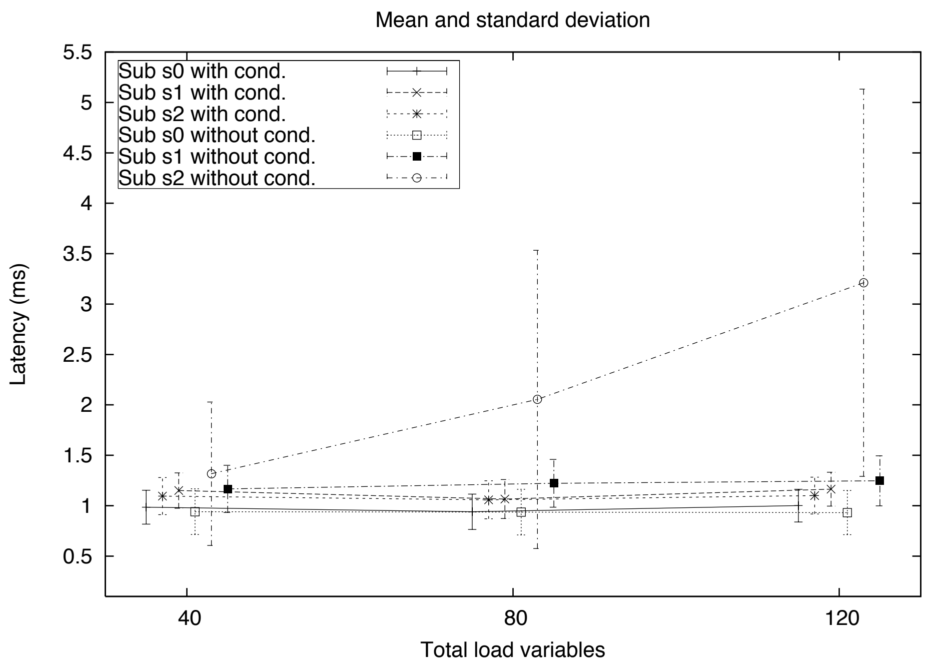


Figure 8.25: Mean and standard deviation for the cond. function alert exp.

presence of the condensation function. This is because for this condensation function a different triggering mechanism is used, which only gets triggered when an event from the alert stream is received.

Total load variables	Throughput (event/ms)			
	Without cond. fun.		With cond. fun.	
	Pub $p_{(0-3)}$	Sub $s_{(0-4)}$	Pub $p_{(0-3)}$	Sub $s_{(0-4)}$
40	17.905	35.649	17.658	0.077
80	31.221	61.42	31.503	0.077
120	41.178	79.278	41.257	0.077

Table 8.45: Throughput for the cond. function alert exp.

System Load Analysis

The system load is presented in Table 8.46 for the different load patterns without and with condensation functions for the different SRs present in the topology. For SR i_0 its system load is less when the condensation function is present. For condensation function 1 the system load was about the same irrespectively of the fact that the condensation function was used or not. The reason that the system load is less when the condensation function is used for condensation function 2, is because this condensation function produces events only about every 100 ms. Condensation function 1 produced an event about every 1 ms.

For SR i_1 and SR i_2 the system load is very low for all the load patterns when the condensation function is used. This is of-course due to the presence of the condensation function. For SR i_1, when there is no condensation function, the system load increases as the number of variables increases. If SR i_0 and SR i_1 is compared when there is no condensation function, it can be seen that the load on SR i_1 is less than that on SR i_0. The reason for this was explained in Section 8.4.1 and is because SR i_0 receives load event streams from two incoming communication links (two threads), while SR i_1 receives its load event streams only from one incoming communication link.

For SR i_2 the system load is a lot larger than that of the other SRs when there is no condensation function. The reason as also presented in Section 8.5.1; this SR is a splitting point of the multicast tree and because of the implementation this split is quite expensive.

Total load variables	System load (%)					
	Without cond. fun.			With cond. fun.		
	SR i_0	SR i_1	SR i_2	SR i_0	SR i_1	SR i_2
40	19-22	11-16	30-40	13-17	0.2-0.6	0.2-0.6
80	30-37	18-25	52-65	24-27	0.2-0.5	0.1-0.3
120	42-45	25-29	69-71	35-38	0.1-0.6	0.1-0.2

Table 8.46: System load on the SRs for the cond. function alert exp.

Event Drop Analysis

The event drop for this experiment is presented in Table 8.47. No events are dropped when the condensation function is present. When there is no condensation function the system load on SR i_2 gets too large and the incoming socket buffers are too small and events are getting dropped. For 80 and 120 tlv the number of events is unacceptable for any real setting. Although, the purpose for these experiments was to test the framework for its scalability and its behavior for different configurations, the number of events dropped can be easily fixed by increasing the socket buffer size in the underlying OS.

Total load variables	Event drop (%)					
	Without cond. fun.			With cond. fun.		
	Pub $p_{(0-3)}$ - SR $e_{(0-3)}$	SR $e_{(0-3)}$ - SR $e_{(4-7)}$	SR $e_{(4-7)}$ - Sub $s_{(0-4)}$	Pub $p_{(0-3)}$ - SR $e_{(0-3)}$	SR $e_{(0-3)}$ SR $e_{(4-7)}$	SR $e_{(4-7)}$ - Sub $s_{(0-4)}$
40	0	0.175	0	0	0	0
80	0	1.14	0	0	0	0
120	0	3.886	0	0	0	0

Table 8.47: Event drop for the cond. function 2 exp.

CHAPTER 9

RELATED WORK

Related work to the GridStat framework is presented only for the closest related middleware frameworks. Other middleware frameworks based of the publish-subscribe paradigm include [13, 70, 22, 57, 26, 7, 14, 11, 28, 19], specifications for such frameworks [35, 33, 50] and industry frameworks [66, 69] exist that provide for status dissemination but they are not closely related to the GridStat framework. This is because these frameworks are designed to store the information as well as to forward it to the subscribers. In addition these frameworks can not take advantage of the semantic of the information that is published because this is not known.

First three systems that are related to the GridStat framework are presented; those are the Data Distribution Service, Piece-wise Asynchronous Sample Service, and QuO's Resource Status Service. Then two categories of related work is presented. The first is related work that belong to the category of aggregation and dissemination of streams, and the second category are distributed databases for continues queries on streams of data items.

9.1 Data Distribution Service

The Data Distribution Service (DDS) for Real-Time System Specification by OMG [34] is the closest related work to the GridStat framework. The DDS specification specifies the interfaces for a service similar to the GridStat framework, but it is more general. It provides data-centric communication for signals, streams[1] and states, compared to GridStat which is specialized for streams. DDS has two levels of interfaces: the Data-Centric Publish-Subscribe (DCPS) and the Data-Local Reconstruction Layer (DLRL). The DCPS interface is provided to efficiently deliver information to the interested subscribers, while the DLRL interface is targeted to simplify the interaction between the application layer and the DDS. The OMG only specifies the interfaces

[1]In GridStat there is no distinction between signals and streams

that the end-points use to interact with the framework, but it does not specify how the framework provides the services to the end-points. The two frameworks conforming to the DDS standard are the Network Data Distribution Service (NDDS) [55] by Real-Time Innovations and SPLICE [71] by Thales Naval Netherland. Both of these implementations are targeted to work in in the LAN domain and are taking advantage of IP broadcast to avoid centralized management of the system.

The GridStat framework, as presented in Chapter 2, provides part of the DCPS interface, but GridStat is specialized for tightly managed dissemination of event streams with QoS in the wide-area setting. The DCPS provides hints of QoS or best effort service, although it is not specified how the resources are managed. It should be noted that there is nothing in the specification that prevents DDS to be used in a WAN setting, but the current implementations of the service is for a LAN setting.

The DLRL layer provide for a SQL like query language to implement application logic into the middleware layer. The difference with the condensation function mechanism presented in Section 2.2.6 is that in the DDS framework the application logic is only used by a single subscriber and evaluated at the subscriber end. In GridStat, the condensation function produces a new status variable that is used by any of the other subscribers in the system. The condensation function is also placed close to the source of the information to save resources as shown in the Section 8.

With some changes, the GridStat framework could potentially conform partially to the DDS specification that involves signals, streams and the DLRL interfaces, but it would not be able to provide the interfaces that create a global “data space”.

9.2 Piece-wise Asynchronous Sample Service

PASS (Piece-wise Asynchronous Sample Service) developed at BBN [79] is a publish-subscriber status dissemination middleware. It is designed to adapt by moving application logic into the middleware layer so as to provide end-to-end QoS. PASS is a service that provides information about the status of communication network resources, indicating their availability or not. The service consists of a set of connected PASS servers that route data in the form of records, that encapsulate name-value pairs. PASS provides low-level status dissemination mechanisms as for boolean variables, but does not provide the QoS management of the resources in the system like GridStat.

The publishers and subscribers register to be members of groups and can then register to write and read records to and from the group. There are policies in the groups that restrict insertion of data to the group (Merge policy) and update of values to the readers of a group (Send policy). A limitation, compared to the GridStat framework, is that all the subscribers in a group have the have the same QoS parameters. These policies are evaluated and enforced at the PASS servers, like placing a condensation function at the status router in the GridStat framework. The policies can be seen as a restricted set of condensation functions where the parameters can be set. Some policies that are supported by PASS are:

- **Translation** of fine grained data into coarser grained, for instance, a float into an int. The coarser grained data will change less frequently.
- **Inference** of a data value from other data values.
- **Summary** of many data values into a new record, for instance, the average of a group of values could be combined into a new record.
- **Filtering** of data values, for instance, there might be a threshold so that values below it will be filtered out.
- **Priority** of some data values, for instance, a record for a critical device might have a higher priority than records produced by non-critical devices.

As a summary, here are a list of the fundamental difference between PASS and GridStat, in that PASS:

- Provides for low-level status dissemination for boolean variables only.
- Does not provide QoS management of the resources.
- Does not provide redundant paths.
- Does not allow for different subscribers to get different QoS (rate, latency, number of redundant path).

9.3 QuO's Resource Status Service

Quality Objects QuO's Resource Status Service (RSS) [80] is a very pragmatic framework for the collection of different kinds of resource monitoring inputs. It supports both in-band (synchronous to a remote object call and along its data path) and out-of-band (asynchronous to a remote object call and from sources outside that data path) measurement inputs. RSS has a tiered architecture, which from bottom to top has well-defined integration points for collection, translation, integration, and inferring (via models). At the top this is input into a QuO system condition object, which of course means that it can be used as input to a QuO contract. The application of course needs some kind of "model" of what should be happening on which to base any adaptation, whether this "model" is rigorous or informal.

The similarities for the RSS and the GridStat framework are that they both disseminate status information in the wide-area setting. The difference is that the RSS is there to gather status information about the resource usage and availability for an application as to provide QoS to the application data. In GridStat the status information is the application data and hence the QoS is given to the status information. The RSS is also designed to be a general purpose service to any application using shared resources, while GridStat is specialized to only disseminating status information in a setting where all the resources are controlled by the framework.

9.4 Aggregation and Dissemination of Streams

The Solar system [15] provides context collection, aggregation, and dissemination of streams of events that are encoded using XML [73]. It has a network of planets that is controlled by one star, publishers that publish streams of events and subscribers that subscribe to event streams. The event streams are forwarded only through the planets, while the star is responsible for setting up subscriptions. The aggregation of information is done by a directed acyclic graph of operators. Each operator can be either stateless or stateful and subscribes to one or more event streams and produces a new event stream. The subscription request that is given to the star is organized as a tree of operators and publication sources. The star can find operators that are already present on a planet and reuse them while the remaining operators will have to be loaded into planets and be instantiated. The events are forwarded in a unicast manner between the planets, while within the plane the events are multicasted to all the operators that are instantiated on the specific planet. Physical mobility of data sources is supported, i.e., if the information source is moved then the subscriptions to the source are removed, the new location of the source is detected and new subscription are established. Solar is organized similar to the GridStat framework with a data plane and a management plane, but in Solar there is only one node in the management plane, although it is mentioned in the future work that there are plans for the support of more than one star. Solar is targeted for a different application domain than GridStat. It is targeted to monitor devices and their location i.e., track the movement of a person in a building, alert if equipment is removed from a room without a prior moving order. (The load in the system that Solar target can not be predicted aprior, so it can not provide the strong end-to-end QoS guarantees.)

A framework for detecting Composite Events (CE) as extension to pub-sub middleware is presented in [59, 58]. The framework provides a generic language to specify CE and therefore it provides a higher level to the application layer instead of subscribing to the individual event streams and implement the logic at the application layer. It is assumed that the underlying pub-sub middleware is organized as a network of brokers where CE detectors can be loaded. After the CE are specified, they are translated into the core CE language, which is based on regular expressions and finite state automata, and then compiled. The CE detector is deployed to one of the event brokers by having the event brokers collectively deciding based on the distribution policy. The CE can be split into subexpression to improve reusability, so a CE can be composed of using only other CE and not use any primitive event streams. The CE framework could use the GridStat framework as its underlying pub-sub framework. Both the condensation function and the framework presented have the same objectives: to present a higher level abstraction to the application layer and at the same time save resources in the system. While this work has concentrated on providing the proper programming/specification language for such systems, the GridStat framework has concentrated on managing the resources as to provide end-to-end QoS guarantees.

The *event web* [48] software architecture is designed to be used for crisis management and provides specification, deployment, observation and management of large numbers of persistent software objects that generate, process or consume streams of events. The architecture consists of four parts: reactive objects that process event streams, a dissemination network that distributes the events to and from the objects (content based pub-sub), a directory service that provides information about the event streams, and a service layer to provide services to the objects. The system has event generators that generate event streams, event processors that receive and process event streams and even generate a new event stream and finally event consumers that consume event streams. The event processors can act as both event generators and event consumers by integrating and processing event streams into a new event stream. A key aspect of the *event web* is that the user only specifies the event processors as a state-transition machine in XML. Therefore, when a crisis occurs, the task force members can specify the conditions that they are interested in and the corresponding action using ”when-then“ rule. This specification is then propagated into the *event web* as processors. This work concentrates on providing the right level of abstraction to the application layer so that non-experts can specify the information that they are interested in during a crisis. The framework is therefore more flexible and easier to deploy, but it lacks the management of the resources that are available in the system which can result in the system overloading creating a second crises.

Other work that fits into this category but is different from GridStat in many aspects includes: XPushMachine [36], NaradaBroker [54], DaQuinCIS [49], ONYX [20], Xyleme [76], and StreamBase [65].

9.5 Distributed Databases for Continues Queries on Streams of Data Items

Work done on distributed databases regarding streams of data items that provide continuously queries has some similarities to the GridStat framework and the condensation functions. Before presenting the different frameworks, there are some common differences between these frameworks and GridStat. Firstly, the database frameworks have requirements for storing the data streams and maintaining consistency, while in the GridStat framework the requirement is to disseminate streams according to the QoS that is specified. Secondly, the database frameworks presented are targeted to support highly dynamic information requests while in the GridStat framework it is assumed that the subscriptions stay mostly static. The similarities between the database frameworks and GridStat is that they disseminate event streams (continuous queries) and provide for filtering and aggregation between the streams (condensation function). Some systems belonging to this category include: COUGAR [77], dQUOB [60], STREAM [52], Sophia [74], PHI [17], and Antarctic Monitoring [8], but the following three are described in more detail.

Tapestry [68] introduces the notion of continuously queries for append-only databases. The queries run continuously, trying to match newly added information and notify the user when a match is found. OpenCQ [46] is a distributed event driven context-sensitive database that provides continual queries. The queries are of the type ”When the stock price of IBM stock drops by 10% then notify me“. NiagaraCQ [16, 15] is similar to OpenCQ, but optimizes by exploring the similarities between queries in order to increase system scalability. Telegraph continuously adaptive continuous query (CACQ) [47] uses the continuously adaptive query processing eddy operator [4] to provide continuous adaptivity to the changing query workload, data delivery rates, and overall system performance. The database frameworks presented here are similar to the condensation function in that application logic on streams is moved into the database layer. It is different in that the result from the continuous queries is not published as a new status variable. The optimizations that are performed is to find sub-parts of multiple queries so that those parts are only performed once.

The IrisNet [31] and Borealis Stream Processing Engine (SPE) [2] are distributed databases specialized to process, store and distribute streams. IrisNet is organized as a hierarchy of organizing agents (OA) that receive streams from sensing agents (SA). It provides for loading senslets into the SA that can change the stream that is forwarded, like filtering, compressing, etc. The difference to the condensation function is that the senslet can only operate on one stream. Compared to the GridStat framework, the IrisNet combines the management and data plane, and that reduces the predictability of the resources available to the data flow. Borealis combines the stream processing functionality of Aurora [1] and the distributed functionality of Medusa [78]. It is organized as a network of Borealis nodes, that process streams that perform queries on. The nodes collaborate and constantly try to optimize to remove network and processing bottlenecks by distributing queries or fragments of the queries and constantly reevaluate the performance. It also supports QoS with respect to which tuples and which operators to prioritize in execution.

A framework for distributed event mining is presented in [56]. The system is organized as an Event Processing Networks (EPN) and it consists of any number of Event Processing Agents (EPA) which are either event sources, event processors, or event viewers. Events flow from sources through processors to viewers. The Processors are typically filters (drop unwanted events) or maps (aggregates multiple events in a new event), but user defined event processor can also be inserted into an EPN. The system has a shared data store where events are stored in case agents want to access past events. The EPAs get notified when new events are written to the data store through a communication service. The flexibility of the system is that it provides for real time pattern queries similar to the continuous queries except that the EPA are specified in the RAPIDE [67] pattern language.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

This dissertation concludes by presenting the contribution of the research work and some potential future directions.

10.1 Conclusions

This dissertation presented the architecture of a status dissemination middleware along with the framework named GridStat, which is a prototype implementation of this architecture. The framework is specialized to disseminate mostly static sets of periodic streams of events from publishers to subscribers with end-to-end QoS. The framework provides strong typing and multiple interaction models at the subscriber side to meet the application needs. A baseline set of mechanisms was presented to show the extensibility of the framework. Other mechanisms could easily be added; either general or application domain specific mechanisms.

The dissertation first gave a general introduction to distributed systems, middleware, and publish-subscribe in Chapter 1. These are the pillars that this dissertation work is built on.

In Chapter 2 the architecture of the GridStat framework was presented which is our implementation of a prototype SDM framework. The architecture consists of two parts, the management plane and the data plane. The reason for this separation was explained and motivated by the requirements of scalability and the need to provide end-to-end QoS. This chapter also gave an overview of the baseline mechanisms. The mechanisms were divided into two categories: mechanisms to enhance resilience and mechanisms to enhance efficiency.

The design of the framework then followed in Chapter 3. The interaction model and API that is given to the application level were explained. The API and interaction model choices reflect the desire to make the framework as intuitive and simple as possible for an application developer to develop applications that would use the framework. By providing this high-level API, the application developer gets to concentrate on the application logic rather than on the distributed aspect of the application.

The chapters 4 - 7 presented in detail the motivation, design, and implementation of the major mechanisms of the framework.

The first mechanism, the multicast rate-filtering routing mechanism followed which provides for deterministic filtering of temporally related variables along with built in multicast. This mechanism provides for efficiency by only forwarding the wanted events and only forward one copy over any of the event channels. In addition, the deterministic filtering provides for the capability of taking snap-shots of temporally related variables. The limited flooding mechanism, provides for the functionality of flooding status variables to a limited region. The operating mode mechanism was presented next, and it provides for adaptability in that it can change between different subscription sets without violating the QoS requested by the different sets. The last mechanism presented is the condensation function mechanism which provides for moving application layer logic into the middleware. This provides for enhanced efficiency as it is possible to move the application logic closer to the source of the input which saves resources further down the path. Combined these mechanisms provide for efficient, reliable, and adaptable enhancements to the base framework.

The experiments conducted on the prototype implementation show that the architectural choices that were initially made fit the application domain envisioned. Although the topologies in the experiments were limited due to lack of testbed machines, the performance measurements show that the system should scale well both with respect to load and the number of status routers. It is observed that a status router approaching its maximum capacity scales linearly as more status routers are added on the path from the publisher to the subscriber. The framework should therefore behave reasonably well in a wide-area setting since the QoS management will be able to predict the delay at each hop and have the complete knowledge of how many hops are necessary in the path. As the QoS management can control/enforce the load in the system, the end-to-end QoS requested by the subscribers can be guaranteed (if operating within the fault model).

The experiments that were conducted for the condensation function showed that placing application logic into the middleware saved resources. The benefits are not only observed by the participants that are interested in the calculation that the condensation function provides, but also any other participant that shares some of the resources with the condensation function participants.

In summary, the key contributions of this dissertation are the following:

- The design and implementation of an architecture for a Status Dissemination Middleware framework
- The design and implementation of mechanisms to enhance the efficiency of this framework
- The design and implementation of mechanisms to enhance the resilience of this framework
- An experimental evaluation of the GridStat's performance under different topologies

Furthermore, the thesis of this dissertation:

Thesis: middleware frameworks can provide efficient and resilient delivery of status information across a wide-area distributed computing system.

has been demonstrated by the research that supported the design, implementation, and evaluation of GridStat. From the hop scalability experiment (see Section 8.4) it was observed that a status router could forward 50,000 events/second over 7 hops with a relative tight end-to-end latency variance.

10.2 Future Work

The future work is divided into two categories. The first involves the future improvements and investigations that can be done to the data plane, while the second category involves the future work for the management plane.

10.2.1 Data Plane

The future work for the data plane can be divided into three categories. The first category is to further optimize the implementation of the (edge) status router. The second category is related to the mechanisms that the (edge) status routers provide and how these can be extended. Finally, the third category is vulnerabilities and security in the current implementation of the data plane and how these can be extended.

Implementation of the Data Plane

As was discovered in the Multicast mechanism experiment (see Chapter 8.5) the performance of the framework deteriorated when multicast occured. The reason for this was explained in Section 8.5.1. So an improvement to current Java implementation would be to modify the `ByteBuffer` class, so that it will give access to the "raw". In this case a wrapper class can be written that will keep track of the references to the `ByteBuffer`. The instances of the wrapper class will then be kept in a buffer pool and recycled as to not initiate the garbage collector. This should remove the deteriorating performance that is observed in the current implementation when multicast is used.

In order to get a more predictable performance of the (edge) status router they could be ported to C++ and run on a real time operating system like QNX [62]. Another option would be to port the (edge) status routers into dedicated hardware like the Intel network processor [72]. With dedicated hardware or a real time operating system the different tasks that a (edge) status router has to perform could be made deterministic. The forwarding task could be given dedicated processors as well as the different condensation functions that are loaded on a ESR.

Mechanisms in the Data Plane

For each of the mechanisms that are present in the data plane a short description of some possible future work is discussed. Some of this work is optimization of the current implementation while others are how the mechanisms can be extended.

Multicast Rate-filtering Routing Mechanism If the (edge) status router is moved into dedicated hardware then the organization of the routing table could be different. If each of the incoming and outgoing event channels has a dedicated processor, then the routing algorithm could be divided into two stages, where the first stage is to forward the status event to all potential outgoing event channels. Then the second stage will be performed in each of the outgoing event channels to decide if a specific event is to be forwarded on this event channel or filtered. This will add a lot more concurrence, but it requires multiple processor units.

Limited Flooding Mechanism There is one optimization for the limited flooding mechanism that can improve the performance of this mechanism. For instance in the case that a variable is set to be flooded to level x. When an event from this variable is forwarded through a SR then the routing algorithm does the following. First the event is forwarded to all the event channels that has a level less than or equal x. Secondly the routing algorithm has to route the event to event channels that are at a higher level x. Currently all the (edge) status routers perform this extra routing. The only place where this is necessary is the border status routers, so in the implementation of the status routers a third kind should be created that will be the only one that will perform the flooding in this way. The other two types, namely edge status router and status router, will only flood the event to all its outgoing event channels.

Another area to investigate is whether another dimension should be added to this mechanism, namely filtering. Should the flooding also be subject to filtering, i.e. only every other event is to be flooded. There may be application domains where the data is produced at a much higher rate than ever needed, even in crises situations. So would it be useful to add a filtering parameter when a variable is set to be flooded?

Operating Modes Mechanism For the operating modes mechanism there is little that can be changed in the data plane. One costly operation for the operating mode mechanism is phase 2, which creates the temporary routing table at the edge status routers. If space is not an issue it might be possible to create all permutation of the temporary routing entry for all the different mode combinations and at the time of each subscription. There are tradeoffs here in that each entry that is added into the routing table becomes more costly, while changing mode becomes faster at the edge status routers.

Another area of future work will be to investigate if it is possible to compress the size of the routing table. Since each mode has its own routing table there might be ways to optimize the datastructure to try and reduce the space that is necessary to provide this mechanism.

Condensation Function Mechanism There are three different areas that will be further explored with respect to the condensation function. The first area involves having the status routers in dedicated hardware and then have part of the hardware dedicated to condensation functions. With dedicated hardware for the condensation function, the latter will not interfere with the critical path of forwarding status events. When a request for a condensation function is sent to the QoS management it can be accompanied with QoS requirements as to how long it should take to perform the condensation function. The QoS management will then take this into consideration when deciding where to place the condensation function, so that it is placed on a status router that can perform the calculation within the specified QoS. This first area will also involve the policy language that was mentioned earlier. Policies determine where the condensation function should be placed. The tradeoff is to place the condensation function close to the source and save bandwidth, while these status routers may become overloaded with condensation function increasing the execution time, so should they be placed away from the sources to improve the execution time by sacrificing bandwidth.

The second area that will be investigated is whether one or more *data centers* should be placed in each cloud and have the condensation function be placed in these data centers. There could also be a hybrid approach where the condensation function could be placed either on a status router or in a data center. The policy in the QoS management would then evaluate where a condensation function should be placed. It can be envisioned that the condensation function that monitors many variables but only performs light computation on them would be placed on a status router close to the source and the computational intensive condensation functions would be placed in the data center.

The third area to be investigated is compound condensation functions. A compound condensation function is a condensation function that needs variables from different clouds. The first possibility is simply to place the condensation function in the cloud with most of the variables in it and forward the remaining variables from the other clouds. For some condensations that can be created using sub-expressions it might be better to create a sub-expression condensation function in each of the clouds that are used as input and then have the result of the sub-expressions be sent to a special combiner condensation function that will publish the final result. The usefulness of such an optimization might be domain specific and limited to that the calculation that the condensation function performs can be decomposed into sub-expressions.

Vulnerabilities and Security in the Data Plane

In addition to the vulnerability that was described for the limited flooding mechanism (see Chapter 5) there are at least two more. The first vulnerability is that it is possible for an adversary to inject false status events. If there are no variables published with the variableId of the false events, then the event is dropped at the first hop. It is possible for the adversary to do a denial of service attack by continuously sending these false events to various (edge) status routers. If the false status event has a variableId that is the same as at least one of the event flows that are routed through the specific (edge) status router, then the false event will be routed according to these event flows.

The second vulnerability is that an adversary can inject false status events with an incorrect `optionLen` field (malformed message). What happens when a (edge) status router receives such an event is that the reading thread will misalign the status events. The consequences of such an event will depend on the data structure of the other events. If all events routed through the (edge) status router do not use option fields, then the reading thread will be aligned itself again after the value set in the `optionLen` number of events has been read in. If some of the events that are routed through the (edge) status router use option fields, then the reading thread may never align itself again. The reason for this is that when the reading thread has become misalign it may treat one of the option fields of a correct status event as the first field of a status event. Then it will depend on what the value is of the 13th byte in this option field if the reading thread will align itself again or not.

10.2.2 Management Plane

The future work for the management plane can be divided into three categories. The first category is to further optimize the implementation of the (leaf) QoS broker. The second category is related to the mechanisms that the (leaf) QoS broker provide and how these can be extended. Finally, the third category is vulnerabilities and security in the current implementation of the management plane and how these can be extended.

Implementation of the Management Plane

The current implementation of the QoS broker only provides for it to be a parent of other leaf QoS brokers and QoS brokers. It does not provide for it to be a child of a parent QoS broker. This means that the current implementation only supports for two layers in the management plane, one QoS broker that has one or more leaf QoS broker children. Some of the “plumbing” code that is needed for the QoS broker to act as a child is provided, but it is not completed.

The current implementation of the management plane provides for the plumbing of using the mechanisms that are present in the data plane. These mechanisms are activated either by the command interface that the management plane provides or by hard coded policies to test these mechanisms. To automate the management plane a policy language must be devised along with a policy driven engine that will use the provided mechaism. When this is provided, the management plane will be policy driven with the option for controllers to manually use the command interface to overwrite actions that the policy engine has initiated.

Mechanisms in the Management Plane

For each of the mechanisms that are present in the management plane a short description of some possible future work is discussed. Some of this work is optimization of the current implementation while others are how the mechanisms can be extended.

Multicast Rate-filtering Routing Mechanism The current function that is used by the management plane to estimate how much resources are consumed by the different subscriptions that are routed through a (edge) status router is fairly coarse. The current estimating function only takes into account the smallest subscription intervals for a variable. This is a rather optimistic estimation due to the deterministic filtering which may forward more events than just the one at the smallest subscription interval. The second deficiency with the estimation function is that it does not include the interarrival interval of the different variables. This means that if all the variables are published at the same time the traffic will be very bursty and hence the resources at the (edge) status routers may be exhausted, since the estimation of resource usage assumed a steady flow of traffic.

Limited Flooding Mechanism There are different tradeoff involved when it comes to take the decision whether a variable is to be flooded. The first is how many subscribers within the region are interested in the variable in the first place. If a large enough number of the subscribers are interested in receiving all the events from a variable so that each of the event channels carries all events anyway, then it might be beneficial to flood this variable within this region. Depending on the urgency of getting the latest state of a variable under certain condition, some of these urgent variables may be set to be flooded within an urgency region. The third option of when a variable is to be flooded may be that under a certain situation the variable is important to many subscribers, but the subscribers will not know before hand which of the variables that this will be. Therefore, it is not possible to set up subscriptions to these variables, since they are not known. The management plane can then set such variables to be flooded and all the subscribers within the region will get them.

The management plane will need a policy that will specify the importance of the tradeoffs specified above of which variables are to be flooded at what time and to what region. This will depend on the application domain and the specific application that the framework is used for. The policy or the decision engine needed for this mechanism to be initiated automatically was not part of this dissertation work.

Operating Modes Mechanism In the current implementation of the management plane it is the latest command that sets the mode that decides which mode the sub-tree operates in, independently if this overwrites the mode set by an ancestor QoS broker. Also it is possible for two neighbor clouds to operate in two different modes, which can interfere with inter-cloud subscriptions. So, how the operating mode mechanism should be used by the management plane, is future work.

Condensation Function Mechanism In Section 10.2.1 improvements to the condensation function mechanism in the data plane were presented. The management plane will need a policy language where the application's needs are specified so that the management plane can take this into consideration when placing a condensation function. A new search algorithm that will find the "optimal" location of a condensation function must also be devised. These algorithms must take into account the tradeoffs that were specified in the policy language and the current usage of the output of the condensation function.

Vulnerabilities and Security in the Management Plane

The focus of this dissertation was to devise a managed status dissemination middleware framework and making it efficient and resilient (with respect to failures.) The future work is then to enhance the framework by making it secure.

In the current implementation an adversary can take complete control of the entire system, if he/she is able to inject false command messages into the management plane. This is because it is the management plane that completely controls the data plane, so if the adversary can control the management plane he/she also controls the data plane.

In order to secure the management plane the commands that are sent must be authenticated. The command messages include the commands that are exchanged within the management plane (the (leaf) QoS brokers), the command messages that are exchanged between the data plane and the management plane ((edge) status routers and leaf QoS brokers), and finally the command messages that are exchanged between the end-entities (publishers, subscribers, condensation creators) and the edge status routers.

APPENDIX

APPENDIX A

INTERFACES

This appendix presents the CORBA idl interfaces for the command interaction and the Java interface classes that are provided to the end-entities. CORBA is used for the command interaction in that each of the entities that can service commands runs a CORBA servants, which the other entities can remotely invoke. The stubs or the interfaces that the framework provides for the end-points to build applications on top of are provided as Java interface classes. In addition for the end-points Java interface classes are also provided for parts of the framework that can be extended with user-defined logic.

In Table A.1 some statistic of the GridStat framework is presented:

Group	Entity	Main Class	Classes	Num. of lines	Num. of code lines
Data plane	common		10	1898	1051
	command		99	9510	6901
	util		5	2458	937
	publisher	Publisher.java	25	4115	1744
	subscriber	Subscriber.java	54	6721	3173
	condensation creator	Condensation.java	3	928	414
	(E)SR common		45	8710	4150
	status router	StatusRouter.java	5	847	445
	edge status router	EdgeStatusRouter.java	10	3227	1786
Management plane	common		19	5321	3059
	command		27	1707	1207
	leaf QoS broker	LeafQoSBroker.java	24	8471	4861
	QoS broker	QoSBroker.java	26	4549	2756
Test applications	publisher	PublisherApplication.java	2	1369	887
	subscriber	SubscriberApplicationSimpleGUIPull.java	12	2756	1710
	flooding subscriber	FloodingSubscriberApplication.java	2	456	268
	condensation creator	CondensationApplication.java	2	954	679
	(E)SR common		1	194	115
	status router	StatusRouterApplication.java	2	198	106
	edge status router	EdgeStatusRouterApplication.java	2	198	108
	(L)QB common		17	1983	1220
	leaf QoS broker	LeafQoSApplicationAdvGUI.java	9	1553	713
	QoS broker	QoSApplicationAdvGUI.java	5	612	305
Test applications GMS GUI	common		16	4503	2202
	subscriber	SubscriberApplicationAdvGUIPull.java	8	1219	808
	leaf QoS broker	LeafQoSApplicationAdvGUIGMS.java	3	1120	792
	QoS broker	QoSApplicationAdvGUIGMS.java	3	423	291
Total			436	76000	42688

Table A.1: GridStat framework statistic.

The command interaction is how the different entities issue commands to each other. In Figures

A.1 and A.2 presents constants and data structures that are used for the command interaction that are issued between the entities in the data plane (also including the leaf QoS brokers). While Figure A.3 and continuing in Figure A.4 presents the different interfaces (methods, commands) that the different entities (data plane) can issue to each other. In figures A.5 and A.6 presents constants and data structures that are used for the command interaction that are issued between the entities in the management plane. While Figure A.7 and A.8 present the different interfaces (methods, commands) that can be called in the management plane.

The initial configuration information that the different entities can read in from a file at startup (bootstrapping) are specified in XML. The Document Type Definition (DTD) for these files are presented in Figure A.9

The Java API that is provided to the application layer for a publisher is illustrated in Figure A.10, A.11, and A.12. If the publisher application is using derived values then the interface to use is presented in Figure A.13. To publish variables that also encapsulate derived values the application layer has to provide the implementation of these derived values. Each of the derived values produces a typed value so the base class for the derived values and derived values that produces a value of type integer is presented in Figure A.14. For derived values that produces vales of types float and boolean are presented in Figure A.15. Finally, for derived values that produces values of user-defined types are presented in Figure A.16.

The Java API that is provided to the application layer for a subscriber is illustrated in Figure A.17, A.18, and A.19. The communication between the application and the framework with respect to the values of the subscribed variables are done through typed holder classes. The base class for these types holder classes are presented in Figure A.21 and A.22. The holders for integer and float datatypes are illustrated in Figure A.23, while the boolean and user-defined datatypes are illustrated in Figure A.24. For subscription to a group of temporal related variables the holder for such a group is illustrated in Figure A.25. If the subscriber wants to access the derived values that may be published with a variable, then a different set of holder classes must be used.

The APIs for holder classes that also provides for derived values are illustrated in Figures A.26, A.27, and A.28. The API for the extrapolation mechanism are illustrated in Figure A.29. The extrapolation mechanism extends the type specific holder interfaces and provides for the functionality to extrapolate a missed values. Two interaction models are provided by the subscriber API, pull (just read the latest value from the holder) and push (framework notifies the application when a new value is received). For the push interaction model the subscriber must implement the `EventNotificationInterface` (Figure A.30). If group subscription is used then the interface `EventForGroupNotificationInterface` is used instead. The subscriber can also register a callback object to be notified about QoS violation, then the interface `QoSViolationInterface` (Figure A.30) has to be implemented.

The Java API that is provided to the application layer for a condensation creator is illustrated in Figure A.31 and A.32. The condensation creator doesn't have any other functionality then to specify a condensation function and request that it be placed into the data plane by the management plane.

The Java API that is provided to the application layer for customizing the condensation function mechanism is presented in Figure A.33, A.34, and A.35. Figure A.33 presents the interface that are used to communicate between different modules and is also used by the framework to forward events to the condensation function. The interface `CondFunInterface` is the base interface provided by the framework and this is the interface that needs to be extended in order to provide a condensation function with a different triggering mechanism then the ones build-in. The base calculator interface is presented in Figure A.35 and this is the interface that has to be exended by the application layer to perform the application logic that the condensation function should provide.

The common Java API that is provided to a control application for the leaf QoS broker and QoS broker is presented in Figure A.36. They both provides the functionality of explicitly control the operating modes and limited flooding mechanisms. The additional Java API that a leaf QoS broker provides is presented in Figure A.37. This interface is used by an application to add and remove (edge) status routers and event channels to the cloud that is controlled by the leaf QoS broker. The additional Java API that a QoS broker provides is presented in Figure A.38. This interface is used by an application to add and remove children (leaf) QoS brokers and remove inter-cloud event channels from the clouds that is controlled by the QoS broker.

```
#ifndef _EDU_WSU_GRIDSTAT_COMMAND_CONSTANTS
#define _EDU_WSU_GRIDSTAT_COMMAND_CONSTANTS

module edu
{
module wsu
{
  module gridstat
  {
    module command
    {
      ///////////////////////////////////////////////////////////////
      // An interface class that only contain constants to be used
      // for the entities to communicate with each other
      interface CommConstants
      {
        // Priority level
        const short PRIORITY_ALERT = 1;
        const short PRIORITY_HIGH = 2;
        const short PRIORITY_MEDIUM_HIGH = 3;
        const short PRIORITY_MEDIUM = 4;
        const short PRIORITY_MEDIUM_LOW = 5;
        const short PRIORITY_LOW = 6;

        // Message Types
        const short STATIC_TYPE = 0;
        const short DYNAMIC_TYPE = 1;
        const short COMPOUND_TYPE = 2;

        // Network status
        const short TOCONNECT = 1;
        const short CONNECTED = 2;
        const short DISCONNECTED = 3;
        const short ERROR_CONNECTING = 4;
        const short COMMUNICATION_FAILED = 5;
        const short COMMUNICATION_FAILED_IN_CONNECT = 6;

        // The flag that is sent to a SR from its LeafQoSBroker
        // when told to reconnect up to the LeafQoSManager
        const short DO_NOTHING = 0;
        const short DO_FIRST_TIME = 1;

        // Status types
        const short INT_TYPE = 0;
        const short INT_TYPE_SIZE = 4;
        const short BOOLEAN_TYPE = 1;
        const short BOOLEAN_TYPE_SIZE = 1;
        const short FLOAT_TYPE = 2;
        const short FLOAT_TYPE_SIZE = 4;
        const short USER_DEFINED_TYPE = 3;
        const short UNKNOWN_TYPE = 9;

        // Modes
        const unsigned long UNKNOWN_MODE = 0;
        const unsigned long INITIAL_MODE = 1;
        const unsigned long MAX_MODES = 16;

        const unsigned long MODE_MASK_0 = 0x0001;
        const unsigned long MODE_MASK_1 = 0x0002;
        const unsigned long MODE_MASK_2 = 0x0004;
        const unsigned long MODE_MASK_3 = 0x0008;
        const unsigned long MODE_MASK_4 = 0x0010;
        const unsigned long MODE_MASK_5 = 0x0020;
        const unsigned long MODE_MASK_6 = 0x0040;
        const unsigned long MODE_MASK_7 = 0x0080;
        const unsigned long MODE_MASK_8 = 0x0100;
        const unsigned long MODE_MASK_9 = 0x0200;
        const unsigned long MODE_MASK_10 = 0x0400;
        const unsigned long MODE_MASK_11 = 0x0800;
        const unsigned long MODE_MASK_12 = 0x1000;
        const unsigned long MODE_MASK_13 = 0x2000;
        const unsigned long MODE_MASK_14 = 0x4000;
        const unsigned long MODE_MASK_15 = 0x8000;

        // Return codes of a successful command
        const short COMMAND_OK = 0;
        const short COMMAND_OK_LINK_FAILURE = 1;
        const short COMMAND_OK_LINK_FAILURE_NEW_CONNECTION = 2;
        const short COMMAND_OK_PUBSUB_RESTART_FAILURE = 3;
        const short COMMAND_OK_ESR_RESTART_FAILURE = 4;

        // Error return codes, Note these HAVE to be NEGATIVE!
        const short ERROR_CHANNEL_FAILURE_TO_SR = -1;
        const short ERROR_CHANNEL_OPEN = -2;
        const short ERROR_CHANNEL_FAILURE_TO_PUBSUB = -3;
        const short ERROR_CHANNEL_OPEN_REMOTE = -4;
        const short ERROR_CHANNEL_SENDING_THREAD = -5;
        const short ERROR_COMM_INITIALIZATION = -6;
        const short ERROR_COMM_FAILURE_TO_SR = -7;
        const short ERROR_COMM_FAILURE_TO_LEAF = -8;
        const short ERROR_COMM_FAILURE_TO_BROKER = -9;
        const short ERROR_COMM_FAILURE_TO_SUB = -10;
        const short ERROR_SERVER_COULD_NOT_START = -11;
        const short ERROR_SERVER_DISCONNECT = -12;
        const short ERROR_SERVER_SHUTDOWN = -13;
        const short ERROR_TREAD_START_PROBLEM = -14;

        // Security violations
        const short ERROR_SECURITY_VIOLATION = -15;
        const short ERROR_WRONG_CLOUD = -16;
        const short ERROR_SR_NOT_REGISTERED = -17;
        const short ERROR_COMM_LINK_NOT_REGISTERED = -18;

        // Errors for publishers
        const short ERROR_PUB_ALREADY_REGISTERED = -20;
        const short ERROR_PUB_NOT_REGISTERED = -21;
        const short ERROR_PUB_VARIABLE_ALREADY_REGISTERED = -22;
        const short ERROR_PUB_VARIABLE_NOT_REGISTERED = -23;
        const short ERROR_PUB_QOS_INTERVAL = -24;
        const short ERROR_PUB_STILL_PUBLICATIONS = -25;
        const short ERROR_PUB_REREGISTER_PUBLICATIONS = -26;

        // Errors for subscribers
        const short ERROR_SUB_ALREADY_REGISTERED = -30;
        const short ERROR_SUB_NOT_REGISTERED = -31;
        const short ERROR_SUB_SUBSCRIPTION_ALREADY_REGISTERED = -32;
        const short ERROR_SUB_SUBSCRIPTION_NOT_REGISTERED = -33;
        const short ERROR_SUB_VARIABLE_NOT_EXIST = -34;
        const short ERROR_SUB_LATENCY_NOTSATISFIABLE = -35;
        const short ERROR_SUB_REDUNDANCY_NOTSATISFIABLE = -36;
        const short ERROR_SUB_INTERVAL_NOTSATISFIABLE = -37;
        const short ERROR_SUB_PUBLISHER_ENDED = -38;
        const short ERROR_SUB_NO_PATH = -39;
        const short ERROR_SUB_WRONG_DATATYPE = -40;
        const short ERROR_SUB_NO_PATH_ALLOCATED = -41;
        const short ERROR_SUB_NO_PUBLISHER_CLOUD = -42;
        const short ERROR_SUB_NO_PARENT_AVAIL = -43;
        const short ERROR_SUB_WRONG_PUB_NAME = -44;
        const short ERROR_SUB_WRONG_PUB_NAME2 = -45;
        const short ERROR_SUB_WRONG_SUB_NAME = -46;
        const short ERROR_SUB_WRONG_SUB_NAME2 = -47;
        const short ERROR_SUB_NO_EDGE_OUT_OF_START = -48;
        const short ERROR_SUB_NO_EDGE_OUT_OF_END = -49;
        const short ERROR_SUB_WRONG_HIERARCH_NAME = -50;
        const short ERROR_SUB_NOT_IMPLEMENTED_YET = -51;
        const short ERROR_SUB_STILL_PUBLICATIONS = -52;
        const short ERROR_SUB_UNKNOWN_GROUP = -53;
        const short ERROR_SUB_ALREADY_IN_GROUP = -54;
        const short ERROR_SUB_QOS_ARGUMENTS = -55;

        // Error for condensation creators
        const short ERROR_COND_ALREADY_REGISTERED = -60;
        const short ERROR_COND_NOT_REGISTERED = -61;
        const short ERROR_COND_CONDFUN_ALREADY_REGISTERED = -62;
        const short ERROR_COND_CONDFUN_NOT_REGISTERED = -63;
        const short ERROR_COND_NO_PLACEMENT = -64;
        const short ERROR_COND_CREATING = -65;
        const short ERROR_COND_DESTROING = -66;

        // Some common errors
        const short ERROR_INTERRUPTED_EXCEPTION = -70;
        const short ERROR_EXCEPTION = -71;
        const short ERROR_UNKNOWNHOST_EXCEPTION = -72;
        const short ERROR_ASSERT = -73;
        const short ERROR_SUBSCRIPTION_PATH_NOT_PRESENT = -74;
        const short ERROR_COMMAND_FAILED = -75;
      };
    };
  };
};
};

#endif // define _EDU_WSU_GRIDSTAT_COMMAND_CONSTANTS
```

Figure A.1: Constants for the command interaction between the data plane entities.

```
#ifndef _EDU_WSU_GRIDSTAT_COMMAND_STRUCTS
#define _EDU_WSU_GRIDSTAT_COMMAND_STRUCTS

#include "commConstants.idl"
#include "hierConstants.idl"

module edu
{
module wsu
{
  module gridstat
  {
    module command
    {
      // Sequence of base types
      typedef sequence<unsigned long,
                       CommConstants::MAX_MODES> IntervalSeq;
      typedef sequence<unsigned short,
                       CommConstants::MAX_MODES> PrioritySeq;

      // Holder for the interval
      struct IntervalInfo
      {
        unsigned long intervalLow;
        unsigned long intervalHigh;
      };

      // Holder for QoS requirement
      struct QoSInfo
      {
        // These are the desired QoS properties desired
        unsigned short desiredSecurity;
        unsigned short desiredRedundancy;
        unsigned short desiredPriority;
        unsigned short desiredTimeliness;

        // Subscription Interval properties
        IntervalHolder interval;

        // These are the worst case QoS properties
        unsigned short minimumSecurity;
        unsigned short minimumRedundancy;
        unsigned short minimumPriority;
        unsigned short minimumTimeliness;
      };

      // Sequence to hold QoS requirements for each mode
      typedef sequence<QoSInfo, CommConstants::MAX_MODES> QoSSeq;

      // Holder for information that a publisher provides when
      // publishing a status variable
      // NOTE: Total message length = messageLength+patternLength
      struct PublicationInfo
      {
        string pubName;    // The name of the publisher
        string variableName;
        unsigned short type; // Static, dynamic, or compound
        unsigned short dataType; // Int, Float, Boolean, User defined
        unsigned long userDataType; // If user defined the data type
        unsigned long messageLength; // How many bytes
        unsigned long patternLength; // How many bytes
        unsigned short numPattern; // How many patterns
        unsigned short priority; // Alert=0, 1-5 High-Low priority
        IntervalInfo interval;
      };

      // Holder information an identifier for a single subscription
      struct SubscriptionIdInfo
      {
        string srcCloud;
        string pubName;
        string variableName;
        string dstCloud;
        string subName;
        unsigned short redundantId;
      };

      // Holder information for a single subscription
      struct SubscriptionInfo
      {
        unsigned long variableId;
        string dstSRName;
        unsigned long pathId;
        unsigned short redundantId;
        IntervalSeq subInterval;
        unsigned short dataType; // Int, Float, Boolean, User defined
        unsigned long userDataType; // If user defined the data type
        unsigned long messageLength; // How many bytes
        PrioritySeq priority;
        unsigned long modes;
      };

      // Holder for information that a subscriber provides when
      // subscribing to a status variable
      struct SubscribeInfo
      {
        string subName;
        string pubName;
        string variableName;
        unsigned long modes;
        unsigned short dataType; // Int, Float, Boolean, User defined
        unsigned long userDataType; // If user defined the data type
        QoSHolder QoS;
      };

      // Holder information for the id of a event channel
      struct EventChannelIdInfo
      {
        string srcSRName;
        string dstSRName;
      };

      // Holder for information for a event channel between two SRs
      struct EventChannelInfo
      {
        EventChannelIdInfo linkId;
        string srcAdr;
        unsigned long srcPort;
        string dstAdr;
        unsigned long dstPort;
        unsigned long level;
        unsigned short bandwidth;
        unsigned long latency;
      };

      // Sequence to hold a number of subscriptions for a cond. fun
      typedef sequence<SubscribeInfo,
                       CommConstants::MAX_SUB_COND> SubscribeSeq;
      typedef sequence<SubscriptionInfo,
                       CommConstants::MAX_SUB_COND> SubscriptionSeq;

      // Holder for information for a condensation function
      struct CondensationRequestInfo
      {
        string creator;
        string placementEdge; // Request cond fun be placed here
        PublishHolder pubHolder;
        boolean inFilterLow;
        float inFilterLowValue;
        boolean inFilterHigh;
        float inFilterHighValue;
        boolean outFilterLow;
        float outFilterLowValue;
        boolean outFilterHigh;
        float outFilterHighValue;
        string calculatorURI;
        string calculatorClassName;
        unsigned short triggerType;
        unsigned long triggerVar1;
        unsigned long triggerVar2;
        SubscribeSeq subSeq;
      };

      // Holder for information for a condensation function
      struct CondensationSetupInfo
      {
        string creator;
        PublishHolder pubHolder;
        boolean inFilterLow;
        float inFilterLowValue;
        boolean inFilterHigh;
        float inFilterHighValue;
        boolean outFilterLow;
        float outFilterLowValue;
        boolean outFilterHigh;
        float outFilterHighValue;
        string calculatorURI;
        string calculatorClassName;
        unsigned short triggerType;
        unsigned long triggerVar1;
        unsigned long triggerVar2;
        SubscriptionSeq subSeq;
      };
    };
  };
};
};

#endif // define _EDU_WSU_GRIDSTAT_COMMAND_STRUCTS
```

Figure A.2: Data structures that are used for the command interaction between the data plane entities.

```
#ifndef _EDU_WSU_GRIDSTAT_COMMAND_INTERFACES
#define _EDU_WSU_GRIDSTAT_COMMAND_INTERFACES

#include "commStructs.idl"

module edu
{
module wsu
{
  module gridstat
  {
    module command
    {
      ///////////////////////////////////////////////////////////////
      // Commands that can be issued by the leaf QoS broker to edge
      // status routers and status routers
      interface CommandLeafToSRCommon
      {
        // Method to connection a event channel
        short connectLink(in EventChannelInfo eventChannel);

        // Method to disconnect a event channel
        short disconnectLink(in string dstSRName);

        // Method to establish a path in the routing table
        short setupPath(in unsigned long variableId,
                        in string dstSRName,
                        in unsigned long pathId,
                        in unsigned short redundantId,
                        in IntervalSeq subInterval,
                        in unsigned long pubInterval,
                        in PrioritySeq priority,
                        in unsigned long modes);

        // Method to terminate a paths in the routing table
        short terminatePath(in unsigned long variableId,
                            in string dstSRName,
                            in unsigned long pathId,
                            in unsigned short redundantId,
                            in IntervalSeq subInterval,
                            in unsigned long modes);

        // Method to change the priority of an established path
        short changePathPriority(in unsigned long variableId,
                                 in string dstSRName,
                                 in unsigned long pathId,
                                 in unsigned short redundantId,
                                 in unsigned long subInterval,
                                 in unsigned long modes,
                                 in unsigned short newPriority);

        // Method to change the interval of an established path
        short changePathSubInterval(in unsigned long variableId,
                                    in string dstSRName,
                                    in unsigned long pathId,
                                    in unsigned short redundantId,
                                    in unsigned long subInterval,
                                    in unsigned long modes,
                                    in unsigned long newSubInterval);

        // Method to change the pub interval of a variable that has
        // at least one path through this SR
        short changeVarPubInterval(in unsigned long variableId,
                                   in unsigned long newPubInterval);

        // Method to create a condensation function
        short setupCondensation(
                               in CondensationSetupInfo condHolder,
                               in unsigned long variableId);

        // Method to remove an condensation function
        short terminateCondensation(in unsigned long variableId);

        // Method to set one of the variables to be flooded
        short setupFlooding(in unsigned long variableId,
                            in unsigned long pubInterval,
                            in unsigned long level,
                            in unsigned long modes);

        // Method to terminate the flooding of one of the variables
        short terminateFlooding(in unsigned long variableId,
                                in unsigned long modes);
      };
```

```
///////////////////////////////////////////////////////////////
// Commands that can be issued by the leaf QoS broker
// to the status routers
interface CommandLeafToSR : CommandLeafToSRCommon
{
  // Method to set the new mode
  short changeMode(in unsigned long mode);
};

///////////////////////////////////////////////////////////////
// Commands that can be issued by the leaf QoS broker to the
// edge status routers
interface CommandLeafToESR : CommandLeafToSRCommon
{
  // Method to inform about a mode change
  short informChangeMode(in unsigned long mode);

  // Method to prepare for a mode change
  short prepareChangeMode(in unsigned long mode);

  // Method to set the new mode
  short changeMode(in unsigned long mode);

  // Method to commit a mode change
  short commitChangeMode(in unsigned long mode);

  // Method to forward an error msg that a subscription failed
  short subscribeErrorMsg(in string subName,
                          in string pubName,
                          in string variableName,
                          in unsigned short errorCode);
};

///////////////////////////////////////////////////////////////
// Commands that can be issued by publishers to its
// leaf QoS broker
interface CommandPubToESR
{
  // Method to register publishing of a variable
  short publish(in PublishInfo pubHolder,
                inout unsigned long variableId);

  // Method to unregister a publishing of a variable
  short unPublish(in string pubName, in string variableName);

  // Method to register a publisher
  short registerPublisher(in string pubName,
                          inout string hostName,
                          inout unsigned long hostPort,
                          in boolean firstTime);

  // Method to unregister a publisher
  short unRegisterPublisher(in string pubName);

  // Used to check if the edge status router is running
  boolean pubPing();
};

///////////////////////////////////////////////////////////////
// Commands that can be issued by subscribers to its
// leaf QoS broker
interface CommandSubToESR
{
  // Method to register a subscription
  short subscribe(in SubscribeInfo subHolder,
                  out unsigned long variableId,
                  out unsigned short numPatterns);

  // Method to unregister a subscription
  short unSubscribe(in string pubName,
                    in string variableName,
                    in string subName);

  // Method to register a subscriber
  short registerSubscriber(in string subName,
                           inout string hostName,
                           inout unsigned long hostPort,
                           out unsigned long currentMode,
                           in boolean firstTime);

  // Method to unregister a subscriber
  short unRegisterSubscriber(in string subName);

  // Used to check if the edge status router is running
  boolean subPing();
};
```

Figure A.3: Interfaces for command interaction between the data plane entities, part 1 of 2.

```
      ///////////////////////////////////////////////////////////////
      // Commands that can be issued by status routers to its
      // leaf QoS broker
      interface CommandSRToLeaf
      {
        // Method to notify the parent that it is online
        unsigned short changeSRStatus(in string SRName,
                                      in unsigned short status,
                                      out unsigned long mode);

        // Method to inform about a state change in a comm. link
        short changeLinkStatus(in string srcSRName,
                               in string dstSRName,
                               in unsigned short status);

        // Method to inform about change in the state for the
        // buffering for on of the comm. links
        short changeLinkBufferingState(in string srcSRName,
                                       in string dstSRName,
                                       in boolean onOff);

        // Used to check if the leaf QoS broker is running
        boolean ping();

        //                                     //
        // Needed to relay from the publishers //
        //                                     //
        // Method to register publishing of a variable
        short publish(in PublishInfo pubHolder,
                      inout unsigned long variableId);

        // Method to unregister a publishing of a variable
        short unPublish(in string pubName, in string variableName);

        // Method to unregister all publications from a publisher
        short unPublishAll(in string pubName);

        // Method to register a publisher
        short registerPublisher(in string pubName,
                                in string edgeStatusRouter);

        // Method to unregister a publisher
        short unRegisterPublisher(in string pubName);

        //                                      //
        // Needed to relay from the subscribers //
        //                                      //
        // Method to register a subscription
        short subscribe(in SubscribeInfo subHolder,
                        out unsigned long variableId,
                        out unsigned short numPatterns);

        // Method to unregister a subscription
        short unSubscribe(in string pubName,
                          in string variableName,
                          in string subName);

        // Method to unregister all subscriptions from a subscriber
        short unSubscribeAll(in string subName);

        // Method to register a subscriber
        short registerSubscriber(in string subName,
                                 in string edgeStatusRouter);

        // Method to unregister a subscriber
        short unRegisterSubscriber(in string subName);

        //                                               //
        // Needed to relay for the condensation creators //
        //                                               //
        // Method to create a condensation function
        short condensationCreate(
                            in CondensationRequestInfo condHolder);

        // Method to remove a condensation function
        short condensationRemove(in string creator,
                                 in string pubName,
                                 in string variableName);

        // Method to register a condensaton creator
        short registerCondensationCreator(in string condName,
                                          in string edgeStatusRouter);

        // Method to unregister a condensaton creator
        short unRegisterCondensationCreator(in string subName);
      };
```

```
      ///////////////////////////////////////////////////////////////
      // Commands that can be issued by condensation creators to its
      // leaf QoS broker
      interface CommandCondToESR
      {
        // Method to create a condensation function
        short condensationCreate(
                           in CondensationRequestInfo condHolder);

        // Method to remove a condensation function
        short condensationRemove(in string creator,
                                 in string pubName,
                                 in string variableName);

        // Method to register a condensaton creator
        short registerCondensationCreator(in string condName);

        // Method to unregister a condensaton creator
        short unRegisterCondensationCreator(in string subName);
      };

      ///////////////////////////////////////////////////////////////
      // Commands that can be issued by leaf QoS brokers,
      // publishers, subscribers, and condensation creators to
      // edge status routers
      interface CommandLeafPubSubCondToESR : CommandLeafToESR,
                                             CommandPubToESR,
                                             CommandSubToESR,
                                             CommandCondToESR
      {
        // No extra methods needed yet
      };

      ///////////////////////////////////////////////////////////////
      // Commands that can be issued by edge status router to a
      // subscriber to inform the subscriber about different issues
      // that might have arisen
      interface CommandESRToSub
      {
        // Method to inform about preparation for a mode change
        short newModePreparation(in unsigned long mode);

        // Method to inform about the change to a new mode
        short newMode(in unsigned long mode);

        // Method to forward a error msg about a subscription
        short subscribeErrorMsg(in string pubName,
                                in string variableName,
                                in unsigned short errorCode);
      };
    };
  };
};
};

#endif // define _EDU_WSU_GRIDSTAT_COMMAND_INTERFACES
```

Figure A.4: Interfaces for command interaction between the data plane entities, part 2 of 2.

```
#ifndef _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_CONSTANTS
#define _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_CONSTANTS

module edu
{
  module wsu
  {
    module gridstat
    {
      module qosBroker
      {
        module command
        {
          ////////////////////////////////////////////////////////////
          // An interface class that only contain constants to be
          // used by the QoS hierarchy
          interface HierConstants
          {
            // Different entities types
            const short EDGE_SR_TYPE = 0;
            const short SR_TYPE = 1;
            const short FOREGN_SR_TYPE = 2;
            const short BOARDER_SR_TYPE = 3;
            const short LEAF_QOS_BROKER_TYPE = 4;
            const short QOS_BROKER_TYPE = 5;
            const short QOS_BROKER_FOREIGN_TYPE = 6;

            // Different communication links types
            const short CLOUD_EDGE = 0;
            const short INTER_CLOUD_EDGE = 1;
            const short INTER_CLOUD_EDGE_PARTIAL = 2;

            // Constants for inter cloud subscriptions
            const short CLOUD_SUB = 0;
            const short INTER_SUB = 1;
            const short INTER_SUB_START = 2;
            const short INTER_SUB_MID = 3;
            const short INTER_SUB_END = 4;

            // The state of an inter cloud subscription
            const boolean TO_COMMITT = FALSE;
            const boolean COMMITTED = TRUE;

            // How many redundant paths
            const long MAX_REDUNDANT_PATHS = 10;
          };
        };
      };
    };
  };
};

#endif // define _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_CONSTANTS
```

Figure A.5: Constants for the communication between the management plane entities.

```
#ifndef _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_STRUCTS
#define _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_STRUCTS

#include "commStructs.idl"

module edu
{
  module wsu
  {
    module gridstat
    {
      module qosBroker
      {
        module command
        {
          // Holder for an inter-cloud subscription that originate in
          // this cloud
          struct SubStartInfo
          {
            string srcCloud;
            string pubName;
            string variableName;
            string dstCloud;
            string dstVertex;
            short dataType;
            unsigned long modes;
            edu::wsu::gridstat::command::QoSSeq subQoS;
          };

          // Holder for an inter-cloud subscription that doesn't have
          // the originate or destination in its cloud
          // this cloud
          struct SubMidInfo
          {
            string srcCloud;
            string srcVertex;
            string dstCloud;
            string dstVertex;
            unsigned long variableId;
            unsigned long pubInterval;
            unsigned long modes;
            edu::wsu::gridstat::command::QoSSeq subQoS;
          };

          // Holder for an inter-cloud subscription that has its
          // destination in this cloud
          struct SubEndInfo
          {
            string srcCloud;
            string srcVertex;
            string dstCloud;
            string subscriberName;
            unsigned long variableId;
            unsigned long pubInterval;
            unsigned long modes;
            edu::wsu::gridstat::command::QoSSeq subQoS;
          };

          // List of EventChannelId
          typedef sequence<edu::wsu::gridstat::command::EventChannelIdInfo, HierConstants::MAX_REDUNDANT_PATHS> SeqEventChannel;

          // Holder for returning an inter-cloud subscription
          struct ReturnPathInfo
          {
            edu::wsu::gridstat::command::SubscriptionIdInfo pathId;
            SeqEventChannel eventChannels;
          };

          // List of ReturnPath
          typedef sequence<ReturnPathInfo, HierConstants::MAX_REDUNDANT_PATHS> SeqReturnPaths;
        };
      };
    };
  };
};

#endif // define _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_STRUCTS
```

Figure A.6: Data structures that are used for the command interaction between the management plane entities.

```
#ifndef _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_INTERFACES
#define _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_INTERFACES

#include "commStructs.idl"
#include "hierStructs.idl"

module edu
{
  module wsu
  {
    module gridstat
    {
      module qosBroker
      {
        module command
        {
          //////////////////////////////////////////////////////////////////////////////////////////////////////////
          // Commands that can be issued by a QoS broker to one of its children (leaf) QoS brokers
          interface HierarchyChildCommon
          {
            // Method to commit a inter-cloud subscription
            short commitSub(in edu::wsu::gridstat::command::SubscriptionIdInfo subId);

            // Method to remove a inter-cloud subscription
            short removeSub(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in short code);

            // Method to set a variable to be flooded to a given levels and modes
            short setFlooding(in string pubName, in string variableName, in unsigned long variableId, in unsigned long pubInterval,
                              in unsigned long level, in unsigned long modes);

            // Method to remove a flooding from the given modes
            short removeFlooding(in string pubName, in string variableName, in unsigned long modes);

            // Method to change the operating mode
            short changeMode(in unsigned long mode);

            // Method to query what the current operating mode is
            short getMode(out unsigned long mode);
          };
        };

        module leafQoSBroker
        {
          //////////////////////////////////////////////////////////////////////////////////////////////////////////
          // Commands that can be issued by a QoS broker to one of its child leaf QoS broker
          interface HierarchyLeafQoSChild : command::HierarchyChildCommon
          {
            // Method to request a start segment of a inter-cloud subscription
            long askSubStart(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in command::SubStartInfo subStart,
                             out unsigned long variableId, out unsigned long pubInterval);

            // Method to request a mid segment of a inter-cloud subscription
            long askSubMid(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in command::SubMidInfo subMid);

            // Method to request a end segment of a inter-cloud subscription
            long askSubEnd(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in command::SubEndInfo subEnd);

            // Method to query what the variableId of a publication is
            short getVariableId(in string pubName, in string variableName, out unsigned long variableId, out unsigned long pubInterval);
          };
        };
```

Figure A.7: Interfaces for command interaction between the management plane entities, part 1 of 2.

```
        module qosBroker
        {
          ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////
          // Commands that can be issued by a (leaf) QoS broker to its QoS broker parent
          interface HierarchyQoSParent
          {
            // Method to add a inter-cloud comm. link
            boolean newInterEdge(in edu::wsu::gridstat::command::EventChannelInfo edge);

            // Method to change the status of a inter-cloud comm. link
            boolean statusInterEdge(in edu::wsu::gridstat::command::EventChannelInfo edgeId, in unsigned short status);

            // Method to remove a inter-cloud comm. link
            boolean removeInterEdge(in edu::wsu::gridstat::command::EventChannelInfo edgeId);

            // Method to initialize one of the children
            boolean childInitialize(in string childName, in unsigned short status, in long coordinateX, in long coordinateY);

            // Method call by a child to tell if it comes online or goes offline
            boolean changeChildStatus(in string childName, in unsigned short status);

            // Method to request an inter-cloud subscription
            short subscribe(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in short dataType, in unsigned long modes,
                            in edu::wsu::gridstat::command::QoSSeq qoSRequirment, out unsigned long variableId);

            // Method to remove a inter-cloud subscription
            short unSubscribe(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in short code);
          };

          ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////
          // Commands that can be issued by a QoS broker to one of its QoS broker children
          interface HierarchyQoSChild : command::HierarchyChildCommon
          {
            // Method to request a start segment of a inter-cloud subscription
            command::SeqReturnPaths askSubStart(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in command::SubStartInfo subStart,
                                                in command::SeqEventChannel interEdges, in unsigned short redundancy);

            // Method to request a mid segment of a inter-cloud subscription
            long askSubMid(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in command::SubMidInfo subMid);

            // Method to request a end segment of a inter-cloud subscription
            long askSubEnd(in edu::wsu::gridstat::command::SubscriptionIdInfo subId, in command::SubEndInfo subEnd);
          };
        };
      };
    };
  };
};
#endif // define _EDU_WSU_GRIDSTAT_QOSBROKER_COMMAND_INTERFACES
```

Figure A.8: Interfaces for command interaction between the management plane entities, part 2 of 2.

```
<?xml version="1.0" encoding='us-ascii'?>
<!ELEMENT xml_Gridstat ANY>

<!-- Specifies the location of the namingservice. -->
<!ELEMENT namingService EMPTY>
<!ATTLIST namingService
      host        CDATA       #REQUIRED
      port        CDATA       #REQUIRED
>

<!-- Information about the (Leaf) QoS Broker itslef (who I am I). -->
<!ELEMENT myInfo EMPTY>
<!ATTLIST myInfo
      myname      CDATA       #REQUIRED
      parentname  CDATA       #REQUIRED
      coordinateX CDATA       "100"
      coordinateY CDATA       "100"
>

<!-- Information about (Edge) Status Routers in a cloud. -->
<!ELEMENT vertexTbl (vertex*)>
<!ELEMENT vertex EMPTY>
<!ATTLIST vertex
      name        CDATA       #REQUIRED
      type        (0 | 1 | 2 | 3) "0"
      coordinateX CDATA       "100"
      coordinateY CDATA       "100"
      msgPerSec   CDATA       #REQUIRED
>

<!-- Information about the event channels in a cloud. -->
<!ELEMENT edgeTbl (edge*)>
<!ELEMENT edge EMPTY>
<!ATTLIST edge
      srcName     CDATA       #REQUIRED
      srcAdr      CDATA       #REQUIRED
      srcPort     CDATA       #REQUIRED
      dstName     CDATA       #REQUIRED
      dstAdr      CDATA       #REQUIRED
      dstPort     CDATA       #REQUIRED
      type        (0 | 1 | 2) "0"
      bandwidth   CDATA       #REQUIRED
      latency     CDATA       #REQUIRED
>

<!-- Information about this QoS Brokers children. -->
<!ELEMENT childNodeTbl (child*)>
<!ELEMENT child EMPTY>
<!ATTLIST child
      name        CDATA       #REQUIRED
      type        (4 | 5)     "4"
>
```

Figure A.9: DTD for the initialization files for the management plane entities.

```
package edu.wsu.gridstat.publisher.interfaces;

/**
 * <p>Title: PublisherInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// GridStat packages.
import edu.wsu.gridstat.publisher.common.PublisherException;

public interface PublisherInterface
{
  /**
  * The <code>connectToEdge</code> method is used to connect to a edgeStatusRouter.
  * <BR>
  * @return Returns <code>0</code> if connected, error code otherwise.
  */
  public short connectToEdge();

  /**
  * The <code>disConnectFromEdge</code> method is used to disconnect from a edgeStatusRouter.
  * <BR>
  * @return Returns <code>0</code> we have unregisteded all the publications
  * and we have disconnected, error code otherwise.
  */
  public short disConnectFromEdge();

  /**
  * The <code>reConnectToEdge</code> method is used to connect to a edgeStatusRouter.
  * <BR>
  * @return Returns <code>0</code> if connected, error code otherwise.
  */
  public short reConnectToEdge();

  /**
  * The <code>isConnected</code> method is used return if we are connected to the edgeStatusRouter or not.
  * <BR>
  * @return Returns <code>true</code> if we are connected, <code>false</code> otherwise.
  */
  public boolean isConnected();

  /**
  * The <code>publish</code> method is used to publish a userdefined event.
  * <BR>
  * @param variableId The unique Id of this status variable
  * @param data The data to be published.
  * @return Returns <code>true</code> if message is published, <code>false</code> otherwise.
  */
  public boolean publish(int variableId, byte[] data);

  /**
  * The <code>publish</code> method is used to publish a userdefined event.
  * <BR>
  * @param variableId The unique Id of this status variable
  * @param data The data to be published.
  * @param timeStamp The timestamp that this event will get.
  * @return Returns <code>true</code> if message is published, <code>false</code> otherwise.
  */
  public boolean publish(int variableId, byte[] data, long timeStamp);

  /**
  * The <code>publish</code> method is used to publish a int event.
  * <BR>
  * @param variableId The unique Id of this status variable
  * @param value The value to be published.
  * @return Returns <code>true</code> if message is published, <code>false</code> otherwise.
  */
  public boolean publish(int variableId, int value);

  /**
  * The <code>publish</code> method is used to publish a int event.
  * <BR>
  * @param variableId The unique Id of this status variable
  * @param value The value to be published.
  * @param timeStamp The timestamp that this event will get.
  * @return Returns <code>true</code> if message is published, <code>false</code> otherwise.
  */
  public boolean publish(int variableId, int value, long timeStamp);

  /**
  * The <code>publish</code> method is used to publish a float event.
  * <BR>
  * @param variableId The unique Id of this status variable
  * @param value The value to be published.
  * @return Returns <code>true</code> if message is published, <code>false</code> otherwise.
  */
  public boolean publish(int variableId, float value);
```

Figure A.10: The publisher Java API provided by the framework, part 1 of 3.

```
/**
* The <code>publish</code> method is used to publish a float event.
* <BR>
* @param variableId The unique Id of this status variable
* @param value The value to be published.
* @param timeStamp The timestamp that this event will get.
* @return Returns <code>true</code> if message is published, <code>false</code> otherwise.
*/
public boolean publish(int variableId, float value, long timeStamp);

/**
* The <code>publish</code> method is used to publish a boolean event.
* <BR>
* @param variableId The unique Id of this status variable
* @param value The value to be published.
* @return Returns <code>true</code> if message is published, <code>false</code> otherwise.
*/
public boolean publish(int variableId, boolean value);

/**
* The <code>publish</code> method is used to publish a boolean event.
* <BR>
* @param variableId The unique Id of this status variable
* @param value The value to be published.
* @param timeStamp The timestamp that this event will get.
* @return Returns <code>true</code> if message is published, <code>false</code> otherwise.
*/
public boolean publish(int variableId, boolean value, long timeStamp);

/**
* The <code>registerPublishInt</code> method is to register a publish with the edgeStatusRouter.
* <BR>
* @param variableName The name of the variable which evente is to be published.
* @param priority The priority of this status variable.
* @param intervalLow The lowest interval that the message will be published.
* @param intervalHigh The highest interval that the message will be published.
* @return Returns the variableId of the registered publication.
* @throws PublisherException If something went wrong during the registration.
*/
public int registerPublishInt(String variableName, short priority, int intervalLow, int intervalHigh) throws PublisherException;

/**
* The <code>registerPublishFloat</code> method is to register a publish with the edgeStatusRouter.
* <BR>
* @param variableName The name of the variable which evente is to be published.
* @param priority The priority of this status variable.
* @param intervalLow The lowest interval that the message will be published.
* @param intervalHigh The highest interval that the message will be published.
* @return Returns the variableId of the registered publication.
* @throws PublisherException If something went wrong during the registration.
*/
public int registerPublishFloat(String variableName, short priority, int intervalLow, int intervalHigh) throws PublisherException;

/**
* The <code>registerPublishBoolean</code> method is to register a publish with the edgeStatusRouter.
* <BR>
* @param variableName The name of the variable which evente is to be published.
* @param priority The priority of this status variable.
* @param intervalLow The lowest interval that the message will be published.
* @param intervalHigh The highest interval that the message will be published.
* @return Returns the variableId of the registered publication.
* @throws PublisherException If something went wrong during the registration.
*/
public int registerPublishBoolean(String variableName, short priority, int intervalLow, int intervalHigh) throws PublisherException;

/**
* The <code>registerPublishUserDefined</code> method is to register a publish with the edgeStatusRouter.
* <BR>
* @param variableName The name of the variable which evente is to be published.
* @param priority The priority of this status variable.
* @param intervalLow The lowest interval that the message will be published.
* @param intervalHigh The highest interval that the message will be published.
* @param userType The type of the user registered type.
* @param length The length iin bytes of the user defined type.
* @return Returns the variableId of the registered publication.
* @throws PublisherException If something went wrong during the registration.
*/
public int registerPublishUserDefined(String variableName, short priority, int intervalLow,
                                      int intervalHigh, int userType, int length) throws PublisherException;

/**
* The <code>registerPublishIntAlert</code> method is to register a publish with the edgeStatusRouter.
* <BR>
* @param variableName The name of the variable which evente is to be published.
* @param minInterval The minimum interval between two alerts being published. This will
* prevent the publisher from flooding the network.
* @return Returns the variableId of the registered publication.
* @throws PublisherException If something went wrong during the registration.
*/
public int registerPublishIntAlert(String variableName, int minInterval) throws PublisherException;
```

Figure A.11: The publisher Java API provided by the framework, part 2 of 3.

```
  /**
  * The <code>registerPublishFloatAlert</code> method is to register a publish with the edgeStatusRouter.
  * <BR>
  * @param variableName The name of the variable which evente is to be published.
  * @param minInterval The minimum interval between two alerts being published. This will
  * prevent the publisher from flooding the network.
  * @return Returns the variableId of the registered publication.
  * @throws PublisherException If something went wrong during the registration.
  */
  public int registerPublishFloatAlert(String variableName, int minInterval) throws PublisherException;

  /**
  * The <code>registerPublishBooleanAlert</code> method is to register a publish with the edgeStatusRouter.
  * <BR>
  * @param variableName The name of the variable which evente is to be published.
  * @param minInterval The minimum interval between two alerts being published. This will
  * prevent the publisher from flooding the network.
  * @return Returns the variableId of the registered publication.
  * @throws PublisherException If something went wrong during the registration.
  */
  public int registerPublishBooleanAlert(String variableName, int minInterval) throws PublisherException;


  /**
  * The <code>registerPublishUserDefinedAlert</code> method is to register a publish with the edgeStatusRouter.
  * <BR>
  * @param variableName The name of the variable which evente is to be published.
  * @param minInterval The minimum interval between two alerts being published. This will
  * prevent the publisher from flooding the network.
  * @param userType The type of the user registered type.
  * @param length The length iin bytes of the user defined type.
  * @return Returns the variableId of the registered publication.
  * @throws PublisherException If something went wrong during the registration.
  */
  public int registerPublishUserDefinedAlert(String variableName, int minInterval, int userType, int length) throws PublisherException;

  /**
  * The <code>unregisterPublish</code> method is to unregister a previous publish with the edgeStatusRouter.
  * <BR>
  * @param variableId The id of the message to be published.
  * @return Returns <code>0</code> if unregistered from ControlServer,
  * error code otherwise.
  */
  public short unregisterPublish(int variableId);
}
```

Figure A.12: The publisher Java API provided by the framework, part 3 of 3.

```
package edu.wsu.gridstat.publisher.interfaces;

/**
 * <p>Title: PublisherWithDV </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// Java packages.
import java.util.Vector;

// GridStat packages.
import edu.wsu.gridstat.publisher.common.PublisherException;

public interface PublisherWithDVInterface extends PublisherInterface
{
  /**
  * The <code>registerPublishInt</code> method is to register a publish with the edgeStatusRouter.
  * <BR>
  * @param variableName The name of the variable which evente is to be published.
  * @param priority The priority of this status variable.
  * @param intervalLow The lowest interval that the message will be published.
  * @param intervalHigh The highest interval that the message will be published.
  * @param dvs A vector of derived values to be added to this publication.
  * @return Returns the variableId of the registered publication.
  * @throws PublisherException If something went wrong during the registration.
  */
  public int registerPublishInt(String variableName, short priority, int intervalLow,
                                int intervalHigh, Vector dvs) throws PublisherException;

  /**
  * The <code>registerPublishFloat</code> method is to register a publish with the edgeStatusRouter.
  * <BR>
  * @param variableName The name of the variable which evente is to be published.
  * @param priority The priority of this status variable.
  * @param intervalLow The lowest interval that the message will be published.
  * @param intervalHigh The highest interval that the message will be published.
  * @param dvs A vector of derived values to be added to this publication.
  * @return Returns the variableId of the registered publication.
  * @throws PublisherException If something went wrong during the registration.
  */
  public int registerPublishFloat(String variableName, short priority, int intervalLow,
                                  int intervalHigh, Vector dvs) throws PublisherException;

  /**
   * The <code>registerPublishBoolean</code> method is to register a publish with the edgeStatusRouter.
   * <BR>
   * @param variableName The name of the variable which evente is to be published.
   * @param priority The priority of this status variable.
   * @param intervalLow The lowest interval that the message will be published.
   * @param intervalHigh The highest interval that the message will be published.
   * @param dvs A vector of derived values to be added to this publication.
   * @return Returns the variableId of the registered publication.
   * @throws PublisherException If something went wrong during the registration.
   */
  public int registerPublishBoolean(String variableName, short priority, int intervalLow,
                                    int intervalHigh, Vector dvs) throws PublisherException;

  /**
   * The <code>registerPublishUserDefined</code> method is to register a publish with the edgeStatusRouter.
   * <BR>
   * @param variableName The name of the variable which evente is to be published.
   * @param priority The priority of this status variable.
   * @param intervalLow The lowest interval that the message will be published.
   * @param intervalHigh The highest interval that the message will be published.
   * @param userType The type of the user registered type.
   * @param length The length in bytes of the user defined type.
   * @param dvs A vector of derived values to be added to this publication.
   * @return Returns the variableId of the registered publication.
   * @throws PublisherException If something went wrong during the registration.
   */
  public int registerPublishUserDefined(String variableName, short priority, int intervalLow,
                                        int intervalHigh, int userType, int length, Vector dvs) throws PublisherException;
}
```

Figure A.13: The publisher with derived value Java API provided by the framework.

```
package edu.wsu.gridstat.publisher.interfaces;

/**
 * <p>Title: DVBaseInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */
public abstract interface DVBaseInterface
{
  /**
  * The <code>getPatternId</code> method is used to return the Id of this pattern type.
  * <BR>
  * @return Returns the pattern id.
  */
  public int getPatternId();

  /**
  * The <code>getDataType</code> method is used to return the dataType of this pattern.
  * <BR>
  * @return Returns the data type of this pattern.
  */
  public int getDataType();

  /**
  * The <code>getDataTypeLength</code> method is used to return the length (in bytes) of the
  * pattern that is produced.
  * <BR>
  * @return Returns the length of the pattern that is produced.
  */
  public int getDataTypeLength();
}

package edu.wsu.gridstat.publisher.interfaces;

/**
 * <p>Title: DVIntInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */
public abstract interface DVIntInterface extends DVBaseInterface
{
  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract int pubSetValue(int value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract int pubSetValue(float value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract int pubSetValue(boolean value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract int pubSetValue(byte[] value);
}
```

Figure A.14: The publisher derived value holder base and derived value holder for integer type.

```
package edu.wsu.gridstat.publisher.interfaces;

/**
 * <p>Title: DVFloatInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */
public abstract interface DVFloatInterface extends DVBaseInterface
{
  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract float pubSetValue(int value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract float pubSetValue(float value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract float pubSetValue(boolean value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract float pubSetValue(byte[] value);
}

package edu.wsu.gridstat.publisher.interfaces;

/**
 * <p>Title: DVBooleanInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */
public abstract interface DVBooleanInterface extends DVBaseInterface
{
  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract boolean pubSetValue(int value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract boolean pubSetValue(float value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract boolean pubSetValue(boolean value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract boolean pubSetValue(byte[] value);
}
```

Figure A.15: The publisher derived value holders for float and boolean types.

```
package edu.wsu.gridstat.publisher.interfaces;

/**
 * <p>Title: DVUserDefinedInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */
public abstract interface DVUserDefinedInterface extends DVBaseInterface
{
  /**
  * The <code>getRealType</code> method is used to return the real data type of this variable.
  * <BR>
  * @return Returns the real type of this variable.
  */
  public int getRealType();

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract byte[] pubSetValue(int value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract byte[] pubSetValue(float value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract byte[] pubSetValue(boolean value);

  /**
  * The <code>pubSetValue</code> method is used by the publisher to set a new value that is being published.
  * <BR>
  * @param value The value that is being published.
  * @return Returns the current pattern for this variable.
  */
  public abstract byte[] pubSetValue(byte[] value);
}
```

Figure A.16: The publisher derived value holder for user-defined type.

```
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: SubscriberInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// Standard java packages.
import java.util.concurrent.ArrayBlockingQueue;

// GridStat packages.
import edu.wsu.gridstat.util.IntHashMap;

public interface SubscriberInterface
{
  /**
  * The <code>connectToEdge</code> method is used to connect to a edgeStatusRouter.
  * <BR>
  * @return Returns <code>0</code> if connected, error code otherwise.
  */
  public short connectToEdge();

  /**
  * The <code>disConnectFromEdge</code> method is used to disconnect from a edgeStatusRouter.
  * <BR>
  * @return Returns <code>0</code> all subscriptions is unregistered and we disconnect
  * from the edge, error code otherwise.
  */
  public short disConnectFromEdge();

  /**
  * The <code>isConnected</code> method is used to return if we are connected to the edgeStatusRouter.
  * <BR>
  * @return Returns <code>true</code> if we are connected,
  * <code>false</code> otherwise.
  */
  public boolean isConnected();

  /**
  * The <code>getCurrentMode</code> method is used to find out the current operating mode.
  * <BR>
  * @return Return the current operating mode.
  */
  public int getCurrentMode();

  /**
  * The <code>isModeChange</code> method is used to return to the application layer if a mode change is in progress.
  * <BR>
  * @return Return <code>true</code> if a mode change is in progress, <code>false</code> otherwise.
  */
  public boolean isModeChange();

  /**
  * The <code>getNextMode</code> method is used to return the next mode if a mode change is in progress.
  * <BR>
  * @return Return the next mode, or if no mode change returns the current mode.
  */
  public int getNextMode();

  /**
  * The <code>subscribeInt</code> method is used to subscribe to a status variable of type int. The leaf QoS broker will set up an
  * path so that we will receive the latest value of the variable.
  * <BR>
  * @param publisherName The name of the publisher.
  * @param variableName The name of the variable to be subscribed to.
  * @param modes The modes that this subscription will be valid in.
  * @param interval The interval that we wish to receive the events.
  * @param priority The priority of the subscription.
  * @param latency The latency that we would like for this subscription.
  * @param redundancy The redundancy that we would like for this subscription.
  * @param holderObject Where the subscribed to value will be placed.
  * @return Returns true if message is subscribed to, false otherwise.
  */
  public short subscribeInt(String publisherName, String variableName, int modes, int[] interval, short[] priority,
                            short[] latency, short[] redundancy, HolderIntInterface holderObject);
```

Figure A.17: The subscriber Java API provided by the framework, part 1 of 3.

```
/**
* The <code>subscribeFloat</code> method is used to subscribe to a status variable of type float. The leaf QoS broker will set up an path
* so that we will receive the latest value of the variable.
* <BR>
* @param publisherName The name of the publisher.
* @param variableName The name of the variable to be subscribed to.
* @param modes The modes that this subscription will be valid in.
* @param interval The interval that we wish to receive the events.
* @param priority The priority of the subscription.
* @param latency The latency that we would like for this subscription.
* @param redundancy The redundancy that we would like for this subscription.
* @param holderObject Where the subscribed to value will be placed.
* @return Returns true if message is subscribed to, false otherwise.
*/
public short subscribeFloat(String publisherName, String variableName, int modes, int[] interval, short[] priority,
                            short[] latency, short[] redundancy, HolderFloatInterface holderObject);

/**
* The <code>subscribeBoolean</code> method is used to subscribe to a status variable of type boolean. The leaf QoS broker will set up
* an path so that we will receive the latest value of the variable.
* <BR>
* @param publisherName The name of the publisher.
* @param variableName The name of the variable to be subscribed to.
* @param modes The modes that this subscription will be valid in.
* @param interval The interval that we wish to receive the events.
* @param priority The priority of the subscription.
* @param latency The latency that we would like for this subscription.
* @param redundancy The redundancy that we would like for this subscription.
* @param holderObject Where the subscribed to value will be placed.
* @return Returns true if message is subscribed to, false otherwise.
*/
public short subscribeBoolean(String publisherName, String variableName, int modes, int[] interval, short[] priority,
                              short[] latency, short[] redundancy, HolderBooleanInterface holderObject);

/**
* The <code>subscribeUserDefined</code> method is used to subscribe to a status variable of type boolean. The leaf QoS broker will set
* up an path so that we will receive the latest value of the variable.
* <BR>
* @param publisherName The name of the publisher.
* @param variableName The name of the variable to be subscribed to.
* @param modes The modes that this subscription will be valid in.
* @param interval The interval that we wish to receive the events.
* @param priority The priority of the subscription.
* @param latency The latency that we would like for this subscription.
* @param redundancy The redundancy that we would like for this subscription.
* @param holderObject Where the subscribed to value will be placed.
* @param userType The type of the user registered type.
* @return Returns true if message is subscribed to, false otherwise.
*/
public short subscribeUserDefined(String publisherName, String variableName, int modes, int[] interval, short[] priority,
                                  short[] latency, short[] redundancy, HolderUserDefinedInterface holderObject, int userType);

/**
* The <code>unSubscribe</code> method is used to unsubscribe from one of the variables that we have subscribed to.
* <BR>
* @param publisherName The name of the publisher.
* @param variableName The name of the variable subscribed to.
* @return Returns <code>0</code> if message is unSubscribed to, error code
* otherwise.
*/
public short unSubscribe(String publisherName, String variableName);

/**
* The <code>unSubscribeAll</code> method is used to remove all the subscriptions.
* <BR>
* @return returns <code>0</code> of all the subscriptions is removed,
* error code otherwise.
*/
public short unSubscribeAll();

/**
* The <code>subscribeAll</code> method is used to subscribe to all flooding alerts that we will receive.
* <BR>
* @param floodedVariables This is where we will store all the flooded values that we will receive
* @param floodedReceived Uses this one to signal the user that a event has been recveived.
*     NOTE 1: if you don't want to be signaled then give <code>null</code> for this argument.
*     NOTE 2: The type that is signaled to you is <code>HolderUnknown</code>.
*/
public void subscribeAllFlooding(IntHashMap floodedVariables, ArrayBlockingQueue floodedReceived);

/**
* The <code>unSubscribeAllFlooding</code> method is used to unsubscribe from
* receiving all the flooded events.
*/
public void unSubscribeAllFlooding();
```

Figure A.18: The subscriber Java API provided by the framework, part 2 of 3.

```
  /**
   * The <code>createGroupSubscription</code> method is used to subscribe to a status variable of type int that will be added to a group.
   * The leaf QoS broker will set up an path so that we will receive the latest value of the variable.
   * <BR>
   * @param groupName The name of the group to be created.
   * @param history How many values should be held for each variable in the history.
   * @param modes The modes that this subscription will be valid in.
   * @param subInterval The interval that we wish to receive the events.
   * @param priority The priority of the subscription.
   * @return Returns the created group, null if it already existed.
   */
  public HolderBaseGroupInterface createGroupSubscription(String groupName, int history, int modes, int[] subInterval, short[] priority);

  /**
   * The <code>removeGroupSubscription</code> method is used to remove one of the group subscriptions that we have.
   * <BR>
   * @param groupName The name of the group to be removed.
   * @return returns <code>true</code> if the subscription is removed, <code>false</code> otherwise.
   */
  public boolean removeGroupSubscription(String groupName);

  /**
   * The <code>subscribeIntForGroup</code> method is used to subscribe to a status variable of type int that will be added to a group.
   * The leaf QoS broker will set up an path so that we will receive the latest value of the variable.
   * <BR>
   * @param groupName The name of the group where the subscription will belong.
   * @param publisherName The name of the publisher.
   * @param variableName The name of the variable to be subscribed to.
   * @param latency The latency that we would like for this subscription.
   * @param redundancy The redundancy that we would like for this subscription.
   * @return Returns true if message is subscribed to, false otherwise.
   */
  public short subscribeIntForGroup(String groupName, String publisherName, String variableName, short[] latency, short[] redundancy);

  /**
   * The <code>subscribeFloatForGroup</code> method is used to subscribe to a status variable of type float that will be added to a group.
   * The leaf QoS broker will set up an path so that we will receive the latest value of the variable.
   * <BR>
   * @param groupName The name of the group where the subscription will belong.
   * @param publisherName The name of the publisher.
   * @param variableName The name of the variable to be subscribed to.
   * @param latency The latency that we would like for this subscription.
   * @param redundancy The redundancy that we would like for this subscription.
   * @return Returns true if message is subscribed to, false otherwise.
   */
  public short subscribeFloatForGroup(String groupName, String publisherName, String variableName, short[] latency, short[] redundancy);

  /**
   * The <code>subscribeBooleanForGroup</code> method is used to subscribe to a status variable of type boolean that will be added to a
   * group. The leaf QoS broker will set up an path so that we will receive the latest value of the variable.
   * <BR>
   * @param groupName The name of the group where the subscription will belong.
   * @param publisherName The name of the publisher.
   * @param variableName The name of the variable to be subscribed to.
   * @param latency The latency that we would like for this subscription.
   * @param redundancy The redundancy that we would like for this subscription.
   * @return Returns true if message is subscribed to, false otherwise.
   */
  public short subscribeBooleanForGroup(String groupName, String publisherName, String variableName, short[] latency, short[] redundancy);

  /**
   * The <code>subscribeUserDefinedForGroup</code> method is used to subscribe to a status variable of type UserDefined that will be added
   * to a group. The leaf QoS broker will set up an path so that we will receive the latest value of the variable.
   * <BR>
   * @param groupName The name of the group where the subscription will belong.
   * @param publisherName The name of the publisher.
   * @param variableName The name of the variable to be subscribed to.
   * @param latency The latency that we would like for this subscription.
   * @param redundancy The redundancy that we would like for this subscription.
   * @param holderObject The object of user defined type where the values will be stored.
   * @param userType The type of the object that the user have defined.
   * @return Returns true if message is subscribed to, false otherwise.
   */
  public short subscribeUserDefinedForGroup(String groupName, String publisherName, String variableName, short[] latency,
                                            short[] redundancy, HolderUserDefinedInterface holderObject, int userType);

  /**
   * The <code>unSubscribeForGroup</code> method is used to unsubscribe from one of the variables that we been subscribed to for this group.
   * <BR>
   * @param groupName The name of the group where the subscription belong.
   * @param publisherName The name of the publisher.
   * @param variableName The name of the variable subscribed to.
   * @return Returns <code>0</code> if message is unSubscribed to, error code
   * otherwise.
   */
  public short unSubscribeForGroup(String groupName, String publisherName, String variableName);
}
```

Figure A.19: The subscriber Java API provided by the framework, part 3 of 3.

```java
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: SubscriberWithDVInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface SubscriberWithDVInterface extends SubscriberInterface
{
  /**
  * The <code>subscribeInt</code> method is used to subscribe to a status variable of type int. The leaf QoS broker will set up an
  * path so that we will receive the latest value of the variable.
  * <BR>
  * @param publisherName The name of the publisher.
  * @param variableName The name of the variable to be subscribed to.
  * @param modes The modes that this subscription will be valid in.
  * @param interval The interval that we wish to receive the events.
  * @param priority The priority of the subscription.
  * @param latency The latency that we would like for this subscription.
  * @param redundancy The redundancy that we would like for this subscription.
  * @param holderObject Where the subscribed to value will be placed.
  * @return Returns true if message is subscribed to, false otherwise.
  */
  public short subscribeInt(String publisherName, String variableName, int modes, int[] interval, short[] priority,
                            short[] latency, short[] redundancy, HolderIntWithDVInterface holderObject);

  /**
  * The <code>subscribeFloat</code> method is used to subscribe to a status variable of type float. The leaf QoS broker will set up an path
  * so that we will receive the latest value of the variable.
  * <BR>
  * @param publisherName The name of the publisher.
  * @param variableName The name of the variable to be subscribed to.
  * @param modes The modes that this subscription will be valid in.
  * @param interval The interval that we wish to receive the events.
  * @param priority The priority of the subscription.
  * @param latency The latency that we would like for this subscription.
  * @param redundancy The redundancy that we would like for this subscription.
  * @param holderObject Where the subscribed to value will be placed.
  * @return Returns true if message is subscribed to, false otherwise.
  */
  public short subscribeFloat(String publisherName, String variableName, int modes, int[] interval, short[] priority,
                              short[] latency, short[] redundancy, HolderFloatWithDVInterface holderObject);

  /**
  * The <code>subscribeBoolean</code> method is used to subscribe to a status variable of type boolean. The leaf QoS broker will set up
  * an path so that we will receive the latest value of the variable.
  * <BR>
  * @param publisherName The name of the publisher.
  * @param variableName The name of the variable to be subscribed to.
  * @param modes The modes that this subscription will be valid in.
  * @param interval The interval that we wish to receive the events.
  * @param priority The priority of the subscription.
  * @param latency The latency that we would like for this subscription.
  * @param redundancy The redundancy that we would like for this subscription.
  * @param holderObject Where the subscribed to value will be placed.
  * @return Returns true if message is subscribed to, false otherwise.
  */
  public short subscribeBoolean(String publisherName, String variableName, int modes, int[] interval, short[] priority,
                                short[] latency, short[] redundancy, HolderBooleanWithDVInterface holderObject);

  /**
  * The <code>subscribeUserDefined</code> method is used to subscribe to a status variable of type boolean. The leaf QoS broker will set
  * up an path so that we will receive the latest value of the variable.
  * <BR>
  * @param publisherName The name of the publisher.
  * @param variableName The name of the variable to be subscribed to.
  * @param modes The modes that this subscription will be valid in.
  * @param interval The interval that we wish to receive the events.
  * @param priority The priority of the subscription.
  * @param latency The latency that we would like for this subscription.
  * @param redundancy The redundancy that we would like for this subscription.
  * @param holderObject Where the subscribed to value will be placed.
  * @param userType The type of the user registered type.
  * @return Returns true if message is subscribed to, false otherwise.
  */
  public short subscribeUserDefined(String publisherName, String variableName, int modes, int[] interval, short[] priority,
                                    short[] latency, short[] redundancy, HolderUserDefinedWithDVInterface holderObject, int userType);
```

Figure A.20: The subscriber with derived value Java API provided by the framework.

```
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderBaseInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public abstract interface HolderBaseInterface
{
  /**
  * The <code>setVariableId</code> method is used to set the variableId of this variable.
  * <BR>
  * @param variableId The variable id of this variable.
  */
  public void setVariableId(int variableId);

  /**
  * The <code>getVariableId</code> method is used to return the variableId of this variable.
  * <BR>
  * @return Returns the variableId of this variable.
  */
  public int getVariableId();

  /**
  * The <code>getModes</code> method is used to return the modes that this subscription is subscribing to.
  * <BR>
  * @return Returns the modes of this variable.
  */
  public int getModes();

  /**
  * The <code>setTime</code> method is used to set the time for the latest value.
  * <BR>
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated Is this value extrapolated.
  */
  public void setTime(long sentTime, long receivedTime, boolean extrapolated);

  /**
  * The <code>getTimeSent</code> method is used to return the time when this value was sent/created.
  * <BR>
  * @return Returns the time when this value was sent/created.
  */
  public long getTimeSent();

  /**
  * The <code>getTimeReceived</code> method is used to return the time when this value was received.
  * <BR>
  * @return Returns the time when this value was received.
  */
  public long getTimeReceived();

  /**
  * The <code>getLatency</code> method is used to return the latency that this value experienced when it was transmitted to us.
  * <BR>
  * @return Returns the latency "of the" value.
  */
  public long getLatency();

  /**
  * The <code>getPublisherName</code> method is used to return the name of the publisher of this variable.
  * <BR>
  * @return Returns the name of the publisher.
  */
  public String getPublisherName();

  /**
  * The <code>getVariableName</code> method is used to return the name of the* variable.
  * <BR>
  * @return Returns the name of the variable.
  */
  public String getVariableName();

  /**
  * The <code>getType</code> method is used to return the type of this variable.
  * <BR>
  * @return Returns the type of the variable.
  */
  public short getType();
```

Figure A.21: The subscriber status variable holder base, part 1 of 2.

```
  /**
  * The <code>getSubInterval</code> method is used to return the subscription interval of this variable.
  * <BR>
  * @return Returns the subscription interval of the variable.
  */
  public int[] getSubInterval();

  /**
  * The <code>getExtrapolated</code> method is used to return if the current value of the variable was extraploated or not.
  * <BR>
  * @return Returns if the current value was extrapolated or not.
  */
  public boolean getExtrapolated();

  /**
  * The <code>getErrorCode</code> method is used to return the current error code if any.
  * <BR>
  * @return Returns the current error code.
  */
  public short getErrorCode();

  /**
  * The <code>getAndResetErrorCode</code> method is used to return the current error code if any, and reset it.
  * <BR>
  * @return Returns the current error code.
  */
  public short getAndResetErrorCode();

  /**
  * The <code>setErrorCode</code> method is used to set the error code.
  * <BR>
  * @param error The new error code.
  */
  public void setErrorCode(short error);

  /**
  * The <code>registerQoSViolationListener</code> method is used to register that callbacks will be made if QoS is violated.
  * <BR>
  * @param qvi If QoS is violated this this method is called.
  * @return Returns <code>true</code> if the qvl is registered, <code>false</code. otherwise.
  */
  public boolean registerQoSViolationListener(QoSViolationInterface qvi);

  /**
  * The <code>isQoSViolationListener</code> method is used to return if QoSViolationListener is activated.
  * <BR>
  * @return Returns if QoSViolationListener is activated.
  */
  public boolean isQoSViolationListener();

  /**
  * The <code>unregisterQoSViolationListener</code> method is used to unregister the callback.
  */
  public void unregisterQoSViolationListener();

  /**
  * The <code>registerEventNotificationListener</code> method is used to register that the user wants a push interaction style.
  * A listener is registered that is called when an event is received.
  * <BR>
  * @param eni The listener that the user registers.
  * @return Returns <code>true</code> if the enl is registered, <code>false</code. otherwise.
  */
  public boolean registerEventNotificationListener(EventNotificationInterface eni);

  /**
  * The <code>isEventNotificationListener</code> method is used to find out if this holder uses push interaction.
  * <BR>
  * @return Returns <code>true</code> if push interaction is used, <code>false</code> otherwise.
  */
  public boolean isEventNotificationListener();

  /**
  * The <code>unregisterEventNotificationListener</code> method is used to unregister the notification of received events.
  */
  public void unregisterEventNotificationListener();

  /**
  * The <code>clone</code> method is used to return a deep copy of this object
  * <BR>
  * @return Returns a deep copy of this object.
  */
  public abstract Object clone();
}
```

Figure A.22: The subscriber status variable holder base, part 2 of 2.

```
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderIntInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface HolderIntInterface extends HolderBaseInterface
{
  /**
  * The <code>getValue</code> method is used to return the current value of
  * this variable.
  * <BR>
  * @return Returns the current value of this variable.
  */
  public int getValue();

  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param value The new value of the variable.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated If this value is extrapolated or not.
  * @return Return <code>true</code> if value set, <code>false</code> otherwise
  */
  public boolean setValue(int value, long sentTime, long receivedTime, boolean extrapolated);
}


package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderFloatInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface HolderFloatInterface extends HolderBaseInterface
{
  /**
  * The <code>getValue</code> method is used to return the current value of
  * this variable.
  * <BR>
  * @return Returns the current value of this variable.
  */
  public float getValue();

  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param value The new value of the variable.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated If this value is extrapolated or not.
  * @return Return <code>true</code> if value set, <code>false</code> otherwise
  */
  public boolean setValue(float value, long sentTime, long receivedTime, boolean extrapolated);
}
```

Figure A.23: The subscriber holders for int type and float type.

```java
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderBooleanInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface HolderBooleanInterface extends HolderBaseInterface
{
  /**
  * The <code>getValue</code> method is used to return the current value of
  * this variable.
  * <BR>
  * @return Returns the current value of this variable.
  */
  public boolean getValue();

  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param value The new value of the variable.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @return Return <code>true</code> if value set, <code>false</code> otherwise
  */
  public boolean setValue(boolean value, long sentTime, long receivedTime);
}


package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderUserDefinedInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// Standard java packages.
import java.nio.ByteBuffer;

public abstract interface HolderUserDefinedInterface extends HolderBaseInterface
{
  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param packet The buffers where the event is encoded.
  * @param startIdx The index of which of the buffers that this event start.
  * @param len The number of buffers that belong to this event.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated If this value is extrapolated or not.
  * @return Return <code>true</code> if value set, <code>false</code> otherwise
  */
  public boolean setValue(ByteBuffer[] packet, int startIdx, int len, long sentTime, long receivedTime, boolean extrapolated);

  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param value The value of the event received.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated If this value is extrapolated or not.
  * @return Return <code>true</code> if value set, <code>false</code> otherwise
  */
  public boolean setValue(byte[] value, long sentTime, long receivedTime, boolean extrapolated);

  /**
  * The <code>getLength</code> method is used to return the length, number of bytes" of the datatype of the variable.
  * <BR>
  * @return Return the number of bytes of the datatype.
  */
  public int getLength();

  /**
  * The <code>getRealType</code> method is used to return the real data type of this variable.
  * <BR>
  * @return Returns the real type of this variable.
  */
  public int getRealType();
}
```

Figure A.24: The subscriber holders for boolean type and user-defined type.

```
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderBaseGroupInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// Standard java packages.
import java.util.Vector;

public interface HolderBaseGroupInterface
{
  /**
  * The <code>getLatestValues</code> method is used to return a set of the latest values that is received for the subscriptions
  * in the group.
  * <BR>
  * @return Returns a Vector of <code>HolderBase</code> that are the latest values received.
  */
  public Vector getLatestValues();

  /**
  * The <code>getLatestValues</code> method is used to return the latest n sets of values that is received for the subscriptions
  * in the group.
  * <BR>
  * @param history Have many of the past sets of values should be returned.
  * @return Returns a Vector of <code>HolderBase</code> that are the latest values received.
  */
  public Vector[] getLatestValues(int history);

  /**
  * The <code>getSubInterval</code> method is used to return the interval for the subscriptions in this group.
  * <BR>
  * @return Returns the interval for the subscriptions.
  */
  public int[] getSubInterval();

  /**
  * The <code>getModes</code> method is used to return the mode for the subscriptions in this group.
  * <BR>
  * @return Returns the mode for the subscriptions.
  */
  public int getModes();

  /**
  * The <code>getPriority</code> method is used to return the priority for the subscriptions in this group.
  * <BR>
  * @return Returns the priority for the subscriptions.
  */
  public short[] getPriority();

  /**
  * The <code>getGroupName</code> method is used to return the name of this subscriptions group.
  * <BR>
  * @return Returns the name of this subscription group.
  */
  public String getGroupName();

  /**
  * The <code>eventReceived</code> method is used to be informed that a variable that we subscribe to has received an event.
  * <BR>
  * @param holder The holder for the variable that got a new event.
  * @throws java.lang.InterruptedException If the threade is interrupted while executing this method.
  */
  public void eventReceived(HolderBaseInterface holder) throws InterruptedException;
}
```

Figure A.25: The subscriber holder for group subscription.

```
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderBaseWithDVInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// Standard java packages.
import java.util.Vector;

// GridStat packages.
import edu.wsu.gridstat.subscriber.derivedValue.DVHolderBase;

public abstract interface HolderBaseWithDVInterface extends HolderBaseInterface
{
  /**
  * The <code>registerDV</code> method is used to register this holder to
  * decode and store the latest value of this pattern.
  * <BR>
  * @param patternHolder Where the latest valus of the dv will be stored.
  * @return Returns <code>true</code> if we registered the pattern, note the pattern might not
  * be published, when we know this we will set Constants.PATTERN_PUBLISHED_NO in the patternHolder.
  * <code>false</code> is returned if we know that this pattern is not published or type mismatch.
  */
  public boolean registerDV(DVHolderBase patternHolder);

  /**
  * The <code>unregisterDV</code> method is used to unregister this holder from
  * decode and store the latest value of this pattern.
  * <BR>
  * @param patternId The is of the pattern to listen to.
  * @return Returns the holder that was used for this pattern.
  */
  public DVHolderBase unregisterDV(int patternId);

  /**
  * The <code>getPubDVs</code> method is used to return all the patterns that the publisher is publishing.
  * <BR>
  * @return Returns a Vector containing PatternInfoHolder of the patterns that the publisher is publishing.
  */
  public Vector getPubDVs();

  /**
  * The <code>clone</code> method is used to return a deep copy of this object
  * <BR>
  * @return Returns a deep copy of this object.
  */
  public abstract Object clone();
}
```

Figure A.26: The subscriber status variable with derived value holder base.

```java
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderIntWithDVInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface HolderIntWithDVInterface extends HolderBaseWithDVInterface
{
  /**
  * The <code>getValue</code> method is used to return the current value of
  * this variable.
  * <BR>
  * @return Returns the current value of this variable.
  */
  public int getValue();

  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param value The new value of the variable.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated If this value is extrapolated or not.
  */
  public void setValue(int value, long sentTime, long receivedTime, boolean extrapolated);
}


package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderFloatWithDVInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface HolderFloatWithDVInterface extends HolderBaseWithDVInterface
{
  /**
  * The <code>getValue</code> method is used to return the current value of this variable.
  * <BR>
  * @return Returns the current value of this variable.
  */
  public float getValue();

  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param value The new value of the variable.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated If this value is extrapolated or not.
  */
  public void setValue(float value, long sentTime, long receivedTime, boolean extrapolated);
}
```

Figure A.27: The subscriber holders with derived value for integer type and float type.

```java
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderBooleanWithDVInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface HolderBooleanWithDVInterface extends HolderBaseWithDVInterface
{
  /**
  * The <code>getValue</code> method is used to return the current value of this variable.
  * <BR>
  * @return Returns the current value of this variable.
  */
  public boolean getValue();

  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param value The new value of the variable.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  */
  public void setValue(boolean value, long sentTime, long receivedTime);
}


package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderUserDefinedWithDVInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// Standard java packages.
import java.nio.ByteBuffer;

public interface HolderUserDefinedWithDVInterface extends HolderBaseWithDVInterface
{
  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param packet The buffers where the event is encoded.
  * @param startIdx The index of which of the buffers that this event start.
  * @param len The number of buffers that belong to this event.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated If this value is extrapolated or not.
  */
  public void setValue(ByteBuffer[] packet, int startIdx, int len, long sentTime, long receivedTime, boolean extrapolated);

  /**
  * The <code>setValue</code> method is used to set the value of this variable.
  * <BR>
  * @param value The value of the event received.
  * @param sentTime The time when this value was sent/created.
  * @param receivedTime The time that we received this value.
  * @param extrapolated If this value is extrapolated or not.
  */
  public void setValue(byte[] value, long sentTime, long receivedTime, boolean extrapolated);

  /**
  * The <code>getLength</code> method is used to return the length, number
  * of bytes" of the datatype of the variable.
  * <BR>
  * @return Return the number of bytes of the datatype.
  */
  public int getLength();

  /**
  * The <code>getRealType</code> method is used to return the real data type of this variable.
  * <BR>
  * @return Returns the real type of this variable.
  */
  public int getRealType();
}
```

Figure A.28: The subscriber holders with derived value for boolean type and user-defined type.

```
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: ExtrapolationInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */
public abstract interface ExtrapolationInterface
{
  /**
   * The <code>extrapolateValue</code> method is used to extrapolate the next value in the incomming sequence.
   * <BR>
   * @param xVal The array of xcoordinates
   * @param yVal The array of sample y coordinates.
   * @throws Exception Throws <code>Exception</code> if the calculation doesn't work.
   * @return Returns The extrapolated value.
   */
  public abstract double extrapolateValue(double[] xVal, double[] yVal) throws Exception;
}

package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderIntExtrapolationInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */
public interface HolderIntExtrapolationInterface extends HolderIntInterface
{
  /**
   * The <code>extrapolateValue</code> method is used to extrapolate a missed value for values already received.
   * <BR>
   * @param sentTime The time that this event is supposed to be sent.
   * @param receivedTime The time that this event is created/extrapolated.
   * @return Returns <code>true</code> if a value could be extrapolated, <code>false</code> otherwise.
   */
  public boolean extrapolateValue(long sentTime, long receivedTime);
}

package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: HolderFloatExtrapolationInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University</p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface HolderFloatExtrapolationInterface extends HolderFloatInterface
{
  /**
   * The <code>extrapolateValue</code> method is used to extrapolate a missed value for values already received.
   * <BR>
   * @param sentTime The time that this event is supposed to be sent.
   * @param receivedTime The time that this event is created/extrapolated.
   * @return Returns <code>true</code> if a value could be extrapolated, <code>false</code> otherwise.
   */
  public boolean extrapolateValue(long sentTime, long receivedTime);
}
```

Figure A.29: The subscriber status variable holders with the extrapolation mechanism.

```
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: EventNotificationInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Washington State University</p>
 * @author Kjell harald Gjermundrod
 * @version 1.0
 */

public interface EventNotificationInterface
{
  /**
  * The <code>eventReceived</code> method is used to be informed that a variable that we subscribe to has received an event.
  * <BR>
  * @param holder The holder for the variable that got a new event.
  * @throws java.lang.InterruptedException If the threade is interrupted while executing this method.
  */
  public void eventReceived(HolderBaseInterface holder) throws InterruptedException;

  /**
  * The <code>eventReceived</code> method is used to be informed that a variable that we subscribe to has received an event.
  * <BR>
  * @param holder The holder for the variable that got a new event.
  * @param delay The end-to-end delay of this event.
  * @throws java.lang.InterruptedException If the threade is interrupted while executing this method.
  */
  public void eventReceived(HolderBaseInterface holder, int delay) throws InterruptedException;
}


package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: EventForGroupNotificationInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Washington State University</p>
 * @author Kjell harald Gjermundrod
 * @version 1.0
 */

public interface EventForGroupNotificationInterface
{
  /**
  * The <code>eventsReceived</code> method is used to be informed that all the
  * subscribed variables has received a new value.
  * <BR>
  * @param holder The holder for the group that has receiveda new set of values.
  * @throws java.lang.InterruptedException If the threade is interrupted while executing this method.
  */
  public void eventsReceived(HolderBaseGroupInterface holder) throws InterruptedException;
}
package edu.wsu.gridstat.subscriber.interfaces;

/**
 * <p>Title: QoSViolationInterface</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Washington State University</p>
 * @author Kjell harald Gjermundrod
 * @version 1.0
 */

public interface QoSViolationInterface
{
  /**
  * The <code>qosViolated</code> method is used to be informed that a qos violation occured .
  * <BR>
  * @param holder The holder for the variable that got a had a violation.
  * @throws java.lang.InterruptedException If the threade is interrupted while executing this method.
  */
  public void qosViolated(HolderBaseInterface holder) throws InterruptedException;
}
```

Figure A.30: Subscriber push interaction model and QoS callback interfaces.

```java
package edu.wsu.gridstat.condensationCreator.interfaces;

/**
 * <p>Title: CondensationInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public interface CondensationInterface
{
  /**
  * The <code>connectToEdge</code> method is used to connect to a edgeStatusRouter.
  * <BR>
  * @return Returns <code>0</code> if connected, error code otherwise.
  */
  public short connectToEdge();

  /**
  * The <code>disConnectFromEdge</code> method is used to disconnect from a edgeStatusRouter.
  * <BR>
  * @return Returns <code>true</code> we have unregisteded all the publications and we have disconnected, <code>false</code> otherwise.
  */
  public boolean disConnectFromEdge();

  /**
  * The <code>isConnected</code> method is used return if we are connected to the edgeStatusRouter or not.
  * <BR>
  * @return Returns <code>true</code> if we are connected, <code>false</code> otherwise.
  */
  public boolean isConnected();

  /**
  * The <code>registerCondCreator</code> method is used to register this condensation
  * creator.
  * <BR>
  * @return Returns <code>CommConstants.COMMAND_OK</code> if the condensation creator is registrered,
  * error code (Negative number) otherwise.
  */
  public short registerCondCreator();

  /**
  * The <code>unregisterCondCreator</code> method is used to unregister this condensation
  * creator.
  * <BR>
  * @return Returns <code>CommConstants.COMMAND_OK</code> if the condensation creator is removed,
  * error code (Negative number) otherwise.
  */
  public short unregisterCondCreator();

  /**
  * The <code>createCondensationHelper</code> method is used pack up the necessary information needed to create a condensation function.
  * Then it issue a request to the QoS management to create this condensation function in the dataplane.
  * <BR>
  * @param pubName The name of the publisher, i.e. this condensation function for the variable produced by it.
  * @param variableName The name if the variable that this condensation function publishes.
  * @param outDataType The datatype of the variable that is published.
  * @param outUserType The user datatype (is a user defined type is used) of the variable that is published.
  * @param outTypeLen The lenght (bytes) of the published variable.
  * @param priority The priority of the published variable.
  * @param intervalLow The low publishing interval.
  * @param intervalHigh The high publishing interval.
  * @param inFilerHigh Should the input values be filtered for a max value.
  * @param inFilterHighVar If high filtering (input) then this is the max value.
  * @param inFilerLow Should the input values be filtered for a minimum value.
  * @param inFilterLowVar If low filtering (input) then this is the minimum value.
  * @param outFilerHigh Should the output value be filtered for a max value.
  * @param outFilterHighVar If high filtering (output) then this is the max value.
  * @param outFilerLow Should the output value be filtered for a minimum value.
  * @param outFilterLowVar If low filtering (output) then this is the minimum value.
  * @param triggerType The triggering type to be used.
  * @param triggerVar1 A variable that can be given to the triggering mechanism.
  * @param calculatorURI The location of the calculator class that should be used for this condensation function.
  * @param calculatorClassName The name of the calculator class that should be used for this condensation function.
  * @return Returns <code>CommConstants.COMMAND_OK</code> if the condensation function could be created, error code otherwise.
  */
  public short createCondensation(String pubName, String variableName, short outDataType, int outUserType, int outTypeLen, short priority,
                                  int intervalLow, int intervalHigh, boolean inFilerHigh, int inFilterHighVar, boolean inFilerLow,
                                  int inFilterLowVar, boolean outFilerHigh, int outFilterHighVar, boolean outFilerLow, int outFilterLowVar,
                                  short triggerType, int triggerVar1, String calculatorURI, String calculatorClassName);
```

Figure A.31: The condensation creator Java API provided by the framework, part 1 of 2.

```
  /**
  * The <code>createCondensationHelper</code> method is used pack up the necessary information needed to create a condenssation function.
  * Then it issue a request to the QoS management to create this condensation function in the dataplane.
  * <BR>
  * @param placementSR Request that the condensation function will be placed on this SR.
  * @param pubName The name of the publisher, i.e. this condensation function for the variable produced by it.
  * @param variableName The name if the variable that this condensation function publishes.
  * @param outDataType The datatype of the variable that is published.
  * @param outUserType The user datatype (is a user defined type is used) of the variable that is published.
  * @param outTypeLen The lenght (bytes) of the published variable.
  * @param priority The priority of the published variable.
  * @param intervalLow The low publishing interval.
  * @param intervalHigh The high publishing interval.
  * @param inFilerHigh Should the input values be filtered for a max value.
  * @param inFilterHighVar If high filtering (input) then this is the max value.
  * @param inFilerLow Should the input values be filtered for a minimum value.
  * @param inFilterLowVar If low filtering (input) then this is the minimum value.
  * @param outFilerHigh Should the output value be filtered for a max value.
  * @param outFilterHighVar If high filtering (output) then this is the max value.
  * @param outFilerLow Should the output value be filtered for a minimum value.
  * @param outFilterLowVar If low filtering (output) then this is the minimum value.
  * @param triggerType The triggering type to be used.
  * @param triggerVar1 A variable that can be given to the triggering mechanism.
  * @param calculatorURI The location of the calculator class that should be used for this condensation function.
  * @param calculatorClassName The name of the calculator class that should be used for this condensation function.
  * @return Returns <code>CommConstants.COMMAND_OK</code> if the condensation function could be created, error code otherwise.
  */
  public short createCondensation(String placementSR, String pubName, String variableName, short outDataType, int outUserType,
                                  int outTypeLen, short priority, int intervalLow, int intervalHigh, boolean inFilerHigh,
                                  int inFilterHighVar, boolean inFilerLow, int inFilterLowVar, boolean outFilerHigh, int outFilterHighVar,
                                  boolean outFilerLow, int outFilterLowVar, short triggerType, int triggerVar1, String calculatorURI,
                                  String calculatorClassName);

  /**
  * The <code>addSubscription</code> method is to add a subscription as a input variable for the condensation function to be created.
  * <BR>
  * @param subscriberName The name of the subscriber.
  * @param pubName The name of the publisher.
  * @param variableName The name of the variable to be subscribed to.
  * @param modes The modes that this subscription will be valid in.
  * @param dataType The datatype of the variable that we subscribe to.
  * @param userType The type of the user registered type.
  * @param interval The interval that we wish to receive the events.
  * @param priority The priority of the subscription.
  * @param latency The latency that we would link for this subscription.
  * @param redundancy The redundancy that we would link for this subscription.
  * @return Returns <code>true</code> if there is space to add this
  * subscription, <code>false</code> otherwise.
  */
  public boolean addSubscription(String subscriberName, String pubName, String variableName, int modes, short dataType, int userType,
                                 int interval, short priority, short latency, short redundancy);

  /**
  * The <code>removeCondensation</code> method is to unregister a previous created condensation function with the edgeStatusRouter.
  * <BR>
  * @param pubName The name of the publisher of the condensation variable.
  * @param variableName The name of the condensation variable.
  * @return Returns <code>CommConstants.COMMAND_OK</code> if the condensation function is removed, error code otherwise.
  */
  public short removeCondensation(String pubName, String variableName);

  /**
  * The <code>clearSubscriptions</code> method is used to clear all the subscriptions that have been adden in addSubscription.
  */
  public void clearSubscriptions();
}
```

Figure A.32: The condensation creator Java API provided by the framework, part 2 of 2.

```
package edu.wsu.gridstat.router.interfaces;

/**
 * <p>Title: OutInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// Standard java packages.
import java.net.*;

// GridStat packages.
import edu.wsu.gridstat.router.common.EventHolder;

public interface OutInterface
{
  /**
  * The <code>pushAlertEvent</code> method is used to send an alert event using this channel.
  * <BR>
  * @param event The event to be sent.
  * @return Returns <code>true</code> if the event will be sent, <code>false</code> otherwise i.e. it will be dropped.
  */
  public boolean pushAlertEvent(EventHolder event);

  /**
  * The <code>pushAlertEvent</code> method is used to send an alert event using this channel.
  * <BR>
  * @param event The event to be sent.
  * @param fromAdr The address where this alert came from, we don't want to
  * send it back again.
  * @return Returns <code>true</code> if the event will be sent, <code>false</code> otherwise i.e. it will be dropped.
  */
  public boolean pushAlertEvent(EventHolder event, SocketAddress fromAdr);

  /**
  * The <code>pushEvent</code> method is used to send a event using this channel.
  * <BR>
  * @param event The event to be sent.
  * @param priority The priority of the event.
  * @return Returns <code>true</code> if the event will be sent, <code>false</code> otherwise i.e. it will be dropped.
  */
  public boolean pushEvent(EventHolder event, int priority);

  /**
  * The <code>getLevel</code> method is used to get the level of this link, i.e. how far up the hierarchy have to be
  * traversed to find the parent.
  * <BR>
  * @return Returns the level of this link.
  */
  public int getLevel();
}
```

Figure A.33: The OutInterface used within a status router to communicate between different modules.

```
package edu.wsu.gridstat.router.interfaces;

/**
 * <p>Title: CondFunInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// Standard java packages.
import java.util.Vector;

// GridStat packages.
import edu.wsu.gridstat.router.common.EventHolder;
import edu.wsu.gridstat.router.condFunction.CondRoutingHolder;
import edu.wsu.gridstat.router.condFunction.CondStatusHolder;

public abstract interface CondFunInterface extends OutInterface
{
  /**
  * The <code>start</code> method is used to start this condensation function.
  * <BR>
  * @return Returns <code>true</code> if we can load the calculator object and start the triggering thread, <code>false</code> otherwise.
  */
  public boolean start();

  /**
  * The <code>stop</code> method is used to stop the thread.
  */
  public void stop();

  /**
  * The <code>addRoutingHolder</code> method is used to add a routing holder to this condensation function. This is a holder for one
  * of the subscription that is condensation function has.
  * <BR>
  * @param holder The routing holder that is added.
  */
  public void addRoutingHolder(CondRoutingHolder holder);

  /**
  * The <code>getRoutingHolders</code> method is used to get all the routing holders that this condensation function has.
  * <BR>
  * @return Returns a Vector of <code>CondRoutingHolder</code>.
  */
  public Vector getRoutingHolders();

  /**
  * The <code>doCalculation</code> method is used to perform the calculation of a new value and publish it as a status event.
  * <BR>
  * @return Returns <code>true</code> if we could calculate a new value and publish it as a status event, <code>false</code> otherwise.
  * @throws java.lang.InterruptedException
  */
  public boolean doCalculation() throws InterruptedException;

  /**
  * The <code>setValue</code> method is used to exctract the value of an event that we have received.
  * <BR>
  * @param statusHolder The holder where we will set the value.
  * @param event The event that we will extract the value from.
  * @return Returns <code>true</code> if the value is exctracted and set, <code>false</code> if the value is filtered.
  */
  public boolean setValue(CondStatusHolder statusHolder, EventHolder event);
}
```

Figure A.34: Base interface for the condensation function mechanism.

```
package edu.wsu.gridstat.router.condFunction;

/**
 * <p>Title: CondCalcInterface </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */
// GridStat packages.
import edu.wsu.gridstat.util.IntHashMap;
import edu.wsu.gridstat.command._CondensationSetupHolder;

public abstract interface CondCalcInterface
{
  /**
  * The <code>initialize</code> method is used to initialize this condensation calculator.
  * <BR>
  * @param setUpHolder The command that is used to set up this condensation   * function.
  * @param statusHolders The holder for the status events that will be used in the calculation.
  */
  public void initialize(_CondensationSetupHolder setUpHolder, IntHashMap statusHolders);

  /**
  * The <code>filterOutValue</code> method is used to test if the value should be filtered.
  * <BR>
  * @param value The value that is tested for filtering.
  * @return Returns <code>true</code> if the value should be filtered, <code>false</code> otherwise.
  */
  public boolean filterOutValue(int value);

  /**
  * The <code>filterOutValue</code> method is used to test if the value should be filtered.
  * <BR>
  * @param value The value that is tested for filtering.
  * @return Returns <code>true</code> if the value should be filtered, <code>false</code> otherwise.
  */
  public boolean filterOutValue(float value);

  /**
  * The <code>filterOutValue</code> method is used to test if the value should be filtered.
  * <BR>
  * @param value The value that is tested for filtering.
  * @return Returns <code>true</code> if the value should be filtered, <code>false</code> otherwise.
  */
  public boolean filterOutValue(boolean value);

  /**
  * The <code>filterInValue</code> method is used to test if the value should be filtered.
  * <BR>
  * @param value The value that is tested for filtering.
  * @return Returns <code>true</code> if the value should be filtered, <code>false</code> otherwise.
  */
  public boolean filterInValue(int value);

  /**
  * The <code>filterInValue</code> method is used to test if the value should be filtered.
  * <BR>
  * @param value The value that is tested for filtering.
  * @return Returns <code>true</code> if the value should be filtered, <code>false</code> otherwise.
  */
  public boolean filterInValue(float value);

  /**
  * The <code>filterInValue</code> method is used to test if the value should be filtered.
  * <BR>
  * @param value The value that is tested for filtering.
  * @return Returns <code>true</code> if the value should be filtered, <code>false</code> otherwise.
  */
  public boolean filterInValue(boolean value);

  /**
  * The <code>filterOutValue</code> method is used to test if the value should be filtered.
  * <BR>
  * @param value The value that is tested for filtering.
  * @return Returns <code>true</code> if the value should be filtered, <code>false</code> otherwise.
  */
  public abstract boolean filterOutValue(byte[] value);

  /**
  * The <code>filterInValue</code> method is used to test if the value should be filtered.
  * <BR>
  * @param value The value that is tested for filtering.
  * @return Returns <code>true</code> if the value should be filtered, <code>false</code> otherwise.
  */
  public abstract boolean filterInValue(byte[] value);

  /**
  * The <code>calculate</code> abstract method is used to do the calculation.
  * <BR>
  * @return Returns <code>true</code> if we produced a new value, <code>false</code> if no new value was produced.
  */
  public abstract boolean calculate();
}
```

Figure A.35: Base interface for the calculator of the condensation function mechanism.

```
package edu.wsu.gridstat.qosBroker.common;

/**
 * <p>Title: QoSBrokerBaseInterface </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

public abstract interface QoSBrokerBaseInterface
{
  /**
  * The <code>start</code> method is used bootstrap the (Leaf) QoS Broker.
  * <BR>
  * @return Returns <code>true</code> if it is started <code>false</code> otherwise.
  */
  public abstract boolean start();

  /**
  * The <code>stop</code> method is used stop the (Leaf) QoS Broker, will stop all the threads and kill the objects.
  * <BR>
  * @return Returns <code>true</code> if it is stopped <code>false</code> otherwise.
  */
  public abstract boolean stop();

  /**
  * The <code>isRunning</code> method is used to find out if this (Leaf) QoS Broker is
  * running or not.
  * <BR>
  * @return Returns <code>true</code> if it is running <code>false</code> otherwise.
  */
  public abstract boolean isRunning();

  /**
  * The <code>setFlooding</code> method is used to set a variable to be flooded.
  * <BR>
  * @param pubName The name of the publisher of the variable.
  * @param variableName The name of the variable.
  * @param modes For which modes whould this variable be flooded.
  * @throws InterruptedException If during the execution of the command the thread dies.
  * @return Returns <code>true</code> if the variable is set to be flooded, <code>false</code> otherwise.
  */
  public abstract boolean setFlooding(String pubName, String variableName, int modes) throws InterruptedException;

  /**
  * The <code>removeFlooding</code> method is used to remove a variable that is flooded.
  * <BR>
  * @param pubName The name of the publisher of the variable.
  * @param variableName The name of the variable.
  * @param modes For which modes is this variable flooded.
  * @throws InterruptedException If during the execution of the command the thread dies.
  * @return Returns <code>true</code> if the variable is not flooded anymore, <code>false</code> otherwise.
  */
  public abstract boolean removeFlooding(String pubName, String variableName, int modes) throws InterruptedException;

  /**
  * The <code>changeMode</code> method is used to change the mode.
  * <BR>
  * @param mode The new mode.
  * @throws InterruptedException If during the execution of the command the thread dies.
  * @return Returns <code>true</code> if the mode is changed, <code>false</code> otherwise.
  */
  public abstract boolean changeMode(int mode) throws InterruptedException;

  /**
  * The <code>getMode</code> method is used return the current mode.
  * <BR>
  * @throws InterruptedException If during the execution of the command the thread dies.
  * @return Returns the current mode.
  */
  public abstract int getMode() throws InterruptedException;
}
```

Figure A.36: The common Java API provided by the (leaf) QoS broker by the framework.

```java
package edu.wsu.gridstat.qosBroker.leafQoSBroker;

/**
 * <p>Title: LeafQoSBrokerInterface </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// GridStat packages.
import edu.wsu.gridstat.qosBroker.common.QoSBrokerBaseInterface;

public interface LeafQoSBrokerInterface extends QoSBrokerBaseInterface
{
  /**
  * The <code>addEventChannel</code> method is used to add an EventChannel at runtime.
  * <BR>
  * @param srcName The name of the SR which is the source of the EventChannel.
  * @param srcAdr The ip address of the SR which is the source of the EventChannel.
  * @param srcPort The port of the SR which is the source of the EventChannel.
  * @param dstSRName The name of the SR which is the destination of the EventChannel.
  * @param dstAdr The ip address of the SR which is the destination of the EventChannel.
  * @param dstPort The port of the SR which is the destination of the EventChannel.
  * @param bandwidth The bandwidth of the EventChannel.
  * @param latency The latency of the the EventChannel.
  * @return Returns <code>true</code> if the EventChannel is added, <code>false</code> otherwise.
  */
  public boolean addEventChannel(String srcName, String srcAdr, int srcPort, String dstName, String dstAdr,
                                 int dstPort, short bandwidth, int latency);

  /**
  * The <code>removeEventChannel</code> method is used to remove an EventChannel at runtime.
  * <BR>
  * @param srcName The name of the SR which is the source of the EventChannel.
  * @param dstSRName The name of the SR which is the destination of the EventChannel.
  * @return Returns <code>true</code> if the EventChannel is removed, <code>false</code> otherwise.
  */
  public boolean removeEventChannel(String srcName, String dstName);

  /**
  * The <code>addStatusRouter</code> method is used to add an StatusRouter at runtime.
  * <BR>
  * @param name The name of the SR.
  * @param type The type: SR, edgeSR, foreignSR.
  * @param coordinateX The x coordinate.
  * @param coordinateY The y coordinate.
  * @param msgPerSec How many events can be routed per sec.
  * @return Returns <code>true</code> if the SR is added, <code>false</code> otherwise.
  */
  public boolean addStatusRouter(String name, short type, int coordinateX, int coordinateY, int msgPerSec);

  /**
  * The <code>removeStatusRouter</code> method is used to remove an StatusRouter at runtime.
  * <BR>
  * @param name The name of the SR.
  * @return Returns <code>true</code> if the SR is removed, <code>false</code> otherwise.
  */
  public boolean removeStatusRouter(String name);
}
```

Figure A.37: The leaf QoS broker Java API provided by the framework.

```java
package edu.wsu.gridstat.qosBroker.qosBroker;

/**
 * <p>Title: QoSBrokerInterface </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Washington State University </p>
 * @author Kjell Harald Gjermundrod
 * @version 1.0
 */

// GridStat packages.
import edu.wsu.gridstat.qosBroker.common.QoSBrokerBaseInterface;

public interface QoSBrokerInterface extends QoSBrokerBaseInterface
{
  /**
  * The <code>removeInterEdge</code> method is to remove a edge, if there is no
  * subscription currently on it.
  * <BR>
  * @param scrVertex The vertex that it the source of the edge.
  * @param dstVertex The vertex that it the destination of the edge.
  * @return Returns
  */
  public boolean removeInterEdge(String scrVertex, String dstVertex);

  /**
  * The <code>addLeafChild</code> method is to add a leaf child, the once that
  * manages a cloud.
  * <BR>
  * @param name The name of the child to be added.
  * @return Returns
  */
  public boolean addLeafChild(String name);

  /**
  * The <code>addInternalChild</code> method is to add a internal child.
  * <BR>
  * @param name The name of the child to be added.
  * @return Returns
  */
  public boolean addInternalChild(String name);

  /**
  * The <code>removeChild</code> method is to remove a child.
  * <BR>
  * @param name The name of this vertex.
  * @return Returns
  */
  public boolean removeChild(String name);
}
```

Figure A.38: The QoS broker Java API provided by the framework.

APPENDIX B

TERMINOLOGY

This appendix presents a terminology of the terms that are defined in this dissertation.

GridStat: A Status Dissemination Middleware framework for critical infrastructures. The framework takes advantage of the semantic of status variables to provide event streams from the producers to the consumers at the specified QoS.

Management plane: The plane of the GridStat framework that manages the resources in the communication infrastructure.

Data plane: The plane of the GridStat framework that consists of the resources that forward status event flows.

Cloud: A sub-part of the data plane that belong to one administration domain.

Leaf QoS broker (LQB): The leafs in the management hierarchy that controls the resources in a cloud.

QoS broker (QB): The none leaf in the management hierarchy that controls the resources that are its descendants.

Status router (SR): A middleware router that forwards event flows, by taking advantage of the semantic of status variables.

Edge status router (ESR): Same as a SR with the added functionality of acting as a proxy for the management plane to publishers, subscribers, and condensation creators.

Border status router (BSR): Same as a SR, but it has communication links to other BSR in other clouds.

Publisher: A producer of event streams from registered status variables.

Subscriber: A consumer of requested event streams with specified QoS.

Condensation creator: A GUI based tool to specify condensation functions.

Event channel (EC): A communication link between (E—B)SR, for status events.

Status variable: A variable that represent some kind of object.

Status event: A time-stamped value of the state of a status variable at the time.

Publication: A registered status variable that events will be published from at a specified interval.

Subscription: A registered variable that events is to be received from at a requested interval and with the requested QoS.

Inter-cloud subscription: A subscription that the events have to traverse multiple clouds.

Subscription path: The path that is allocated from the publisher to the subscriber for a specific subscription subject to the requested QoS.

Status event stream: Same as a subscription path.

Experiment: A experiment consists of one or more experiment topologies. The purpose an experiment is to validate a mechanism or feature of the GridStat framework.

Experiment topology: A topology is how the Status Routers are organized as a network, and where the publishers and subscribers connect to the GridStat network. For each experiment topologies multiple runs are preformed. Sometimes a experiment topology may also be referred to as a topology.

Experiment run: An experiment run is the number of status variables that are published by the different publishers in the experiment topology, and also the number of variables subscribed to be the various subscribers and at what interval. Sometimes a experiment run may also be referred to as a run.

Load variable: A variable that is used to incur load on some of the resources during an experiment.

Total load variables (tlv): The total number of load variables that are subscribed to during an experiment.

Reference variable: A variable that is used to take measurements from during an experiment.

Total reference variables (trv): The total number of reference variables that are subscribed to during an experiment.

Load variable interval: The interval that a set of load variables are published at during an experiment.

Reference variable interval: The interval that a set of reference variables are published at during an experiment.

Load publisher: A publisher that publishes load variables during an experiment.

Load subscriber: A subscriber that subscribes to load variables during an experiment.

Reference publisher: A publisher that publishes reference variable(s) during an experiment.

Reference subscriber: A subscriber that subscribes to reference variable(s) during an experiment.

Load system: A load publisher and a load subscriber along with the edge status routers that they are connected to.

Reference system: A reference publisher and a reference subscriber along with the edge status routers that they are connected to.

Multicast rate-filtering routing mechanism: A mechanism in the data plane that deterministically filters and multicasts the event streams.

Limited flooding mechanism: A mechanism in the data plane (initiated by the management plane) that can flood a variable within a number of clouds.

Operating modes mechanism: A mechanism in the data plane (initiated by the management plane) that can change between pre-allocated subscription sets without violating the QoS for any of the subscriptions.

Condensation function mechanism: A mechanism to move application logic into the data plane, the user specifies the functionality and the management plane places the condensation function into the data plane.

APPENDIX C

THE CONSTANT *K* VALUE

This appendix presents how the value *k* was derived that are used in the experiments chapter.

The results from the hop scalability experiments (see Section 8.4) was used to find the *k* value. The Function 8.3 was used to find the *k* value when all the other variables was known. So the function became:

$$k = \frac{er}{\frac{Mean_{delay}(ev)-0.419)}{s} - 0.227} \tag{C.1}$$

when $s > 1$ and

$$k = \frac{er}{\frac{Mean_{delay}(ev)-0.419)}{s}} \tag{C.2}$$

when *s* is 1 and

By filling in the *er*, $Mean_{delay}(ev)$, and *s* (number of hops on the path) and taking the average out of all the 28 runs that were performed for this experiment it yielded 209.16, which was rounded up to 210. This was how the *k* value was derived for the experiment chapter.

BIBLIOGRAPHY

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[2] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[3] GridWise Alliance. Gridwise alliance declaration. http://www.gridwise.org/GridWiseAlliance.pdf, 2005.

[4] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, Dallas, TX, May 2000.

[5] D. Bakken. Middleware. Available from: http://www.eecs.wsu.edu/ bakken/middleware.pdf, 2003.

[6] D. Bakken, A. Bose, C. Hauser, I. Dionysiou, H. Gjermundrød, L. Xu, and S. Bhowmik. Towards for extensible and resilient real-time information dissemination for the electric power grid. In *Proceedings of Power Systems and Communications Systems for the Future, International Institute for Critical Infrastructures*, Beijing, China, September 2002.

[7] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, and W. Tao. Information flow based event distribution middleware. In *ICDCS Workshop on Electronic Commerce and Web-based Applications/Middleware*, pages 114–121, Austin, TX, May/June 1999.

[8] Steve Benford, Neil Crout, John Crowe, Stefan Egglestone, Malcom Foster, Chris Greenhalgh, Alastair Hampshire, Barrie Hayes-Gill, Jan Humble, Alex Irune, Johanna Laybourn-Parry, Ben Palethorpe, Timothy Reid, and Mark Sumner. e-science from the antarctic to the grid. In *Proceedings of the 2nd UK e-Science All Hands Meeting*, 2003.

[9] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure call. *ACM Transactions on Computing Systems*, 2(1):39–59, 1984.

[10] Anjan Bose. Power system stability: New opportunities for control. In Derong Liu and Panos J. Antsaklis, editors, *Stability and Control of Dynamical Systems and Applications*. Birkhauser (Boston), 2003. Also from http://gridstat.eecs.wsu.edu/Bose-GridComms-Overview-Chapter.pdf.

[11] Paul Brett, Rob Knauerhase, Mic Bowman, Robert Adams, Aroon Nataraj, Jeff Sedayao, and Michael Spindel. A shared global event propagation system to enable next generation distributed services. In *WORLDS '04*, San Francisco, California, December 2004.

[12] R. Bulirsch and J. Stoer. *Introduction to Numerical Analysis*, chapter 2.2. Springer, New York, NY, 3rd edition, 2002.

[13] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, OR, July 2000.

[14] Rapha El Chand. Xnet: A reliable content-based publish/subscribe system. In *23rd Symposium on Reliable Distributed Systems*, pages 264–273, Florianopolis, Brazil, October 2004.

[15] Guanling Chen and David Kotz. Context aggregation and dissemination in ubiquitous computing systems. In *Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 105–114, Callicoon, NY, June 2002.

[16] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *SIGMOD Conference*, pages 379–390, Dallas, TX, May 2000.

[17] Brent Chun, Joseph M. Hellerstein, Ryan Huebsch, Petros Maniatis, and Timothy Roscoe. Design considerations for information planes. In *WORLDS '04*, San Francisco, California, December 2004.

[18] David D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM*, pages 106–114, Stanford, CA, August 1988.

[19] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *In Proceedings of the 10th International Conference on Extending Database Technology*, Munich, Germany, March 2006.

[20] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proceedings of the 30th VLDB Conference*, pages 612–623, Toronto, Canada, August 2004.

[21] Ioanna Dionysiou, Kjell Harald Gjermundrød, and David Bakken. Fault tolerance issues in publish-subscribe status dissemination middleware for the electric power grid. In *Supplement of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, June 2002.

[22] Greg Eisenhauer, Fabian E. Bustamante, and Karsten Schwan. A middleware toolkit for client-initiated service specialization. *Operating Systems Review*, 35(2):7–20, 2001.

[23] EPRI. Smart power delivery – a vision for the future. Technical report, EPRI Newsletter Online, June 2003. Available: http://www.epri.com/journal/.

[24] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Mermarec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, July 2003.

[25] P.T. Eugster, R. Guerraoui, and J. Sventek. Type-based public/subscribe. Technical Report TR-DSC-2000-029, Swiss Federal Institute of Technology, Lausanne, June 2000.

[26] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):115–126, 2001.

[27] B. Fardanesh. Future trends in power systems control. *IEEE Computer Applications in Power*, 15(3), July 2002.

[28] Ludger Fiege, Mariano Cilia, Gero Mhl, and Alejandro Buchmann. Publish-subscribe grows up: Support for management, visibility control, and heterogeneity. *IEEE Internet Computing*, 10(1):48–55, January/February 2006.

[29] Trustworthy Cyber Infrastructure for the Power Grid. Trustworthy cyber infrastructure for the power grid (tcip). http://www.iti.uiuc.edu/tcip/index.html, 2006.

[30] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[31] Philip B. Gibbons, Brad Karp, Yan Ke, Suman Nath, and Srinivasan Seshan. Irisnet: An Architecture for a Worldwide Sensor Web. *Pervasive Computing*, 2(4):22–33, October–December 2003.

[32] Harald Gjermundrød, Ioanna Dionysiou, David Bakken, and Carl Hauser. Fault tolerance mechanisms in status dissemination middleware. In *Supplement of the 2003 International*

Conference on Dependable Systems and Networks (DSN-2003), San Francisco, CA, June 2003.

[33] Object Managment Group. Distributed simulation. Technical Report Version 2.0, Object Management Group, January 2001.

[34] Object Managment Group. Data distribution service for real-time systems specification. Technical Report Version 1.0, Object Management Group, December 2004.

[35] Object Managment Group. Notification service specification. Technical Report Version 1.1, Object Management Group, October 2004.

[36] A. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *In Proceedings of SIGMOD*, pages 419–430, San Diego, California, June 2003.

[37] Carl Hauser, David E. Bakken, and Anjan Bose. Failure to communicate: Next generation requirements, technologies, and architecture for the electric power grid. *IEEE Power and Energy Magazine*, March/April, to appear 2005.

[38] Carl H. Hauser, David E. Bakken, Ioanna Dionysiou, K. Harald Gjermundrød, Venkata S. Irava, and Anjan Bose. Security, trust, and qos in next-generation control and communication for large power systems. In *Proceedings of the Workshop on Complex Network and Infrastructure Protection (CNIP06)*, Rome, Italy, March 2006.

[39] Joel Helkey. Achieving end-to-end delay bounds in a real-time status dissemination network. Master's thesis, Washington State University, Pullman, WA, August 2006. Available from http://griffin.wsu.edu.

[40] Electricity Innovation Institute. Integrated energy and communications systems architecture (iecsa). http://www.iecsa.org/IECSASummaryLong.pdf.

[41] Venkata S. Irava. *Low-Cost Delay-Constrained Multicast Routing Heuristics and their Evaluation*. PhD thesis, Washington State University, Pullman, WA, August 2006. Available from http://griffin.wsu.edu.

[42] H. Jeffreys and B.S. Jeffreys. *Methods of Mathematical Physics*, chapter 9.011, page 260. Cambridge University Press, Cambridge, U.K., 3rd edition, 1988.

[43] Ping Jiang. A naming and directory service for publisher-subscriber's status dissemination. Master's thesis, Washington State University, Pullman, WA, May 2004. Available from http://griffin.wsu.edu.

[44] Ryan A. Johnston. Obtaining high performance phasor measurements in a geographically distributed status dissemination network. Master's thesis, Washington State University, Pullman, WA, May 2005. Available from http://griffin.wsu.edu.

[45] Ryan A. Johnston, Carl H. Hauser, K. Harald Gjermundrød, and David E. Bakken. Distributing time-synchronous phasor measurement data using the gridstat communication infrastructure. In *Proceedings of 39th Annual Hawaii International Conference on System Sciences*, Kauai, Hawaii, January 2006.

[46] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *Knowledge and Data Engineering*, 11(4):610–628, 1999.

[47] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, Madison, WI, June 2002.

[48] K. Mani, Brian E. Aydemir, Elliott Karpilovsky, and Dan Zimmerman. Event webs for crisis management. In *Proceedings 2nd IASTED International Conference on Communications, Internet, and Information Technology*, pages 715–720, Scottsdale, AZ, November 2003.

[49] Massimo Mecella, Monica Scannapieco, Antonino Virgillito, Roberto Baldoni, Tiziana Catarci, and Carlo Batini. The DaQuinCIS Broker: Querying Data and Their Quality in Cooperative Information Systems. *Journal of Data Semantics*, 1(LNCS 2800), 2003.

[50] Sun Microsystems. Java message service. Technical Report 011801, SUN, April 2002.

[51] Sun Microsystems. Jini network technology. http://wwws.sun.com/software/jini/, 2003.

[52] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *In Proc. of CIDR*, pages 245–256, Asilomar, California, January 2003.

[53] United State Department of Energy. National transmission grid study. http://certs.lbl.gov/NTGS/Reliability_ntgs.html, May 2002.

[54] Shrideep Pallickara, Marlon Pierce, Harshawardhan Gadgil, Geoffrey Fox, Yan Yan, and Yi Huang. A framework for secure end-to-end delivery of messages in publish/subscribe systems. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, page to appear, Barcelona, Spain, September 2006.

[55] Gerardo Pardo-Castellote. Omg data-distribution service: Architectural overview. In *Proceedings of the 23 rd International Conference on Distributed Computing Systems Workshops (ICDCSW03)*, pages 200–206, Providence, RI, May 2003.

[56] Lois Perrochon, Walter Mann, Stephane Kasriel, and David C. Luckham. Event mining with event processing networks. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 474–478, Beijing, China, April 1999.

[57] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In

Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS02), pages 611–618, Vienna, Austria, July 2002.

[58] Peter R. Pietzuch and Brian Shand. A framework for object-based event composition in distributed systems. Malaga, Spain, June 2002. Presented at the 12th PhDOOS Workshop (ECOOP'02).

[59] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A Framework for Event Composition in Distributed Systems. In *Proc. of the 4th ACM/IFIP/USENIX Int. Conf. on Middleware (Middleware '03)*, pages 62–82, Rio de Janeiro, Brazil, June 2003.

[60] Plale and Schwan. Dynamic Querying of Streaming Data with the dquob System. *IEEE Transactions in Parallel and Distributed Systems*, 14(4):422–432, April 2003.

[61] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, chapter 16.4, pages 718–725. Cambridge University Press, Cambridge, U.K., 2nd edition, 1992.

[62] QNX. qnx operating system. http://www.qnx.com, July 2006.

[63] R. E. Schantz. Quality of service. In P. Dasgupta and J. Urban, editors, *Encyclopedia of Distributed Computing*. Kluwer Academic Publishers, to appear 2003.

[64] Richard E. Schantz and Douglas C. Schmidt. Middleware for distributed systems: Evolving the common structure for network-centric applications. *Encyclopedia of Software Engineering*, page to appear, 2001.

[65] Streambase. When now means right now. http://www.streambase.com/, 2006.

[66] Talarian. Talarian: Everything you need to know about middleware. http://www.talarian.com/industry/middleware/whitepaper.pdf, 2000.

[67] RAPIDE Design Team. Rapide-1.0 pattern language reference manual. Technical report, Stanford University, July 1997.

[68] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD Conference*, pages 321–330, San Diego, CA, June 1992.

[69] TIBCO. Tib/rendezvous white paper. http://www.rv.tibco.com/, 1999.

[70] Robbert van Renesse, Kenneth Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[71] J. H. van 't Hag. Data-Centric to the Max — The SPLICE Architecture Experience. In *Proceedings of the 23 rd International Conference on Distributed Computing Systems Workshops (ICDCSW03)*, pages 207–212, Providence, RI, May 2003.

[72] Bapi Vinnakota, Paul Dormitzer, Mark Rosenbluth, and Sridhar Lakshmanamurthy. Scalable Intel® IXA and its Building Blocks for Networking Platforms. *Intel Technology Journal Communications Processing*, 7(4):107–121, November 2003.

[73] w3c. Extensible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml/, 2004.

[74] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *In Proc. of the 2nd Workshop on Hot Topics in Networks*, Cambridge, MA, Novermber 2002.

[75] Z. Xie, G. Manimaran, V. Vittal, A. G. Phadke, and V. Centero. An information architecture for future power systems and its reliability analysis. *IEEE Transactions on Power Systems*, 17(3):857–863, August 2002.

[76] Xyleme. Xml for content management exploiting the full value of enterprise information. http://www.xyleme.com/, 2006.

[77] Yong Yao and J. E. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *Sigmod Record*, 31(3), 2002.

[78] S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26(1):3–10, March 2003.

[79] J. Zinky, L. O'Brien, D. Bakken, V. Krishnaswamy, and M. Ahamad. PASS: A service for efficient large scale dissemination of time varying data using CORBA. In *International Conference on Distributed Computing Systems*, pages 496–506, Austin, TX, June 1999.

[80] John Zinky, Joseph Loyall, and Richard Shapiro. Runtime performance modeling and measurement of adaptive distributed object applications. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 755–772, Irvine, California, October 2002.

[81] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems (Special Issue on CORBA and the OMG)*, 3(1):55–73, April 1997.