RATATOSKR: WIDE-AREA ACTUATOR RPC OVER GRIDSTAT WITH TIMELINESS, REDUNDANCY, AND SAFETY

By

ERLEND SMØRGRAV VIDDAL

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

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2007

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of ERLEND SMØRGRAV VIDDAL find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

I would like to thank my advisor Dave Bakken for his advice and guidance throughout my studies at WSU, and for taking an active interest in the well-being of his students beyond professional obligations. I would also like to thank Carl Hauser and Min Sik Kim for taking the time to be on my committee, and especially Carl Hauser for his help with the work on my thesis. Further, I would like to tank all past and current members of the GridStat team for their great work, and for valuable discussion and contributions on my research.

A special thanks goes to my friends in Norway and in Pullman, and my family for their continuing support during my research, and for making my stay here much more enjoyable.

Finally, I would like to thank the organizations that have provided financial support for education and research. In particular, I have received a stipend from The Norwegian State Educational Loan Fund and tuition reduction from Washington State University. In addition, my research has been supported in part by grants CNS 05-24695 (CT-CS: Trustworthy Cyber Infrastructure for the Power Grid(TCIP)) and CCR-0326006 from the US National Science Foundation.

PUBLICATIONS

Erlend S. Viddal, Stian Abelsen, David Bakken and Carl Hauser, Ratatoskr: Wide-Area Actuator RPC over GridStat with Timeliness, Redundancy, and Safety, in DSN ’08: Proceedings of the International Conference on Dependable Systems and Networks (DSN’08). To be submitted in Fall 2007

RATATOSKR: WIDE-AREA ACTUATOR RPC OVER GRIDSTAT WITH TIMELINESS, REDUNDANCY, AND SAFETY

Abstract

by Erlend Smørgrav Viddal, M.S.
Washington State University
December 2007

Chair: David E. Bakken

The development of the communication infrastructure for the north-American electrical power grid has failed to fully incorporate important developments in the field of computer science, affecting the stability and efficiency of the power grid as a whole. The current power-grid communication standard, SCADA, utilizes protocols specialized for centralized communication, hampering communication between field sites key for envisioned improvements of power grid safety and efficiency. Further, a number of different proprietary communication protocols are in use, making communication between power utility companies very difficult.

GridStat is a communication infrastructure designed for a power grid environment that solves many of the problems with the current situation. GridStat uses a specialization of the publish-subscribe middleware paradigm, status dissemination, that takes advantage of the semantics of status data to provide flexible acquisition of power-grid data with multiple dimensions of QoS semantics. The middleware approach enables communication between utilities independent of proprietary network protocols, and allows enhanced network features such as forwarding data through multiple redundant paths. While GridStat provides excellent support for data acquisition, the publish-subscribe architecture supports only one-way communication and provides syntax and semantics unsuitable for control communications.

This thesis presents Ratatoskr, a novel scheme for control of actuators using GridStat

communication. It constructs a two-way communication channel on top of GridStat publish/subscribe paths, and utilizes the QoS semantics and middleware properties GridStat provides. For control communication Ratatoskr uses remote procedure call (RPC), providing programmer friendliness and familiarity. The QoS semantics of GridStat are drawn upon to provide the timeliness required for power-grid operation. Reliability concerns are addressed by providing three redundancy schemes, ACK/resend, transmitting multiple copies of a single packet, and spatial redundancy through GridStat's redundant routing paths feature. Additionally, pre- and post-condition expressions over GridStat status variables are built into call semantics. The architecture and design of Ratatoskr is presented, along with results from an evaluation of a prototype implementation.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER ONE

INTRODUCTION

The North-American electric power-grid is among the largest and most complex systems created by man. Its critical mission of balancing changing demand and generation of power involves coordinating diverse sets of components over a very large areas, and in a large number of utility-domains. This balancing process requires extensive communication between components in the Grid for monitoring system state and controlling actuator devices. The development of the grid communication infrastructure has failed to incorporate important developments in the field of computer science, affecting the stability and efficiency of the power grid as a whole, [2]. GridStat is a communication infrastructure designed for a power grid environment that would solve many of the problems with the current situation, but it does not conveniently control communication, [8]. This thesis proposes a novel scheme for control of actuators using GridStat communication.

1.1 Current Power Grid Communication Infrastructure

In the 1960s, utilities started shifting from mainly using field personnel and telephone communication for control of the power grid to electronic schemes. Today the predominant Grid communication architecture is SCADA (Supervisory Control and Data Acquisition). The SCADA architecture has not changed notably from its origins. It is a centralized approach, in which a manned regional control center gathers data from and issues control signals to devices in geographically dispersed field sites. Early systems were developed without any official standards, resulting in numerous proprietary protocols. SCADA systems have since developed incrementally, and often incorporate a blend of new and old communication technology. Topologies are predominately varieties of star-shapes, and protocols are mostly designed solely for communication between control center and field sites, [12].

With increasing stress on the transmission network, distribution models growing more complex and looming threats of terrorism and cyber security risks, there is a pressing need for better monitoring of grid dynamics and improved control schemes, [2]. The inherent inflexibility of the SCADA architecture is unable to accommodate this. Communication between utilities is mostly done by telephone between operators, making observation and containment of grid-wide phenomena such as rolling blackouts very difficult. Fast automated control schemes involving substation to substation communication have yet to be standardized, and are implemented using expensive, specialized point-to-point links, [2]. The Intelligrid project, a vision of a future power grid created by an international consortium of power researchers, industry representatives, equipment manufacturers and government representatives, argues for several applications of communication substation to substation, substation to field equipment, and field equipment to field equipment, yet it does not propose a wide-area communication mechansism, [6]. IEC 61850 is a widely accepted standard for substation automation that includes standardized self description of devices independent of brand and an event-driven communication model, [17]. While IEC 61850 holds great potential for improved substation control, it does not specify a wide-area network mechanism in itself. Continued incremental development of the existing centralized and inflexible communication structure will severely inhibit potential growth in power-grid efficiency and stability.

1.2 GridStat

Gridstat is a framework for power-grid communication centered around a middleware network for power-grid data acquisition, [8]. It provides a flexible communication scheme with the reliability and timeliness required in a power-grid network. GridStat routes traffic on top of existing communications infrastructure through a series of application-layer routers, overcoming the inherent heterogeneity of legacy networking technology. The unifying middleware framework creates a

flexible overlay topology on top of the centralized designs of existing power-grid network infrastructures, allows for easy interoperability between power utility companies despite use of proprietary transport protocols and offers abstractions to network services, in addition to several other features well suited for a power-grid infrastructure that are less relevant in context of this thesis.

GridStat follows the publish-subscribe (pub-sub) paradigm. A device can publish status information either directly to the GridStat framework, or through an intermediary middleware publisher module, possibly located on another computer. The GridStat framework makes the information available as one or more *status variables*, published values that are regularly updated . Applications may retrieve status updates by subscribing to status variables through a GridStat subscriber interface. The GridStat framework forwards status updates from the publisher through the application-layer routers and finally to the subscriber. This overlay-network scheme allows GridStat to offer a wide range of network features independent of the underlying technology. The most important of these are multicast and redundant forwarding paths (for fault-tolerance). In addition to offering functionality additional to that provided by the underlying network, GridStat improves the network Quality of Service (QoS), the nonfunctional properties of the network. QoS enhancments provided by GridStat include bounded delay, reliability and security.

Currently GridStat forwards status updates in a one-way, pub-sub fashion, addressing the data acquisition needs of a grid operations infrastructure. While it would be possible to forward control commands using the existing status update mechanism, such communication would be cumbersome with the pub-sub interface and in many cases would require operation success feedback which is impossible over the one-way paths. Use of SCADA protocols for control while restricting use of GridStat to data acquisition would require modifying inflexible proprietary legacy code for each new control operation introduced, and would not be able to utilize the flexible topology and interoperability introduced with GridStat. Use of other existing QoS-enabled control schemes would require implementing an overlay transport protocol to allow interoperability and flexible

topologies, which is redundant when GridStat already provides middleware routing. Further, existing solutions would not be designed with the capabilities already found in GridStat in mind, and mechanisms exploiting these would have to reside in the application layer voiding any advantages that could be achieved by designing use of these features into the control semantics.

1.3 Ratatoskr

This thesis proposes a power grid control scheme, Ratatoskr[1], using GridStat publications and subscriptions for communication. Ratatoskr is designed primarily for control of field sites from a control center, but use between field sites is imaginable. Remote Procedure Call (RPC) semantics are used because of its programmer friendliness and familiarity. Some of the traditional RPC features, especially transparency towards local procedure calls, are downplayed to better support the reliability and timeliness aspects required of a power grid control scheme. Reliability concerns are addressed by providing three redundancy schemes, ACK/resend, transmitting multiple copies of a single packet, and spatial redundancy through GridStat's redundant routing paths feature. ACK/resend represents a tradeoff between the timeliness and the reliability of the call, and multiple resends and redundant paths trades off reliability for network resources. Since the desired tradeoff parameters might vary between applications, Ratatoskr exposes these parameters to the programmer, along with other QoS properties of the call.

Further, Ratatoskr allows pre- and post-conditions, which are predicate expressions, to be placed on the procedure calls. *Pre-conditions* are evaluated before the execution of a call, and will abort the call if the expression is not satisfied. *Post-conditions* are evaluated after the exectution of a call and the result returned back to the client application to indicate system state. Pre- and post-conditions in Ratatoskr may use status variables published to GridStat in the expressions, accommodating usage of data from remote locations. These predicates are built into the call semantics, providing standardized usage patterns, simplifying reuse and providing the option of

[1] In norse mythology, Ratatoskr is a squirrel running around the great life-tree Yddgarsil, carrying insults between mythological creatures living on the branches.

delayed execution of post-conditions. Pre-conditions are tested before a call is carried out on the server side, aborting execution if the expression fails. Calls may then verify a safe system state before potentially dangerous operations, such as avoiding re-energizing a line if manned maintenance is scheduled in a endpoint substation at the time. Post conditions are carried out on the server after a call has completed, possibly after a specified delay. This allows grid programmers to verify the effects of operations. Power grid field sites often contain various mechanical devices which affect each other in complex ways, and the outcome of an operation could be unexpected even if the operation itself was successful.

1.4 Contributions of Thesis

The research contributions of this thesis are:

- Design and implementation of a novel control scheme for an electical power grid environment where remote procedure calls are transported over a QoS enabled one-way publish subscribe middleware network (GridStat).
- Design and implementation of three distinctive techniques for redundancy, offering a tradeoff between worst-case deadline, use of network resources and resiliency towards a variety of network failure categories. Applications are allowed fine control of redundancy semantics.
- Design and implementation of pre- and post- conditions mechanisms designed into RPC semantics provides additional functionality over application-level implementation and allows for a standardized mechanism for control signals between utilities.
- An experimental evaluation quantifying the tradeoffs between the redundancy techniques and their performance.

1.5 Organization of Thesis

The rest of this thesis is organized as follows: Chapter 2 summarizes related work and gives an introduction to GridStat required for understanding the contributions of this work. An overview of the Ratatoskr RPC mechanism and its underlying transport protocol is found in chapter 3. Chapter 4 details the design of a prototype implementation. Chapter 5 presents the findings of an experimental evaluation of the prototype. Finally, chapter 6 provides a summary of future work and the conclusion.

CHAPTER TWO

BACKGROUND AND RELATED WORK

This chapter gives an overview of relevant technologies, an overview of the GridStat framework architecture and details on the GridStat design related to the Ratatoskr mechanism. A more detailed introduction to GridStat can be found in [8].

2.1 Middleware

Distributed computing involves processes on separate machines cooperating, commonly over a network. If there are differences in the runtime environments of the interacting processes, such as data representation, some sort of translation must be performed between processes to ensure correct interaction. Middleware is software layered between the OS and the application offering abstractions to inter-process interactions and providing any needed translation services between process environments. Many different types of middleware interaction styles exist, accommodating a wide range of distributed system architectures.

2.2 Remote Procedure Call

Remote Procedure Call (RPC), first presented in [4], is a style of middleware providing abstractions for remote execution of code in a client-server fashion. Client applications call remote procedures through an interface similar in syntax to local procedures at the client, and the RPC mechanism handles packing the call with parameters and sending it over the network, executing the code corresponding to the call at the server, and transmitting the result back to the client application. Remote procedure calls allow for return values in spite of the traditional sense of procedure as a returnless call. RPC calls are in nature synchronous and blocking. A frequent design goal in RPC systems has been to make remote calls indistinguishable from local calls both in syntax and semantics, although the latter has been shown to be impossible, [30].

2.2.1 Failure Semantics

Opposed to local procedures, a remote procedure call may fail during remote operation while the local client process remains operating correctly. Such failures could stem from errors during network transfer or failure during server execution. The failure semantics of an RPC mechanism is defined by the way remote failures are handled and the guarantees of successful execution provided to the client application. As any network in practice can be made reliable by resending messages until an acknowledgment (ACK) is received, there are mainly three schools of thought for failure semantics, [28]:

- *At-least once* - Provides guarantee that an RPC procedure is successfully executed given eventually reliable communication, but allows for repeated executions of the same call. This may be achieved by having the client repeatedly send a call until a result is received. The server executes all calls, no matter if they have been executed before, and sends results upon successful execution. This provides a strong guarantee, but at-least once is only practical for idempotent procedures.

- *At-most once* - Provides guarantee that execution of an RPC procedure is attempted exactly once at server given eventually reliable communication, but does not guarantee that the attempted execution is successful. A client retries sending a call until it receives a response from the server. To ensure that the call is attempted at most once redundant calls are filtered at the server, possibly using logs in stable storage to retain filtering after server crash. The server must respond negatively to filtered calls so the client knows when to stop sending. When the client receives a negative response, the execution status of the call is uncertain.

- *Exactly once* - Provides a guarantee that the RPC is executed exactly once at the server, and so is the ideal case. This is impossible in the general RPC paradigm, as the RPC mechanism is active only before and after application-level execution of a call on the server, and thus cannot infer about the success of execution if server fails between these, [29]. This can in

some cases be resolved through cooperation with the overlying application, but this must be at the expense of programmability, mechanism complexity and frequent writes to stable storage, and is seldom used in practice.

While the beforementioned paradigms ideally rely on an eventually reliable network, it is often not practical to resend messages for an infinite number of times until success. The solution is most often to utilize no-loss transport protocols, that is transport protocols performing sends using ACK/retry schemes and that report back the delivery status of the send. While this type of transport protocol gives a high probability of delivery even over a faulty network, the overhead and high duration bound of such sends has given rise to a subdivision of at-most once semantics. *Maybe once* semantics provide zero-or-once execution semantics, but distinguishes from regular at-most once in that the underlying network sends do not ACK and so does not resend. This best-effort communication scheme provides a lower bound for end-to-end calltimes, and has little overhead, but at the cost of low reliability compared to regular at-least-once.

2.2.2 CORBA

Common Object Request Broker Architecture (CORBA) is a comprehensive standard for inter-operability between distributed object frameworks, [9]. Distributed objects are processes offering remote execution that are treated as abstract objects to separate the remote execution interface from the underlying implementation and platform. While CORBA is not strictly an RPC mechanism, the most common mechanism for making calls to distributed objects is so close to RPC in both syntax and semantics that it is relevant for this thesis. Many extensions to CORBA have been proposed, among them extensions targeting real-time operation, [11], and fault-tolerance[1], [10]. CORBA allows for the use of any underlying transport protocol, but dynamic configuration of communication protocols are not standardized and left to be specified by vendors, [24].

[1]It should be noted that Fault Tolerant CORBA focuses on fault tolerance through replication of services, while Ratatoskr focuses on replication of communication.

2.2.2.1 Real-time CORBA

Real-time CORBA is an extension to CORBA for interoperability between frameworks accomodating real-time distributed systems. The extensions emphasize resource management in addition to the introduction of extensive call prioritization semantics including mapping to OS thread prioritization. Real-time CORBA supports setting transport protocol QoS properties upon object binding, [26]. This allows setting policies per invocation by rebinding for each invocation. Real-time CORBA is a mature standard with several field-tested implementations. For example, the TAO orb is being used for operation flight programs by the Boing corporation, [27]. Two strategies for using existing implementations of Real-time CORBA for actuator control in the power-grid would be to route Real-time CORBA traffic directly on top of utility networks, or to route Real-time CORBA traffic over a middleware layer that overcomes incompabilities.

An alternative to using Ratatoskr over a GridStat for actuator control is to employ real-time CORBA on top of QoS aware networking technologies, such as ATM or diffserv IP. Such a real-time CORBA approach would provide timely control messages. Further, network level fault-tolerance may be achieved by using multiple temporally redundant sends of each network packet. In addition to temporal redundancy, Ratatoskr uses the GridStat redundant paths feature to provide fault tolerance against network faults. In chapter 5, an evaluation of the performance of the fault-tolerance capabilities of Ratatoskr shows that redundant path routing provides fault tolerance against certain fault categories that affect all temporally redundant sends along a single path. We are not aware of any wide-area network technology providing routing with redunant paths.

While this thesis presents an RPC mechanism designed specifically for actuator control over a GridStat connection, an alternative approach would be to implement a transport protocol enabling Real-time CORBA to communicate over GridStat. Where Ratatoskr is a pure RPC system, CORBA provides the advantages of a distributed object architecture, and compability to a large set of existing third party software components. Since Real-time CORBA extends the complex

CORBA standard, it requires adherence to a set of standardized semantics. While some requirements are provided in [6] and [17], the desired functionality of a power-grid control system is still largely unmapped and could potentially gain from mehcanisms not compatible the CORBA standard. The more minimalistic Ratatoskr design allows for rapid experimentation with features such as pre- and post-conditions and fine grained QoS semantics. Further, the communication subsystem of Ratatoskr can easily be adapted to carry Real-time CORBA traffic instead of Ratatoskr RPC calls, if Real-time CORBA is deemed desirable for a grid deployment.

2.2.2.2 Fault-tolerance in CORBA

The distributed object paradigm architecture of CORBA lends itself well to service replication. As the distributed object interface is decoupled from the underlying implementation and environment, an object interface can be replicated into several implementations running in separate environments with minimum impact on observed behavior. Several CORBA implementations provide replicated objects, [23, 25, 20]. A replicated distributed object scheme, coupled with a real-time CORBA implementation, would provide timely delivery and fault-tolerance. Such a scheme would still have to rely on a the underlying network for network-level fault tolerance, and would not be able to reap the benefits of redundant path routing. Further, object replication has to rely on strong multicast guarantees for synchronization between replicas, which gives a high worst-case message rounds in face of communication failures and thus scales badly with geographical distance.

2.3 Publish/Subscribe

The Publish/Subscribe middleware architecture centers around producers of information (publishers) and information consumers (subscribers). Publishers make information events available to a middleware network, and subscribers can request that events be forwarded to them by the network. The network forwards only subscribed data and can often optimize delivery paths through multicast, conserving bandwidth, [3]. The information flow is one-way; subscribers make subscription

requests to the middleware network itself rather than the publishers, allowing a decoupling between data producers and consumers. Further, published events can be stored in the network until the subscribers are ready to consume them, allowing a decoupling between publishing time and delivery to the subscriber, [7].

2.3.1 Status Dissemination

Status dissemination is a specialization of the publish/subscribe paradigm where publishers maintain *status variables*, [8]. Status variables are published values of a given type that are updated by publishing *status events*. Status events are limited by a maximum rate, and these restrictions in publication rate and type allow for additional QoS semantics compared to publish-subscribe systems without such restrictions.

2.4 GridStat

This section presents an overview of GridStat's architecture, and details the design of modules relevant to Ratatoskr. The purpose of this overview is to provide a background for the rest of the thesis. A more complete introduction to gridstat can be found in [8] and [2].

2.4.1 Architecture

The GridStat architecture is separated into two main subsystems, the *data plane*, a middleware databus where status updates supplied by publishers are forwarded to subscribers, and the *management plane*, a set of servers that manages system resources and organizes subscriptions by receiving subscription requests from subscribers and configuring the data plane towards forwarding accordingly. GridStat uses two kinds of communication traffic: *Data traffic* is always forwarded through the data plane message bus; *control traffic* between GridStat entities can be sent over any middleware control mechanism. The current implementation of GridStat uses CORBA and Ratatoskr as control message mechanisms.

Forwarding in the data plane is perfomed by *status routers*, middleware routers placed throughout a wide area network. Status routers form an overlay network by forwarding status events from router to router. The status routers retain implementations of all protocols used in the wide area network, and may function as bridges between the parts of the network using different networking technologies or with separate addressing spaces. Network connections in the data plane (from publishers and subscribers to status routers and betweens status routers) are represented as *event channels* that contain abstractions of data forwarding properties required for resource managment. Each publisher and subscriber has event channels to one or more status routers.

Whereas the data plane has a flat organization, the managment plane consists of a hierarchy of servers called *QoS brokers*. QoS brokers in the lowest level of the hierarchy are *leaf QoS brokers*, and are the only QoS brokers that directly communicate with entities in the data plane. QoS brokers above the leaf level are called internal QoS brokers and act as the sole *parent QoS Broker* of one or more *child QoS brokers*. All QoS brokers have a parent, with the exception of the *root QoS broker*, and leaf QoS brokers do not have child QoS brokers. Each QoS broker is associated with a set of entities in the data plane, the QoS broker's *cloud*. The data plane is divided up so each status router belongs to the cloud of exactly one leaf QoS broker. Status routers that have event channels to the same publisher or subscriber must be in the same cloud, and publishers and subscribers belong to the same cloud as their status routers. The clouds of internal QoS brokers are defined as the union of the clouds of their children, and thus the cloud of the root broker is all entities in the data plane. Entities are named according to their relationship to the managment plane hierarchy. A GridStat element must have a name unique within the scope of its parent; its full name is the name within the scope with an added prefix of the parent's name. This hierarchy of clouds is meant to correspond to a natural organization of managment domains in the power grid, such as levels of geographical areas.

As the data plane provides bounded delay and other QoS guarantees for subscription data, additional subscriptions must not overload network resources. The managment plane administers

the use of resources in the data plane, and so handles subscription requests. Subscription requests are made by the subscriber to its leaf QoS broker. If both the publisher and the subscriber of a new subscription are within the leaf level QoS broker's cloud, the leaf-level QoS broker is responsible for verifying that the connection will not overload network resources and update the status routers with the new subscription. If the publisher and subscribers are in different leaf-level clouds, the subscription request is propagated up in the hierarchy to the first QoS broker that has both within its cloud.

Ratatoskr is build on top of GridStat subscription paths, and the most relevant GridStat modules in the context of this thesis are the publisher and the subscriber.

2.4.1.1 Publisher

A publisher is a GridStat entity in the form of a module residing in an application program for publishing data to a GridStat network. It retains two connections to each of its status routers, an event channel for forwarding published status updates, and a middleware control channel for control messages that the status router forwards to the managment plane. The application can announce a new published variable through the module interface by providing a string name as identifier, a type, and the rate at which it is published. There is currently no policing on the maximum and minimum rates of publish updates. The managment hierarchy returns a 32-bit integer for identifying the variable within the GridStat network, a *variableID*. The application may update the value of a status variable through the module interface by specifying the variableID and the new value. The types of variables provide semantics for subscribed events, in addition to additional functionality outside the context of this thesis. The current types are various primary types (integer, floating point, bool...) and a user defined type, which is treated as a simple byte array by GridStat. The user defined type contains semantics for division into further subtypes, defined by the application.

2.4.1.2 Subscriber

The subscriber is a GridStat entity module used by applications to subscribe to data published over the GridStat network by a publisher. Similar to the publisher, the subscriber also retains two channels to each of its status routers: An event channel for receiving subscribed updates and a control channel for subscribing or unsubscribing to status variables. To subscribe to a published status variable, the application passes the variable name, the name of the publisher, QoS parameters and a *SubscriptionHolder*, an object that stores the status value and is updated by the subscriber when it receives updated values from its status router. Applications can access the values directly through the SubscriptionHolder interface, or can specify a callback method that will be invoked when the SubscriptonHolder is updated. There are several implmentations of SubscriptionHolders corresponding to the types of status variables, and applications can provide additional implementations for added functionality, or for semantics supporting subtypes of user defined variables. GridStat allows subscribers to specify that subscription data should be sent over *redundant paths*. Subscriptions over redundant paths are sent through more than one path in the GridStat network, where, with the exceptions of Entry-point SRs, a status router or event channel present in one path is not present in any other paths.

CHAPTER THREE

THE RATATOSKR RPC MECHANISM

GridStat's mission is to provide a complete communication framework for the power-grid. In addition to the existing publish-subscribe functionality, a standardized control-mechanism is needed for allowing power-utilities to control field equipment through the GridStat infrastructure. Such a mechanism will have to accommodate timely execution and high fault tolerance due to the critical nature of Grid operation. Ratatoskr is an RPC mechanism designed to run on top of GridStat's publish-subscribe system, utilizing the QoS mechanisms provided by GridStat. Built into the RPC semantics are pre- and post-conditions on calls, intended for predicates over GridStat published variables. Ratatoskr's intended primary use is for control-center operators and mechanisms to send control-messages to actuators in substations, either directly accessing actuators or through an intermediary RPC server that can communicate with actuators through legacy APIs. This chapter gives an overview of the features of Ratatoskr.

3.1 Definition of terms

The parts of the text regarding the transport protocol uses terms as defined in [18]. Additional terms are defined below.

- *2WoPS transport protocol* - 2-Way over Publish Subscribe. Communication protocol defining two-way communication over two GridStat one-way subscription paths.
- *2WoPS peer* - An application connected to a GridStat framework that utilizes the 2WoPS protocol for two-way communication using a GridStat publisher for sending data and a GridStat subscriber for receiving data.
- *Ratatoskr peer* - A device connected to a GridStat framework that utilizes Ratatoskr RPC for communication.

- *Entry-point SR* - The GridStat status-router a publisher or subscriber connects to. When used in relation to a 2WoPS peer, the entry-point SR signifies the edge status-router used to connect both the publisher and subscriber of the 2WoPS peer. The current implementation of GridStat allows publishers and subscribers to connect only to a single status router, while the architecture allows for multiple connections. The rest of this thesis considers only the case of a single entry-point SR per publisher or subscriber, as the exact semantics of multiple entry-point SRs are still undefined.

- *TSDU* - Transport Services Data Unit, a chunk of data from an overlying application that is sent through a transport layer connection.

- *transport protocol control message* - Similar to a TSDU, but data is for control of the 2WoPS protocol, not for application use.

- *TPDU* - Transport Protocol Data Unit, a chunk of data from the transport layer that is sent over a network layer connection. In this context, GridStat pub/sub communication is seen as a network layer. A TPDU can be a TSDU with added transport layer headers, or data used exclusively for control information by the transport layer. Several TPDUs can duplicate the same TSDU, and a single TPDU can be spread over multiple TSDUs, although the latter is not implemented in the prototype (see section 6.2.3.1).

- *NSDU, NPDU* - Network Service Data Unit and Network Protocol Data Unit, similar to TSDU and TPDU but for the network layer (GridStat pub-sub). A NSDU is exactly the same data as a corresponding TPDU, but viewed in context of the network protocol layer. An NSDU with an added network-layer header is an NPDU.

3.2 Two-way Communication over a Publish-Subscribe Framework

GridStat is a publish-subscribe system. Publishers in the system make data in form of status updates available to the GridStat framework. Subscribers may request subscriptions to these variables, and GridStat will forward subscribed information from publishers to subscribers according to QoS properties specified at subscription time. Communication is strictly one-way; subscribers have no way of sending information to publishers. RPC communication requires a two-way communication as procedure calls often will return values to the client, and acknowledgments on successful calls are almost universally required even when the call has no return value. To allow for a two-way communication link to be established, Ratatoskr utilizes a transport protocol called the *2WoPS protocol* on top of GridStat. The 2WoPS protocol achieves two way communication by instantiating both a publisher and a subscriber behind a single interface. To set up a two-way data path, two 2WoPS peers each publish a data variable specific to the session, and subscribe to the other peer's corresponding variable. Data is sent over the connection by publishing a status update containing the data, and received by the other peer through the subscriber interface. The 2WoPS interface masks the publisher and subscriber behavior.

Using a layered approach to communication allows for other uses than Ratatoskr RPC traffic of the 2WoPS protocol. For example, the 2WoPS protocol was used for control communication between QoS Brokers in [1]. Figure 3.1 shows the relationship between the modules of Ratatoskr (light shade), the GridStat modules used by Ratatoskr (dark shade), and examples of potential other applications using GridStat or Ratatoskr modules (white). The example shows the architecture stack for a control center and a substation. The main intended use of Ratatoskr is illustrated by the control center control system using Ratatoskr RPC to execute control operations on an actuator in the substation. Other uses of the 2WoPS protocol may be to transport legacy control messages to actuators if the actuator API remains to be fully implemented for Ratatoskr. The publisher and subscriber used by the 2WoPS protocol may have other uses, such as sending sensor data from the

substation to the control center, or publishing reports of power-grid state aggregated in the control center to be used by protection schemes in the substation. Finally, while GridStat requires control of the underlying network resources, network technologies that manage resource use may reserve bandwidth for uses outside GridStat, such as transferring video feeds from surveillance cameras in the substation.

3.2.1 Properties of the 2WoPS Protocol

The 2WoPS protocol is designed specifically for the Ratatoskr RPC. While this does not block out other uses for two-way communication over GridStat, care should be taken in noting the properties of the protocol, as these differ from the most common transport protocols, TCP and UDP. Some suggested extensions to the protocol to enhance use for other applications can be found in section 6.2.3. This section gives a summary the main properties of the 2WoPS protocol.

- *Connection oriented* - This was a necessary design decision as the underlying GridStat communication is connection-oriented. The 2WoPS protocol interface provides method to open and close a connection.

- *Controlled-loss* - An adjustable ACK/resend scheme similar to the *k_XMIT* scheme found in [21]. A TSDU will be retransmitted up to *k* times, where *k* is a user specified number. No ACK status is sent by the server on the *k*-th resend. This reduces the deadline for the sending process by the time for sending the ACK, at the expense of knowledge of the delivery status. It should be noted that while a missing ACK suggests that the message was not delivered, it cannot guarantee a failed delivery, as the message might have arrived while the ACK was lost. Because delivery status is unclear, the overlying RPC mechanism must still wait for a return from the server. When *k* is set to 0 the scheme has uncontrolled-loss properties. The controlled loss scheme gives little indication of the success of a call, which might be impractical for non-RPC use, so a *no-loss* scheme is also provided.

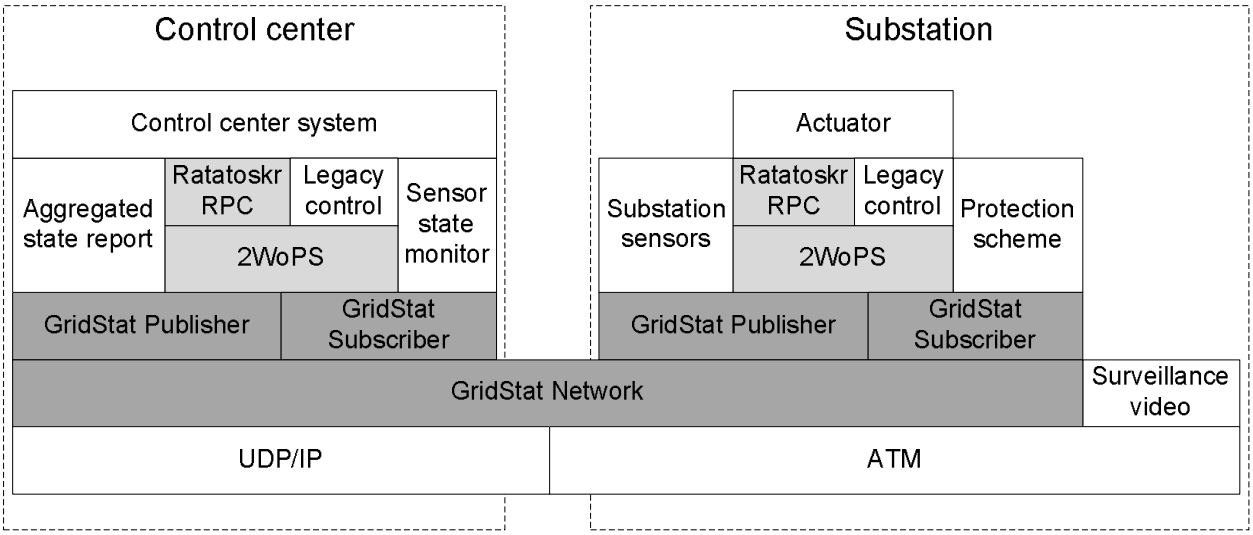


Figure 3.1: Ratatoskr Module Stack

- *No-loss* - adjustable ACK/resend scheme. Similarly to controlled-loss, TSDUs are retransmitted up to *k* times, only the no-loss scheme delivers an ACK even on the final send. For no-loss, delivery of a TSDU is uncertain only if each send attempt experiences faults, while for controlled-loss, delivery of a TSDU is uncertain even if the *k*-th send-attempt experiences no faults. This gives weaker failure semantics for controlled-loss, and more so at a low *k*. Controlled loss blocks $2 * k - 1$ trip-times per send and uses $2k - 1$ TSDU-transfers of bandwidth where no-loss blocks for $2 * k$ trip-times and uses $2k$ TSDU-transfers per send.

- *Timeliness* - GridStat provides delivery guarantees for subscriptions. The delivery guarantees of the underlying subscriptions are used to calculate tight timeout values for ack/resends, and delivery guarantees for TSDU sends.

- *Blocking* - Execution of a sending thread is blocked until the send is completed. A send is completed either when delivery is confirmed by receiving an ACK from the receiver, when the k-th ack times out for no-loss, or after the k-th send for controlled-loss. Multiple threads are still allowed to send in parallel.

- *Unordered delivery* - No message ordering is provided. Received NSDUs containing TSDUs are delivered to the application in the order they were delivered to the 2WoPS protocol by GridStat.

- *No duplicates* - TPDUs duplicating the same TSDU are filtered so the TSDU is delivered only once to the server application.

- *Error control* - A simple *cyclic redundancy check* (CRC) is used to discard TPDUs containing bit errors.

- *Hierarchical naming* - A naming scheme similar to the one for publishers and subscribers in GridStat is used. A 2WoPS peer is identified within its GridStat cloud by a string *s* with no spaces. The peer registers the publisher and subscriber used for communication with names based on this string, the publisher is named *s*PUB and the subscriber *s*SUB. The names of the publishers and subscribers must be *locally unique*, that is no other 2WoPS peer may have the same name within the leaf-level cloud of the entry-point, and since clouds have unique names the fully qualified name is globally unique. A leaf-QoS broker stores the names of all elements in its cloud and prevents registry of locally non-unique names.

- *Message oriented* - TSDUs are bounded by the maximum size of GridStat status updates, which is again bounded by an underlying transport protocol (UDP for the research prototype of GridStat).

3.2.2 Reliability Measures

A serious concern in any wide area network is that the number of components, geographical outstretch, and usage patterns of such networks inevitably lead to lowered reliability when compared to local area networks. This is especially apparent in the Internet, where most traffic uses the TCP transport protocol which uses TPDU drops to indicate congestion so it can regulate bandwidth usage. While GridStat controls network traffic at the network edges to avoid network overload

at least during normal operation, a GridStat deployment must be expected to share many of the loss properties of the internet stemming from other sources than traffic overload. These include hardware failure, maintenance, line damage or short-term miscommunication between routers. A 2002 study on an internet backbone found that with respect to mean failure rate, the median link failed every ten days, [16]. The mean failure had a duration of over one minute and 10% over 20 minutes. Such failure patterns are acceptable in the Internet because routing protocols will discover link errors and reconfigure routing to direct traffic around the affected links in a manner of seconds, and because few Internet applications depend on high network reliability. Also, while network drop rates during to transfer are negligible in the fiber and copper lines common in wide area networks today, GridStat is an overlay network and underlying physical network technologies might display other properties. Connecting remote substations to a utility network by fibre is expensive, and alternatives include microwave signaling, WiFi, power-line communications or satellite, all suffering from various forms of signal interference. The 2WoPS protocol provides several kinds of redundancy to overcome network failures.

3.2.2.1 Reliability Techniques in the 2WoPS Protocol

The 2WoPS protocol employs three techniques for overcoming network losses:

- *ACK/resend*: allows specially marked TPDUs to be ACKed back to the sender, enabling the sender to resend the TPDU until it is confirmed successfully sent. ACK/resend is allowed for TPDUs containing TSDUs, enabling ACK/resend semantics on application messages. If an ACK is lost the sender will not be aware of delivery success and resend the TSDU, so redundant TSDUs must be filtered at the receiver. This technique guarantees successful delivery given an unlimited number of resends and an eventually-consistent network connection. Further, the technique uses a very limited amount of bandwidth to achieve fault tolerance. The main disadvantage with the technique is that the sender must wait a full RTT before a packet is confirmed lost and resend is commenced, and so the time for successful

delivery of a packet is $RTT * losses$. Because the intended use of Ratatoskr is for power-grid actuator commands which are often time-sensitive, ACK/resend by itself will often be inadequate. The 2WoPS protocol also offers limitations to the number of resends to allow applications with strict deadlines to give up on unsuccessful sends after a deadline has been missed. The number of resends is a parameter to a send command and can be different for each TSDU.

- *Temporal redundancy* - performs multiple network level sends for each TPDU and filters redundant TPDUs at the receiver. While this technique is implemented in 2WoPS modules, it is in practice a network layer technique: for each TPDU sent by the transport protocol with temporal redundancy, the network layer sends multiple NPDUs carrying copies of the TPDU. Temporal redundancy tolerates $n - 1$ losses with n copies of the TPDU. The technique consumes n times more bandwidth than regular sends. The main problem with temporal redundancy is that network losses may be temporally concentrated. For example, network congestion will often lead to periods of high loss rates when a router's buffer for an outgoing link is filled or maintenance on a router might disable all connecting links for several minutes. To add to this, by sending several copies of the same TPDU in a short span, the extra bandwidth use might add to existing congestion. The 2WoPS protocol allows applications to specify a delay between sending each temporally redundant copy of a TPDU. Delays between sends will only be truly effective for overcoming failures if the durations of all periods of temporally related failures were known beforehand, but might help to relieve the congestion aggravation effect of multiple sends. The degree of temporal redundancy is a parameter to the send command.

- *Spatial redundancy*: The 2WoPS protocol uses GridStat's redundant path function to offer spatial redundancy: all NPDUs and thus TPDUs over a connection with spatial redundancy

are copied in the GridStat network and sent over multiple paths. Redundant TPDUs are filtered at the receiver. Spatial redundancy tolerate $n-1$ losses with n redundant paths, and is not affected by temporal concentration of errors. It should be noted that common mode failures thoughout the network, such as very high traffic levels during attacks or crisis situations may affect components involved in all redundant paths. The use of spatial redundancy is heavily dependent on whether network topology allows for redundant paths, but it is expected that the high reliability requirements of a critical infrastructure as the power-grid will justify the expense of building a network with a high degree of redundancy. While spatially redundant TPDUs are forwarded throughout the network in parallel, a small delay overhead is introduced by the routing mechanisms for multiple paths. Further, it is unlikely that all redundant paths through the network will provide as low delivery deadlines as the best path achieved with single-path routing, and so use of redundant paths is likely to increase the end-to-end delay deadline of a subscription. Spatial redundancy consumes, assuming that redundant paths are of equal length to a single path connection, $n-1$ times more bandwidth than sends through a single path. It should be noted that the process of allocating redundant paths through the network is more complex than allocating a single path, and so spatial redundancy induces overhead to the network management. The degree of spatial redundancy is a property of the subscription used for sending data, and so is set at connection setup time and subsequently used for every TPDU sent over the connection. A 2WoPS connection can have different spatial redundancy parameters for each direction. For purposes of analysis and evaluation this thesis conciders only the case where both directions of a connection have the same degree of spatial redundancy. The current implementation of GridStat supports only up to two redundant paths, and does not allow redundant paths between leaf-level clouds, but work is done to eliminate these limitations.

A comparison of the various redundancy techniques can be seen in table 3.2.2.1.

	ACK/resend	Temporal redundancy	Spatial redundancy
Overview	Receiver ACKs successful sends, client retries upon missing ACKS up to n times	Each TPDU is sent n times, with k delay between sends	Each TPDU is sent through n physical paths
Failure tolerance	n failures	$n-1$ failures	$n-1$ failures
Bandwidth	$f * (\|TSDU\| + \|ACK\|)$ where f is min of n and # of failures	$n * \|TSDU\|$	$n*\|TSDU\|$, depending on topology
Added delay	$(f-1) * RTT$	$(n-1) * k$	routing overhead, more depending on topology

Table 3.1: Comparison of Redundancy Techniques

3.2.2.2 Combining Redundancy Techniques

The 2WoPS protocol allows for combinations of the redundancy techniques. Temporal and spatial redundancy measures are cumulative and as they reside in the network layer affect all TPDUs, including ACKs. For example, if a sending a TSDU with 3 temporally redundant sends and 4 ACK/resends over a connection with 2 spatially redundant paths in each direction, three NPDUs containing copies of the TSDU will be sent. At the entry-point router, each of the NPDUs will be forwarded to the first routers of the redundant paths, and, assuming no network failures, six NPDUs containing the same TPDU will arrive at the receiver. The receiver will similarly send a single ACK TPDU, which will be sent in three temporally redundant NPDUs, which again will be copied onto the redundant paths. If all ACKs are lost in the network, the sender will resend three new NPDUs containing the TSDU, and so on. This gives application designers the ability to tailor a connection to the exact needs of the application, allowing use of high spatial and temporal redundancy where a low delay is required, or relying on ACK/resend for redundancy for less delay-sensitive applications or where bandwidth is scarce.

3.3 The Ratatoskr RPC

Ratatoskr RPC is a remote procedure call protocol for power-grid control communication built on top of the 2WoPS protocol. The primary application for Ratatoskr is remote operation of power-grid actuators, either to a gateway interfacing multiple devices or by directly controlling intelligent electronical devices (IEDs) with remote interfaces embedded in the acuator itself. The current grid communication system is unable to support developments in the power-grid stress levels and the security threat picture, and propoposed solutions require real-time, reliable control [6].

Ratatoskr draws extensively on the features provided by the 2WoPS protocol. Delivery guarantees for calls is achieved using GridStat's QoS enabled network communications, and fault-tolerance is provided through the redundancy techniques found in the 2WoPS protocol. As the use of redundancy trades off network resources, or worst case delay in the case of ACK/retry, against safety, applications designers are allowed detailed control over the level techniques used for redundancy.

In addition to increased safety through fault-tolerance, pre- and post-conditions on calls are built into the RPC semantics. Pre-conditions are conditional expressions over GridStat variables that are evaulated before execution of an RPC call, and if the expression is negative, the call is negated. Post-conditions are similar expressions evaluated after the call has executed, and negative results are reported back to the application, allowing evaluations of operation outcome.

3.4 RPC semantics

The RPC semantics are not meant as a final specification for a deployment implementation, as a full mapping of the requirements and additional functionality needed for this is well beyond the scope of a master's thesis. It is rather a platform for further research on control communication in GridStat, specifically to:

- Make practical an evaluation of the tradeoff space between and performance of the redundancy techniques employed.

- Create a platform for further exploration of the possibilities of the pre- and post-condition mechanism.
- Illustrate the QoS semantics available with the capabilities provided by the 2WoPS protocol.
- Provide a building block for future GridStat research projects where timely and reliable control communication is needed.
- Function as a simple GridStat control mechanism in power-grid simulations or test deployments, allowing the mapping of new requirements.

As follows, call semantics are designed to be simple and flexible, with extensive use of platform-specific, generic serialization to allow rapid experimentation at the cost of performance. The prototype was implemented on the Java platform as Java was used for underlying GridStat elements.

Call semantics are use dynamic binding: the server exposes methods to remote invocation by registering local calls with names and parameter types. The client specifies the method name and an array of objects representing the parameter values, and these are serialized and transported over the network where the name and parameter types are used as the identifier to the correct call. Return types are unspecified, and so returned objects must be cast to the expected class at the client. The RPC semantics will catch and serialize any exception cast by the method when executed at the server, and these will be wrapped in a special exception and cast again when deserialized at the client. This is a brittle system and requires extensive care during application design, but the dynamic semantics may easily be hidden behind a wrapper class implementing some sort of static interface, for example, Interface Description Language (IDL) semantics.

Ordering is implemented on a per-thread-basis. The RPCs are blocking, and so if a single thread makes two calls to the same server, the first call must either succeed or time out before the next can be attempted. An RPC call timeout includes the time for sending the return, and so for the second call to reach the server before the first, the first call would have to be delayed to twice

it's maximum guaranteed delay. It is assumed this will not occur as GridStat will use a router scheduling algorithm that drops NPDUs that exceed their delay, [15].

3.4.1 Pre-and Post Conditions

For many power-grid control-operations, placing pre-conditions on the execution may help in preserving safety in face of unwarranted situations such as power-grid anomalies or unexpected mechanical operation. Because of the distances involved, communication must be expected to suffer from bandwidth and delay limitations. Thus one may assume that the client-side information about server state is limited and not fully updated, and following this, pre-conditions should be placed on the server-side of the call. Such conditions could be placed in application code using RPC exceptions, but this could lead to variations in semantics between vendor implementations.

Ratatoskr incorporates pre-conditions in call semantics. This gives standardized predicate semantics, easing interoperability between equipment vendors and inter-utility communication. Predicate expressions are modules, accommodating reuse and laying ground for future extensions (see section 6.2.4.4).

Examples of pre-conditions in power-grid operation may be:

- Isolators are actuators that connect and disconnect de-energized power circuits. A precondition could be to verify that a line is de-energized before attempring isolation.
- High voltage equipment carries with it electocution hazard, and another pre-condition could be to verify that no manned maintenance is scheduled at a field site when performing operations that might place maintenance personel in danger.

Ratatoskr further allows application designers to use the same predicate modules used for pre-conditions for placing post-conditions on calls. Power-grid operations are complex, and actuator operations may give unexpected results in face of situations such as mechanical malfunctions or operator overrides. Server-side post-conditions will be able to utilize the rich information environment local to the substation for analyzing the physical outcome of an execution and return only a

brief report to the client. Thus client applications will be able to review the results of calls without having to retrieve large amounts of data from the substation. By allowing a delay before the post-condition is evaluated, the effect of the operation is allowed to stabilize. By designing the post-conditions into the RPC semantics, the result of the post-condition is transferred to the client as a separate send, without affecting the duration of the RPC call itself.

Examples of post-conditions in power-grid operation may be:

- Load tap changer are components of certain transformers that allow adjustment to voltage output during load. A post condition could be to verify the new voltage level after load tap changer operations, or even generate a status report from all connected devices and send it back to the client.

- Transformer protection is a scheme to detect internal faults in a transformer and isolate it by braking all connected lines if a fault is detected. A post-condition could be to trigger a transformer protection scheme after all transformer operations.

3.5 Limitations

Ratatoskr has the following limitations:

- No restarts after failures in client or server are handled. This is discussed in section 6.2.4.1

- Security is not adressed in a proper manner. See section 6.2.5.

- Packet sizes in the 2WoPS protocol are restricted to the maximum packet size of the underlying network.

- The prototype design uses many mechanisms specific to Java, and so is platform dependent.

3.6 Assumptions

Ratatoskr makes the following assumptions:

- Applications are not subject to byzantine behavior.
- All NSDUs are delivered within two times their guaranteed maximum latency or not delivered.

CHAPTER FOUR

DESIGN OF RATATOSKR

This chapter describes aspects of the design of an experimental protoype of Ratatoskr implemented in Java. It is by no means meant as a final version for deployment, but rather as a proof-of-concept implementation, in addition to providing the means of an evaluation of the fault tolerance capabilities of Ratatoskr and as a platform for further research. The purpose of this chapter is to give a proper understanding of the processes used in the evaluation, and to detail the parts of the design related to the key contributions of this thesis.

4.1 Design of the 2WoPS Transport Protocol

This section details aspects of the design of the 2WoPS Transport Protocol relevant to the evaluation of Ratatoskr.

4.1.1 Modules

This section describes the main modules of the 2WoPS transport protocol. The term module here designates a purely abstract collection of related behavior, not necessarily with a mapping to a single class or package. Some modules are implemented as part of GridStat, but are described here as their use is essential for the 2WoPS protocol.

4.1.1.1 2WoPSPeer

The module containing the main interface for the 2WoPS protocol is the `2WoPSPeer`. An application must instantiate a `2WoPSPeer` and connected it to the GridStat network for 2WoPS protocol use. The `2WoPSPeer` module manages the publisher and subscriber used for communicating over the GridStat network, provides the interface for initiating both sending and receiving data, and maintains open connections. An application must instantiate a `2WoPSPeer` and connect it to a GridStat status router with a locally unique name to communicate over the 2WoPS protocol. To receive data, an application registers a *service handler*, a callback function for incoming

connections requests, with a corresponding *service identifier* (a 32-bit integer (*int*) comparable to a UDP/TCP port number). The service handler provides a *message handler*, a callback function for receiving messages, for incoming connections of the associated service type. An application wishing to communicate with a remote application must request a new connection to the remote application's `2WoPSPeer` by specifying the full name (hierarchy position and locally unique name) of the remote `2WoPSPeer`, the desired QoS parameters for the connection, a desired service type and a callback for incoming TSDUs. If the connection setup is successful, the application receives a `2WoPSConnection` object for sending data.

4.1.1.2 2WoPSConnection

A connection over the 2WoPS protocol is contained in a `2WoPSConnection` at each peer. A `2WoPSConnection` connects exactly two end-point `2WoPSPeer`s and is associated with a published variable for sending and a subscription for receiving at each peer. TSDUs are delivered to applications using the callback methods provided during connection setup, and each 2WoPS connection serves only a single service type. Send delay guarantees and spatial redundancy characteristics are specified at connection setup time and remain fixed for the duration of the connection. `2WoPSConnection` provides methods for sending packets, closing the connection and verifying wether the connection is open. A connection at one peer is open if it knows that the other side has set up the connection successfully, and if the connection has not started a closing procedure. Application data can only be sent over a connection if it is open. To send data, an application specifies the TSDU to send, the number of retries for ACK/retry redundancy, and the level of temporal redundancy. For the prototype, an additional parameter for specifying the timeout of the ACK/resend was also included for overcoming delays induced by evaluation environment timeliness inconsistency. A release version of the protocol must have real-time properties to accommodate power-grid operations and the ACK/retry timeout could be derived from the delay properties of the link. Two methods for sending are provided; the `send` method provides controlled-loss semantics and the

`sendAcked` method provides no-loss semantics.

4.1.1.3 Publisher

See section 2.4.1.1. A GridStat `Publisher` module is used by the 2WoPS protocol for sending data. Outgoing connections register a user defined variable with subtype `2WoPS_TYPE`, which allows the publishing of byte arrays. A peer receiving data on the outgoing connection subscribes to the published variable. Data is sent by updating the status variable with a TPDU in the form of a byte-array.

4.1.1.4 2WoPSSubscriptionHolder

See section 2.4.1.2. The subtype of `SubscriptionHolder` for the `2WoPS_TYPE` implements filtering of duplicate TPDUs, TPDU data validation and a queueing system for delivery of TPDUs to the 2WoPS transport layer. Each `2WoPSConnection` is associated with an underlying `2WoPSSubscriptionHolder` that handles incoming NPDUs for the subscription used for receiving data for that connection.

4.1.2 Sending Process

A diagram of the sending process is found in figure 4.1. Each of the steps are described below.

1. Application calls send on `2WoPSConnection` with the TSDU and ACK/resend and temporal redundancy levels as parameters. In `2WoPSConnection`, the TSDU size is verified to be below max, a TPDU header is assigned to the TSDU. If the QoS parameters dictate ACK/resends, or if the send type is no-loss, a request for an ACK is noted in the TPDU header. The packet is forwarded to `2WoPSPeer` with temporal redundancy and the variableID for the status variable used for send by the connection as parameters. The sending thread is blocked until an ACK is received or until a timeout occurs if an ACK is requested, else control is returned as soon as all temporally redundant sends are perfomed. The buffer containing application data is the same for the whole sending process for performance reasons. Blocking for sends is done so the application thread is prevented from editing the

buffer contents, and this must also be avoided by other application threads.

2. The TPDU is inserted into a scheduler queue. Sends for all connections are scheduled into the same queue. This is to allow scheduling between connections when several connections are to send at similar times, and to allow a single sending thread for the whole system instead of one per connection. For temporally redundant messages, multiple sends for the same TPDU are scheduled. If TPDUs have temporally redundant sends with delays between each redundant send, the scheduler allows for an earliest possible send time to be placed on each send. Packets are sent in order of earliest possible send time, with FIFO access to the queue[1]. Sending of the TPDU is done by publishing the TPDU byte-array as a GridStat user defined variable through the publisher kept in the `2WoPSPeer`.

3. Upon publishing, the publisher transfers the TPDU wrapped in a NPDU to its status router.

4. The GridStat network forwards the NPDU to the receiver's subscriber. If spatial redundancy is used, all NPDUs are copied onto all redundant paths, even those already copied with temporally redundant sends. That is, if temporal redundancy three is over a connection with spatial redundancy two, six copies of each TPDU will be delivered to the remote peer, assuming no network losses.

5. The subscriber at the receiver reads the vaiableID of the NPDU from its header and forwards it to the corresponding `SubscriptionHolder`. For 2WoPS TPDUs, this is the `2WoPSSubscriptionHolder` is associated with the `2WoPSConnection` to which the TPDU is addressed.

6. The `2WoPSSubscriptionHolder` unwraps the TPDU from the NPDU and verifies data

[1]This scheduling technique is inadequate for a deployment of the system, and should take into account delivery deadline and perhaps connection priority. A simple scheme was chosen as the prototype implementation is inherently unreliable with respect to timeliness, and so the effects on timeliness from an improved scheduling system would be of little value. Further, the selection of a proper scheduler is outside the scope of this thesis, and left for future work. See section 6.2.1.1.

integrity. If the TPDU is redundant to an earlier TPDU it is dropped, else it is delivered to the `2WoPSConnection` associated with the `2WoPSSubscriptionHolder` for further processing. Further processing is done in a separate thread because the subscriber thread is common to all connections. TSDUs are delivered for further processing by a FIFO queue.

7. The TPDU is processed according to its type. If the type dictates that the TPDU should be ACKed, the ACK is sent out on the network before other operations (7.1) so as to not extend the round-trip time. Temporal redundancy parameters to be used for sending the ACK are noted in the TPDU header and are the same as for the original send. If the TPDU contains a TSDU, the connection verifies that the TSDU has not been delivered to the application before and if so delivers the TSDU and records the delivery (7.2). An eventual ACK is returned to the sender before any TSDUs are delivered to the application, so the sending of the ACK is not delayed by application processing. The `2WoPSConnection` executes the callback method for application message receival in the same thread as is used for processing of TPDUs, and cannot process any TPDUs before control is returned. Together with FIFO scheduling between the `2WoPSSubscriptionHolder` and the `2WoPSConnection`, this guarantees that TSDUs are delivered to the application in the same order they are received.

8. Spatial redundancy is in effect also for the ACK TPDU.

9. When the 2WoPSConnection processes an ACK, the thread that sent the packet is notified. The thread should wait for an ACK unless it has timed out, which would mean the delivery guarantee is broken. If the thread has timed out and started a resend, the ACK for this send is still delivered as it still confirms that the TSDU was delivered. If the thread has timed out and returned to the application, the received ACK is ignored. When the receiver thread is awakened, it returns a successful delivery notice to the application. If a no-loss send times out for all retries, an exception specifying that delivery status is uncertain is returned to the

application. As controlled-loss receives no ACK on the final send, an uncertain delivery is part of its intended behavior and no exception is cast. All ACK/resends are done with a new TPDU so it is not filtered at the `2WoPSSubscriptionHolder` in the receiving peer.

4.2 Design of the RPC Mechanism

The design of the Ratatoskr RPC mechanism centers around a similar architecture to the 2WoPS protocols `2WoPSPeer` and `2WoPSConnection`; an application creates a peer object to establish use of the mechanism, which provides interfaces for both RPC server and client tasks. The peer object registers as a 2WoPS peer with the GridStat network and readies itself for communication. An application can take the role of either client, server or both. To act as a server, the application registers Java methods with the peer object, which make these available for remote invocation. A client must first obtain a connection to a Ratatoskr peer with registered server methods through the peer interface, and when a connection is established may perform remote procedure calls through the connection interface. An application can register as several Ratatoskr peers allowing for multiple interfaces, and each peer interface may establish several connections with different QoS parameters to the same server.

4.2.1 Modules

4.2.1.1 RPCClientSession

A client-side connection to a server for sending RPCs is represented by an `RPCClientSession`. `RPCClientSession` provides methods to the applications for sending RPC calls or closing the connection. An `RPCClientSession` contains a corresponding `2WoPSConnection` for sending the RPC calls. `RPCClientSession`s cannot share the same 2WoPS connection.

4.2.1.2 CallRepository

`CallRepository` is a repository of calls exposed to remote invocation. It also contains behavior for executing the calls.

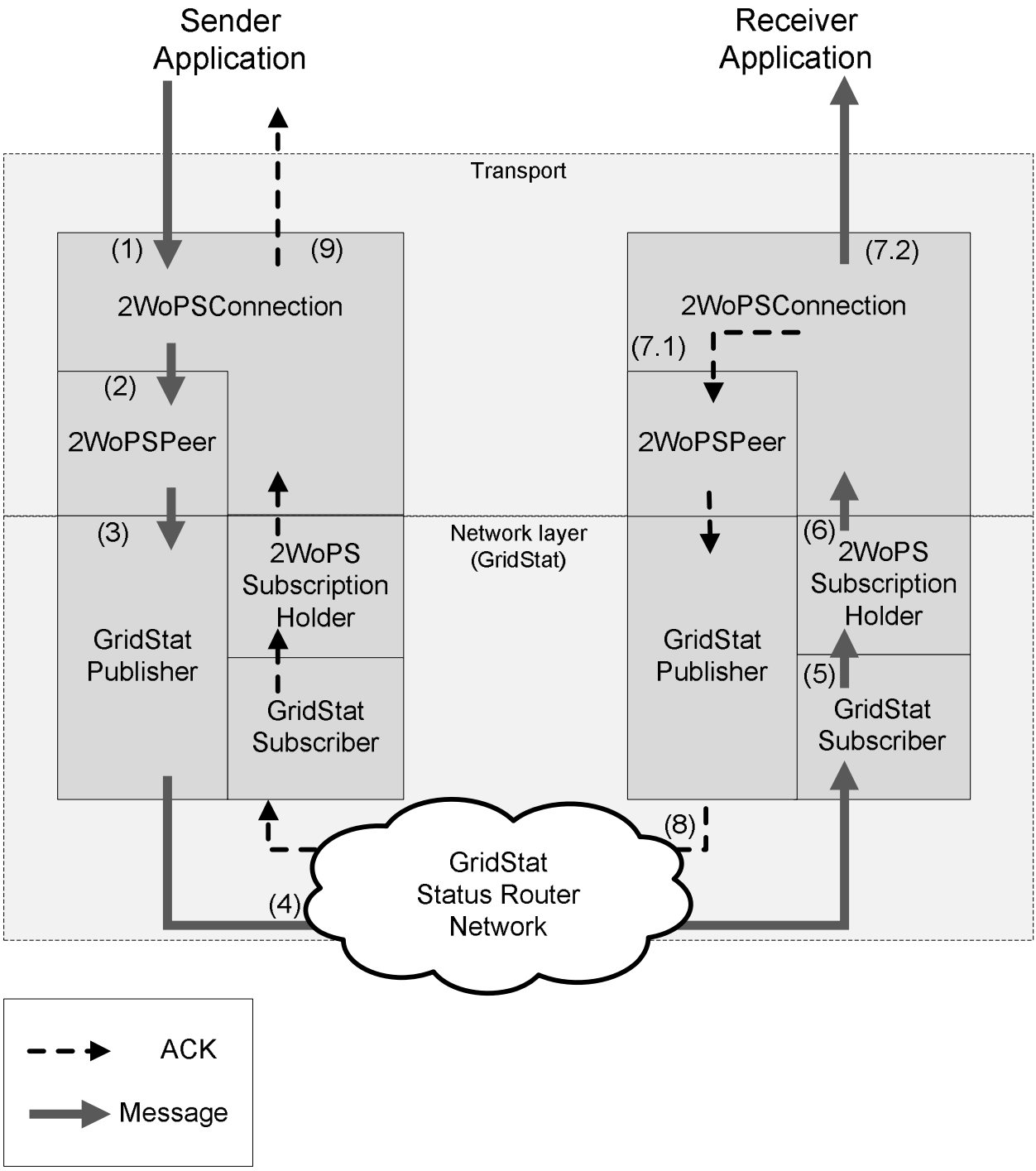


Figure 4.1: Sending Process for the 2WoPS Protocol

4.2.1.3 RPCPeer

The main interface for Ratatoskr RPC usage is the `RPCPeer`. The `RPCPeer` contains and manages the 2WoPSPeer used for communication and receives all incoming messages, and provides methods to the application for opening new sessions and for exposing local calls for remote invocation.

4.2.2 Use of Reflection and Serialization

Ratatoskr RPC draws extensively on the reflection and serialization features of the Java language. Reflection allows observation of the structure of the running program, and serialization allows automatic conversion between bytestreams and Java objects based on class definitions. This is counter to the interoperability goal of GridStat as a middleware platform as these features may only be used in Java environments, but it allows for very simple and flexible semantics for prototyping before defining the requirements of a final, platform independent implementation.

To register a method for remote invocation, the client must specify the name by which the method is identified among the remote methods, an object and a Java `Method` object corresponding to a method implemented by the object. Java's `Method` object contains an array of `Type` objects corresponding to the types of parameters to the method. All parameter types to the method and the type of the return value are required to be serializable. The name given to the method must be unique within the peer, so no method overloading is available.

To perform a remote invocation, the client specifies a name and an array of serializable objects corresponding to the arguments to the call. The name and parameters are serialized and sent to the server. At the server, the name and argument array is deserialized. The name is matched to the names of remotely exposed methods, and the argument object types are checked to be the same as the types of the parameters of the call. If the name and arguments match, reflection is used to call the method with the arguments on the object the server application registered with the method. If the call is successful, the result is serialized back over the network. If an exception is thrown it

is wrapped in an `ApplicationException` which is serialized instead of a result. If a method with the client-specified name is not found, or if the parameter types of the corresponding method does not match the types of the arguments provided, a `SemanticsException` is serialized and returned. When receiving the result TSDU, the client deserializes the result. The result object's class is not specified, and so the client must typecast it dynamically. If the result is an exception, it is thrown to the client.

This dynamic binding approach can be replaced with static bindings. For example, it would be possible to implement a program that takes XML interface descriptions as input and generates wrappers around RPC client and server. The server wrapper would then create hook objects where the application designer attached the code to be executed, and automatically register these hooks with the `RPCPeer`. The client wrapper would generate an interface with methods corresponding to the server calls and typecast the return.

4.2.3 RPC Flow

A diagram of the process of sending an RPC call is outlined in figure 4.2. Before the RPC invocation can be initiated, the client must have set up an RPCSession to a server that has registered methods for remote invocation. Explanations for the points in the figure are as follows:

1. The application initiates a remote method invocation to the `RPCClientSession` by providing the method's remote name, an array of objects corresponding to the call's parameters, and the QoS parameters for sending the call. The QoS parameters include temporal redundancy and ACK/retry settings and a maximum expected server execution time. The name and arguments of the method, the QoS parameters and a call sequence number is inserted into a struct representing the call, which is serialized into a bytebuffer and sent to the server. The thread waits on the sequence number using a timeout of up to two times the maximum delay guarantee for the `2WoPSConnection` (for sending the invocation and the result, substracting time spent waiting while sending) plus the server execution time, thus blocking

the application for the duration of the call.

2. The RPCClientSession controls a 2WoPSConnection to the server over which the serialized call is sent using the temporal redundancy and ACK/retry parameters specified by the application. The thread then waits on the sequence number inserted into the call.

3. The `RPCPeer` receives all incoming messages. Receiving calls for all sessions at the same place allows for easy prioritization in the future. As the call is a TSDU with regards to the 2WoPS protocol, any ACKs are sent before the call is processed.

4. The message is deserialized and typecast to the callstruct. Incoming calls are placed into a queue for asynchronous processing so the receiving thread can return to the 2WoPS protocol to receive new messages. A threadpool processes calls from the queue. A thread retrieves the next call to be processed and forwards it to the CallRepository.

5. The call retrieves the method corresponding to the call and invokes the call using Java reflection.

6. The result or any exception is inserted into a struct representing an RPC result. The struct is returned to the call processing thread.

7. The threadpool thread inserts the call serial number from the call struct into the result struct, serializes the result struct and sends it back to the client using the QoS parameters stored in the call struct.

8. When the `RPCPeer` receives the result, it wakes the thread waiting on the sequence number and passes the result as a parameter.

9. If the thread is awoken by a result, it unwraps the result value and returns it to the application. If the thread is awoken by a timeout, it throws an exception specifying that the execution status of the call is unknown.

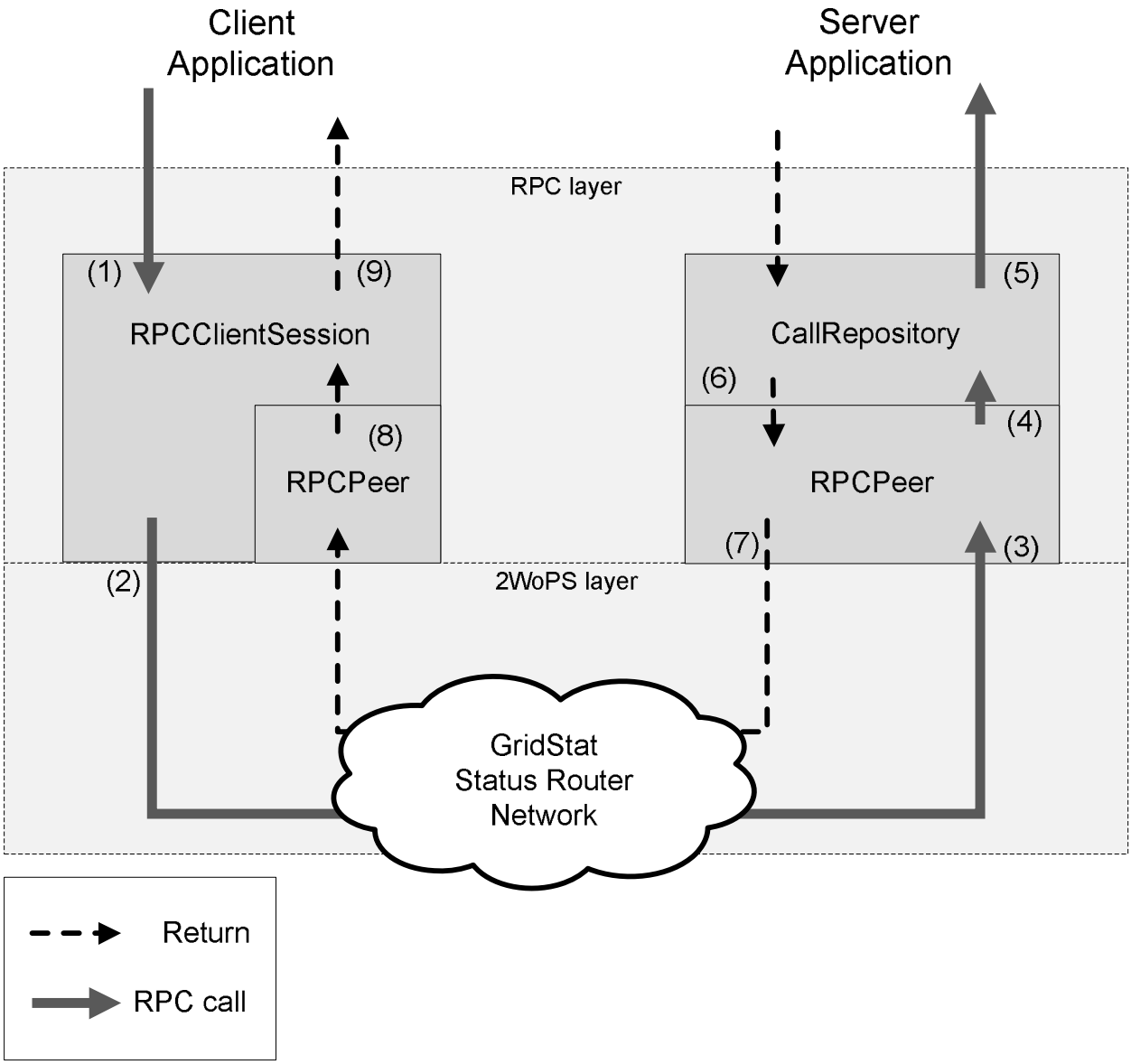


Figure 4.2: RPC Send Process

4.2.4 Pre- and Post-conditions

Application designers define pre- and post-conditions with classes implementing the `Condition` interface. The `Condition` interface implements a single method, `evaluate()`, that takes no parameters and returns a void. `evaluate()` should only return normally if the condition is satisfied. If the condition fails `evaluate()` should throw a `ConditionException`. Information on why the condition failed can be added to the exception, and the application programmer can implement subclasses for additional semantics. For the current prototype, pre- and post-conditions are simple interfaces: Implementations must be provided by the application designer. A recommended strategy for implementing conditions is to give condition access to GridStat subscriptions, and to use subscribed values in the implementation of `evaluate()`.

When registering a call for exposure to remote invocation, the server may specify two `Condition`s, `preCondition` and `postCondition`, and a delay. The delay allows post-conditions to execute a duration after the call, to allow the state of the system to settle before the condition is evaluated. Pre- and post-conditions registered with a remote method are stored in the `CallRepository` at the server, together with the call state. The `CallRepository` is also responsible for the verification of the conditions during the call process.

If a call is registered with a precondition, the precondition must be satisfied before the call can be executed. If the pre-condition is violated, a `ConditionException` is thrown to the client application instead of a result. When the `CallRepository` receives a call request from the `RPCPeer`, it will first check whether the call is registered with a pre-condition. If so, the CallRepository will invoke the `evaluate()` method of the pre-condition before executing the call. If `evaluate()` returns normally, the call is executed normally, but if `evaluate()` throws a `ConditionException`, the call is aborted and the exception is inserted into the result struct returned to the client. The client unwraps the exception and throws it back to the client application. The sending process when failing a pre-condition is illustrated in figure 4.3.

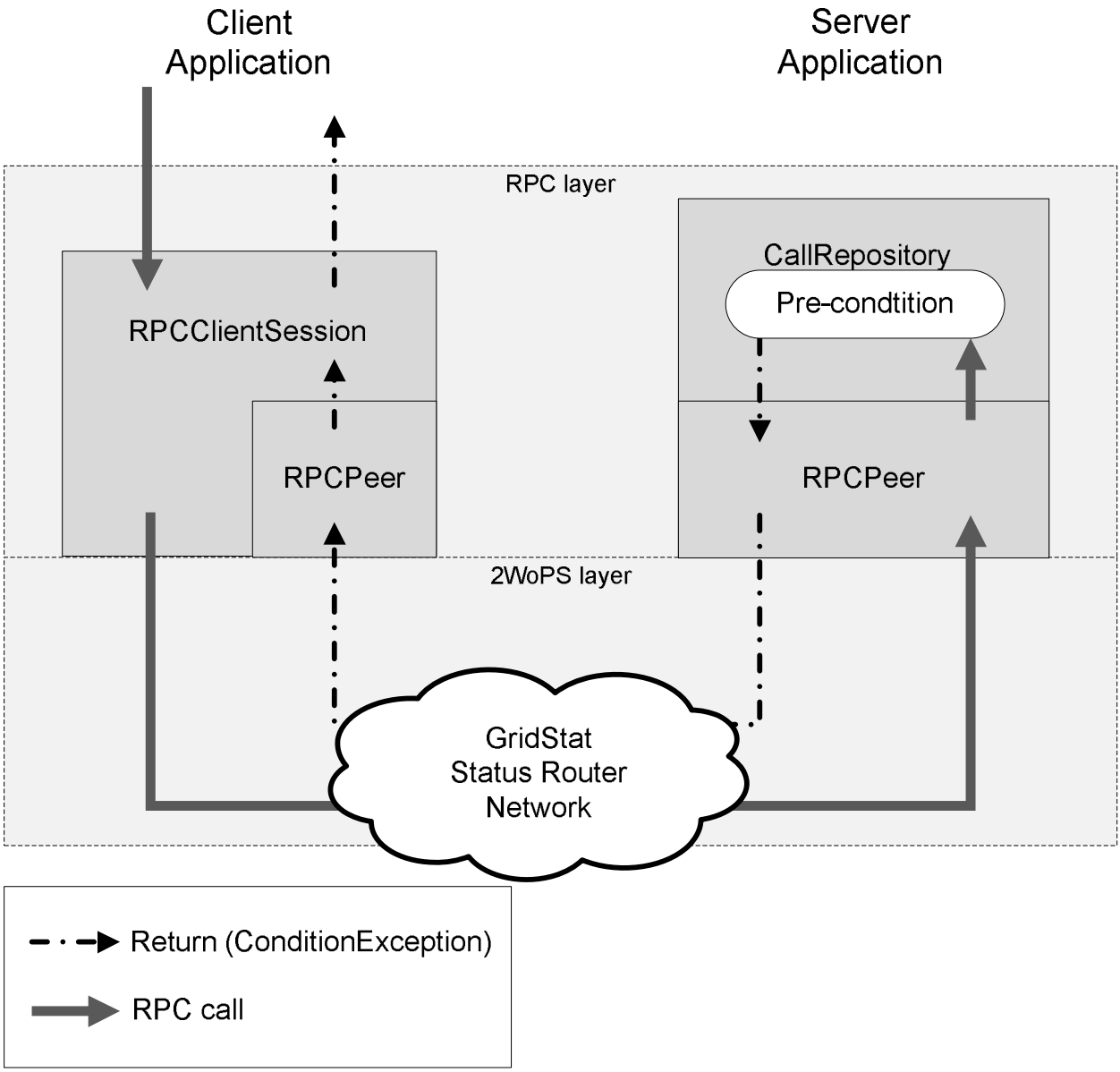


Figure 4.3: Call Process When Failing Pre-Condition

Post-conditions are placed outside the regular call-procedure to prevent any delays before condition evaluation to affect the end-to-end delay of the call. The sending process with a registered post-condition is illustrated in figure 4.4. The CallRepository checks whether a call has any post-conditions only after executing the call. If so, the a struct containing the `Condition` module, addressing information to the client and the sequence number of the call is inserted into a scheduler queue using the post-condition delay as the earliest time at which the structure can be retrieved from the queue. A pool of threads processes the post-condition structs. After a struct has been retrieved from the queue, the thread-pool thread executes the `evaluate()` method on the `Condition` contained in the struct, and generates a `ConditionStruct` using the result. The `ConditionStruct` is the result sent back to the client. It contains a variable indicating the state of the post condition (satisfied or violated), any `ConditionException`s thrown, and the sequence number of the call. A ConditionStruct is returned no matter the result of the expression, so that the client will receive no post-condition results only in the cases where no post-conditions are present or when the post-condition is lost. The pool thread sends the `ConditionStruct` with the same level of redundancy used for the call. To obtain post-condition results, the client application has the option to specify a callback handler for post-condition results when it invokes a remote call. If no such handlers are specified, any post-conditions will be ignored. The handler is associated with the sequence number of the call. When the RPCPeer receives a `ConditionStruct`, it looks up the sequence number in the struct against its registered handlers, and delivers it if a handler is present.

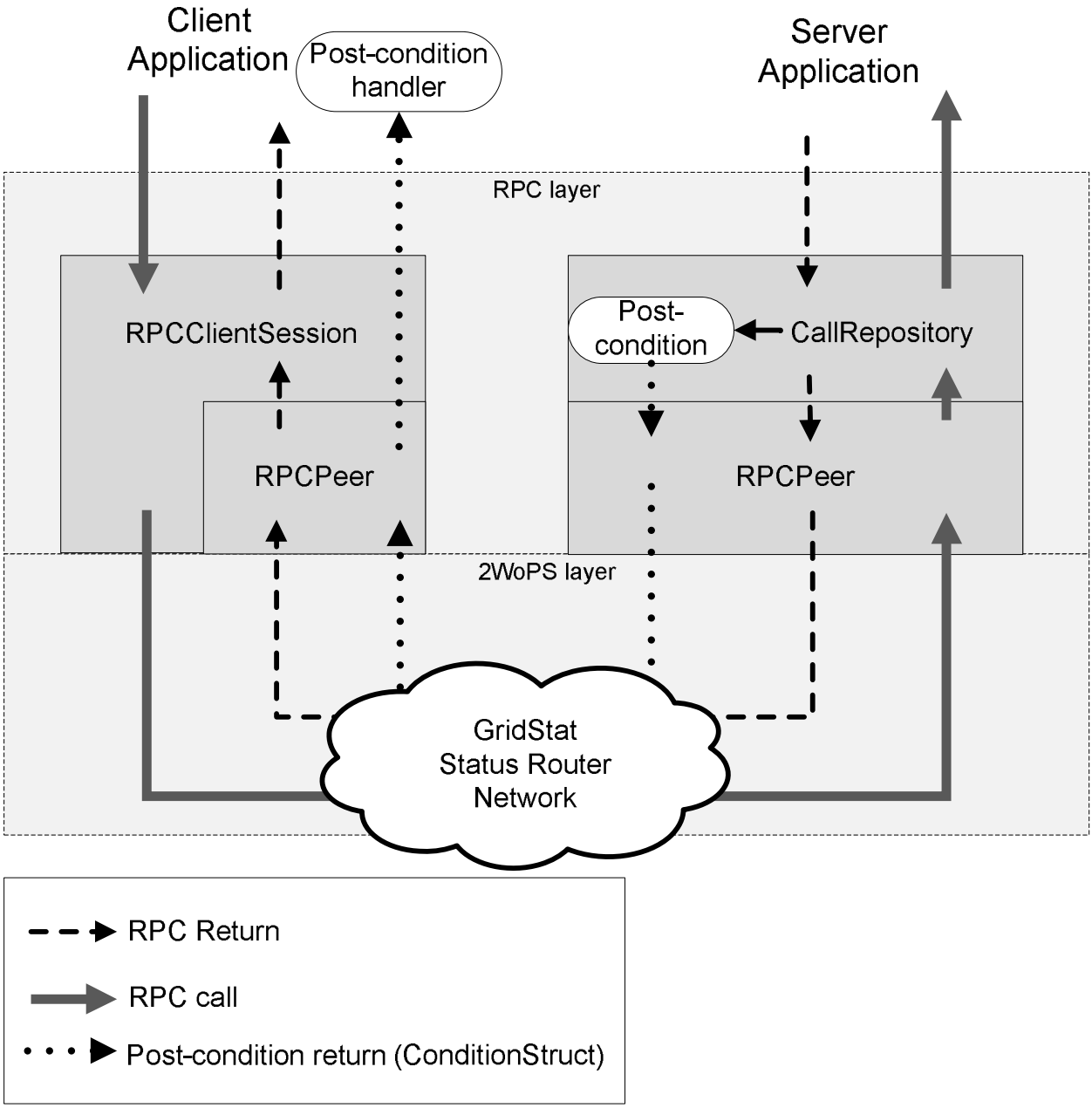


Figure 4.4: Call Process With Post-Condition

CHAPTER FIVE

EVALUATION

Ratatoskr is evaluated with respect to performance in face of network faults. The purpose is to understand the efficiency of the implemented fault tolerance techniques on RPCs over a faulty network, not to evaluate the performance of the prototype implementation, as a real-time implementation is outside the scope of this thesis.

5.1 Evaluation Procedure

Evaluations were performed by connecting Ratatoskr client and server processes to a small GridStat network and commencing a number of RPC calls from the client to the server. To introduce network faults, event channels between status routers were routed through a network link emulator introducing delay and errors. Links going through the link emulator are referred to as *emulated links*. This setup tries to emulate control traffic over a wide area network of status routers, where both the client and server are either connected to their entry-point SR by LAN or running as processes on the same computer. No delay or loss is induced on the link between Ratatoskr peers and their entry-point SRs. This is based on the expectation that a deployment of GridStat will include a wide deployment of status routers throughout the power grid to provide a high degree of network redundancy, which makes it likely that sites using RPC also contain a status router.

5.1.1 Topology

The topology of the evaluation setup is shown in figure 5.1. 13 status routers form two paths between the client and the server, one of 7 links and one of 6. This is to allow for the fact that when using spatial redundancy additional paths may often be longer than a single best path. The current implementation of GridStat does not allow more than two redundant paths. Future versions of GridStat will not have this limitation, which will allow an evaluation of more than two spatially redundant paths, see 6.2.6. No additional links and status routers outside the two paths were

employed as routing between the same two peers in a static network (as the current version of GridStat is) will result in the same path no matter the errors.

5.1.2 Network Fault Model

GridStat uses multiple underlying network technologies, spans a wide area, and will sustain several usage patterns (usage patterns to this point includes rate based status updates and bursty RPC traffic). This makes for very complex behavior and it is difficult to provide a good fault model for GridStat without field testing. For this evaluation, two fault models were combined to account for the rich diversity of potential fault patterns in a GridStat deployment.

- *Omission-fault* - Each link is assigned a uniform probability of dropping each packet passing through it. The drop probability is the only variable for the omission-fault model. Each drop is completely isolated; no other link or later or earlier packet on the same link is affected by a drop. Omission-fault attempts to model temporally and spatially isolated drops in links where the no retry-upon-failure is attempted below the transport layer in the protocol stack. Examples of unifrom drop rate errors are background noise or very short term physical interference in links causing packet data corruption: passing physical objects in the way of the beam of a microwave beam, bursts of electromagnetic noise from power anomalies in a substation wired with copper or frequency noise from grid devices in a broadband over power link.

- *Duration-fault* - Each link is in one of two states: disabled or enabled. If disabled, all packets passing through the link is dropped. If enabled, the operation of the link is not affected, and all packets pass through the link unless omission failures occur. Links are ordinarily enabled, except for 1-second periods of disable state. Disabled state periods occur by a Poisson process, where the average number of occurrences per second (λ) is the only variable for the duration-fault model. Duration-fault attempts to model transient failures in network components. Examples may include: router maintenance, fiber cuts or local power-outs.

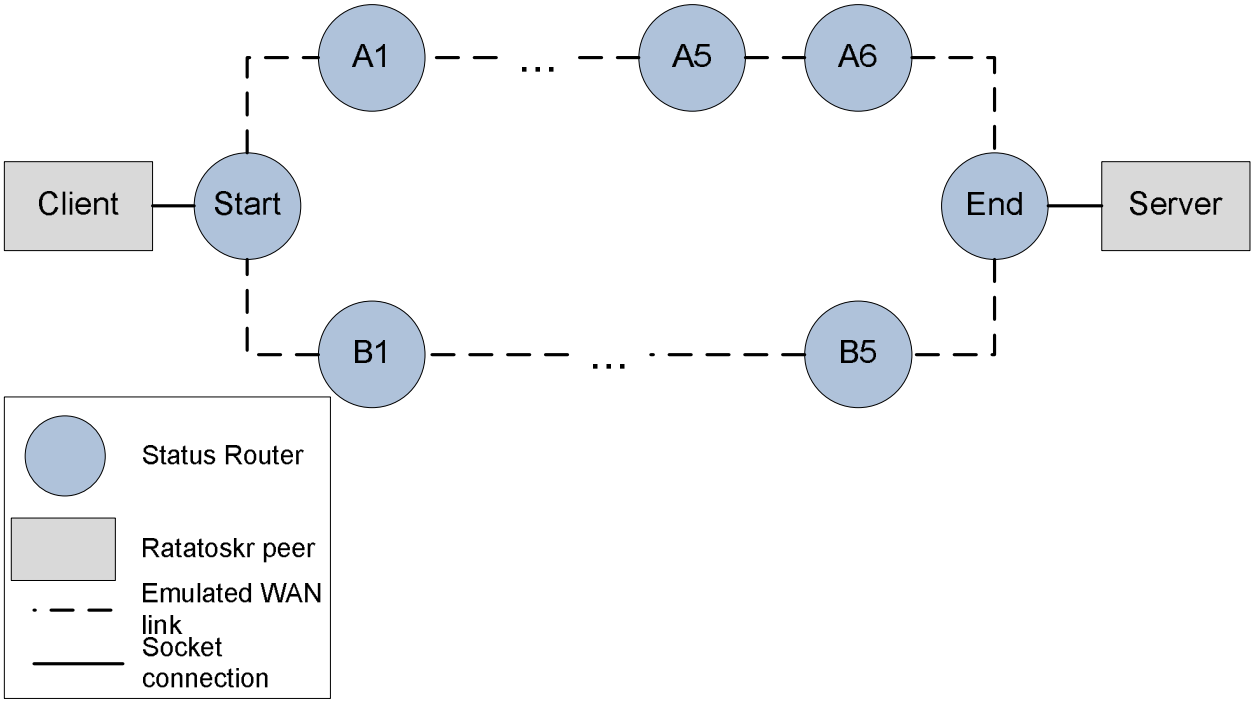


Figure 5.1: Evaluation Topology

Such failures may certainly have a duration of well over a second, but a one second disable state duration is enough to notice the effects of duration failures on communication. It should be noted that dynamically routed networks quickly will adjust so signals will circumvent duration faults, even nearly instantaneously [19]. GridStat uses static routing, and so paths will not be adjusted even if faults are detected. A future solution for this in GridStat is for the QoS hierarchy to create new paths around failures, but this is an expensive operation when resource management calculations and communication to status routers is taken into concern, and must be expected to be time consuming. The primary mechanism for overcoming longer-term failures in GridStat is spatial redundancy.

5.2 Evaluation Testbed

5.2.1 Processes

The processes used for testing were:

- 13 GridStat status routers
- 1 GridStat leaf QoS broker
- 1 evaluation program implementing a Ratatoskr client peer *(client)*
- 1 evaluation program implementing a Ratatoskr server peer *(server)*
- 1 Sun Java CORBA nameserver
- 1 Java program providing socket tunnels with loss and delay properties *(network link emulator)*

5.2.2 Hardware

The evaluation was tested on a single computer, running a Core2 duo 2.13GHz dual-core processor and 4 gigabytes of RAM. The operating system was Ubuntu linux, kernel 2.6.20-16 compiled with 1 millisecond kernel tick intervals and full kernel preemption. All evaluated programs were implemented in Java, compiled and run with sun java2SE 6 (version 1.6.0.00). All inter-process communication was done over operating system UDP sockets for GridStat data traffic and Sun's Java 2 Platform CORBA Package for control. All tests were performed in user mode with regular process priority and with the graphical operating system interface turned off.

5.2.3 Garbage Collection Handling

The Java platform used for evaluation does not support explicit deallocation of process memory. Freeing memory is handled by a garbage collector (gc). The Java garbage collector locks process execution during deallocation, which affects experimental results. This behavior was especially apparent in the client and server processes, mainly due to serialization and deserialization of call arguments, which expends several kilobytes of memory for caching. To compensate for this, a mechanism was implemented to measure the impact of garbage collection and substract this from call results. The mechanism queries the Java

`java.lang.management.GarbageCollectorMXBean` interface for recent garbage collection operations and their durations, and if gc operations have occured, substracts the gc operation duration from the end-to-end duration of the affected call. Plots of operation with and without garbage collection compensation can be found in figure 5.2. With garbage collection compensation, four calls have an end-to-end duration of 25 milliseconds or more, against 10 without the compensation.

5.2.4 Java Virtual Machine Arguments

- Client, server and link emulator were run with arguments -Xms256m -Xmx256m, allowing up to 256 megabytes of process memory before commencing garbage collection. No garbage collection was observed for the link emulator during the evaluation.

- Status routers were run with -Xms128m -Xmx128m, providing 128 megabytes of process memory. No garbage collection in the SRs was observed during the evaluation.

IntelliGrid

5.3 Experiment Procedure

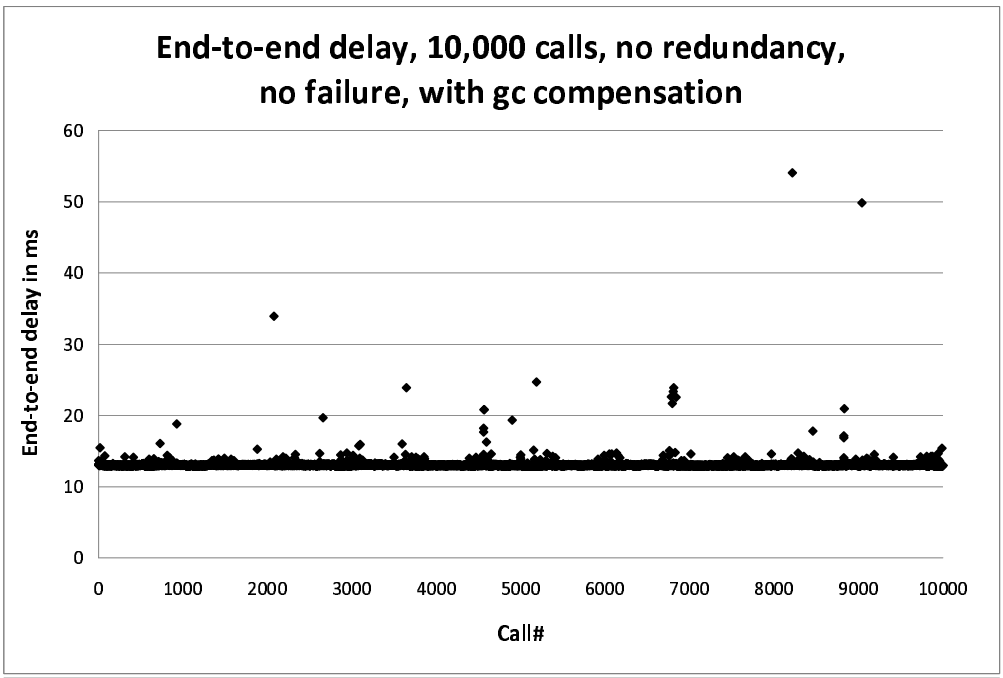
- For all experiments, the process running the Ratatoskr client also functioned as experiment coordinator.

- For each new experiment session, 10,000 RPC calls were made to warm up the system, and then one or more experiments were run sequentially.

- An experiment consists of 10,000 RPC calls with data gathered from each call.

- A new Ratatoskr connection was established for each experiment, and closed at the end of the experiment.

- All calls were made with an unlimited number of ACK/retries.



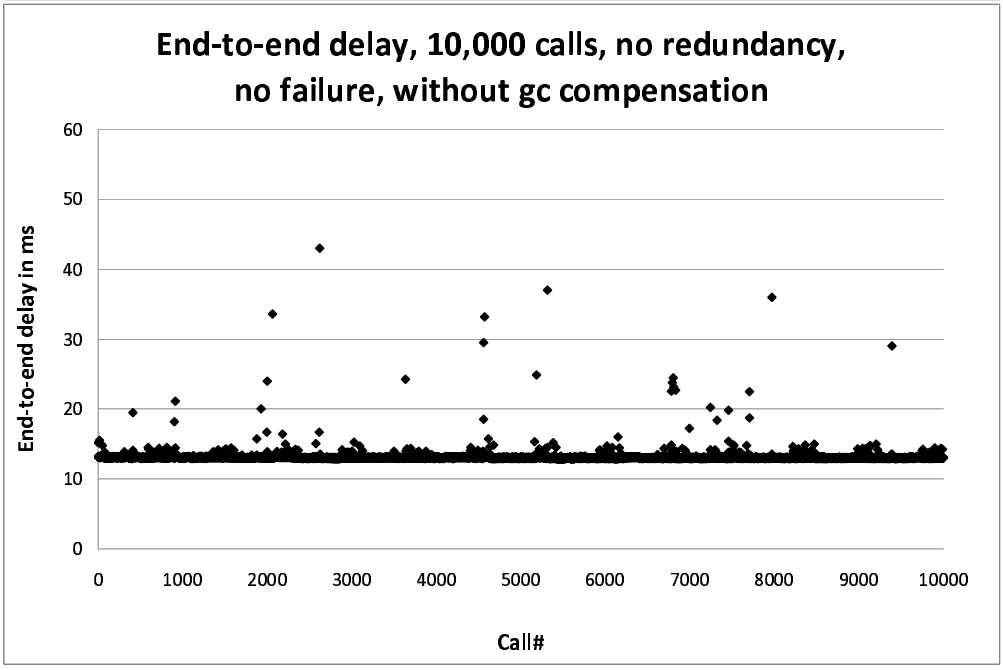


Figure 5.2: Comparison of Performance With and Without Garbage Collection Compensation

- Between calls, all links emulated in the link emulator was reset, specifically by setting the internal clock used for duration failure $2 * (1/\lambda)$ seconds into the future, in effect ending all previous disabled states and allowing new ones to arrive.

- A link delay of 1 millisecond was used unless specified otherwise in the experiment description.

- When temporal redundancy was used, a two millisecond delay between redundant sends was used.

- For all experiments without spatial redundancy, calls were made over a 6 emulated link path. For experiments using spatial redundancy, calls were made over one path of 6 emulated links and one path of 7 emulated links.

- The timeout for the ACK/retry technique had a base of 25 milliseconds, allowing 12 ms for transfer delay, 20 ms for garbage collector and 3 ms for link emulator and Ratatoskr overhead. Higher timeouts were assigned by Ratatoskr to sends with temporal or spatial redundancy due to the wait between redundant sends and the extra hop in the spatial paths. Specifically, using spatial redundancy added 2 ms to the timeout, and using temporal redundancy added 2 ms for each redundant send.

5.3.1 Result Data

The following data was extracted from each call:

- Time was measured from when the call was made until the result arrived (*end-to-end delay*).

- The number of timeouts experienced was recorded. This includes any timeouts the server experienced while attempting to send the result. Specifically, the number of timeouts for a call is the number of timeouts for the first time of a call to arrive at the server *not including*

timeouts introduced by missed ACKs, and the number of timeouts for the result to reach the client the first time, again not including timeouts introduced by missing ACKs.

- Early success rate for experiments is defined as the number of calls with no retries divided by the total number of calls.

5.4 Expected Results

While Ratatoskr provides tolerance for network failures, sends made with redundant network communication will still fail if all redundant NPDUs fail. What Ratatoskr provides is a much higher tolerance for network failures than RPC mechanisms with lower levels of redundancy techniques. This means that a Ratatoskr RPC call is more likely to succeed with fewer timeouts. In the context of real-time distributed systems, it is highly desirable to obtain a very high probability for calls to succeed without any retries, in this evaluation referred to as an *early success*. While the exact number of retries is difficult to predict without extensive data on network fault patterns from a deployment, this evaluation attempts to establish a model for evaluating the level of redundancy needed to achieve a desired level of early success rate. An analysis of failure probabilities related to RPC calls can be found in table 5.4. It should be noted that the probability for failing two redundant packets is the square of failing a single packet, and so the efficacy of redundancy increases with the reliability of the network. For example, if a path has a drop rate of 10% end-to-end, two redundant sends over the link will have 1% drop rate, while four redundant sends will have 0.01%. For a link with 1% drop rate, the drop rate with two resends is 0.01% while with four resends it is 0.000001%.

For the purpose of analyzing the results of the experiment results, where error probability is involved, some simplifications are made. As duration faults appear in 1-second durations with an average n seconds between, the probability of a link being down on the first call attempt is simplified to $\lambda = 1/n$). Further, the probability of a link entering a disabled state after being enabled at the beginning of a call is ignored. This is justified in that experiments operate with a

Symbol	Description	Formula
l	#of links in path	constant
p_e	Error probability for each link	constant
t	Temporally redundant sends	constant
$p_{sn}(l, p_e)$	Probability of successful delivery on a path with l links and link error probability p_e, without redundancy measures	$p_{sn}(l, p_e) = (1 - p_e)^l$
$p_{st}(t, p_{sn})$	Probability of successful delivery on a path with temporal redundancy t	$p_{st}(t, p_{sn}) = 1 - (1 - p_{sn})^t$
$p_{rt}(p_{s_1}, ..., p_{s_r})$	Probability of successfull delivery over r redundant paths with success probability $p_{s_1}, p_{s_2}, ..., p_{s_r}$	$p_{rt}(p_{s_1}, p_{s_2}, ..., p_{s_r}) = 1 - (1 - p_{s_1}) * (1 - p_{s_2}) * ... * (1 - p_{s_r})$
$p_{RPC}(p_s)$	Probability of early success for an RPC call (including return) over a connection with successful delivery probability p_s	$p_{RPC}(p_s) = p_s^2$

Table 5.1: Expected Failure Rates for Redundancy Techniques

relatively high n, and calls are unlikely to span much over a few tens of a millisecond, and so this probability is very small. The one case where it is non-negligible is when a call already experiences a disable-duration as this may lead the call to having to retry for a full second, although such cases must be expected to be extremely rare except for very frequent arrivals of loss durations.

5.5 Experimental Results

5.5.1 Resiliency of Temporal Redundancy

To evaluate the resiliency of the temporal redundancy mechanism, a series of experimental runs was performed with increasing degrees of temporal redundancy (1, 2, 4 and 8 sends). Each of the temporal redundancy levels was tested over a set of omission fault rates (1%, 2%, 4% and 8%) applied to all emulated links. Duration loss was omitted from the evaluation, and is addressed in a later experiment. The results are shown in figure 5.3, with corresponding expected results from analysis. The experimental results match the analysis very closely. It should be noted that the omission failure rate cited in the x-axis is per link, and so the overall omission failure rate for the end-to-end path is higher than the cited number. A comparison of the full end-to-end failure rate

for varying link failure rates is found in table 5.2. The end-to-end loss for a single send (6 links from client to server) is also included as the return send uses full temporal redundancy no matter the number of NPDUs carrying the call that are successfully delivered to the server.

Figure 5.3 shows that two temporally redundant sends are not sufficient to entirely overcome a 1% loss rate, but with 99.2% early successes against 88.5% for the non-redundant calls it is still a good improvement. With 4 resends, 99.92% early successes are achieved at 4% failure rate. For 8 resends, 99.92% of calls were early successfull even at 8% loss rate, where the end-to-end early success rate without reliability was 36.5%. Even during periods of intensive network loss, critical applications where the extra bandwidth for temporal redundancy can be spared should be able to perform RPC calls with very few retries, given that the loss patterns accomodate temporal redundancy.

5.5.2 Resiliency of Spatial Redundancy

The efficiency of spatial redundancy was evaluated in a manner similar to temporal redundancy. Runs were performed with spatial redundancy enabled over increasingly higher occurrence frequencies of duration faults (1 second failure every 10000 seconds, 1s/1000s, 1s/500s, 1s/100s and 1s/50s). It should be noted that 1 second failure per 10000 seconds corresponds to 99.99% availability per link, and that 1 second failure per 50 seconds corresponds to 98% availability, where a minimum of 99.999% component availability is a common requirement for critical networks. The fault rates used in the evaluation are lower than for the evaluation of temporal redundancy as the

Per link failure rate	6-link end-to-end failure rate	12-link end-to-end failure rate
0.01%	0.06%	0.12%
0.1%	0.599%	1.193%
0.2%	1.194%	2.374%
1%	5.852%	11.362%
2%	11.416%	21.528%
4%	21.724%	38.729%
8%	39.364%	63.233%

Table 5.2: Calculated End to End Loss Compared to per Link Loss

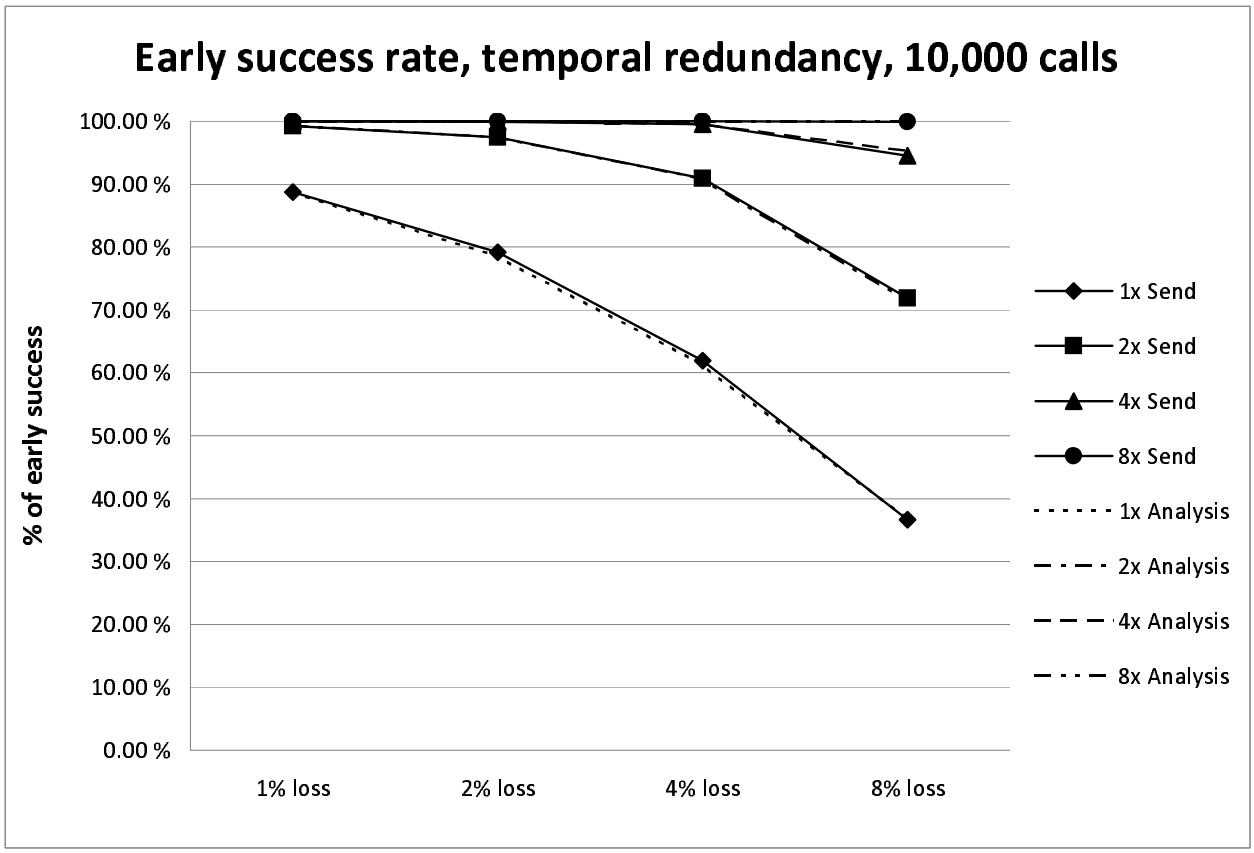


Figure 5.3: Early Success for Temporal Redundancy over Varying Omission Fault Rates

current version of GridStat only supports two redundant paths and as each duration failure results in consecutive retries until the fault duration expires.

Values for expected results based on analysis are included in the diagram. Again, failure rates are per link, and end-to-end failure rates are higher. An estimate of the failure rates can be found in table 5.2, when simplifying the arrival rates of fault durations for links as fault percentages (1s/50s=2%, 1s/100s=1%...). Calls without reliability measures were not included in the test. A comparison for calls with and without spatial redundancy for 1 second failure every 10000 seconds is made in a later experiment.

For the experiments, 100% early successes was achieved for both 1s/10000s and 1/1000s fault rates. As this is for 10,000 calls, this is encouraging as it satisfies at least a 99.99% end-to-end reliability requirement. For 1s/500s fault durations, 99.96% early success is achieved. This seems like a high rate, but a single fault duration incurs several retries which could be catastrophic for a real-time system, and so for moderately critical applications communicating over networks with less than 99.9% component availability more than two spatially redundant paths should be used. The experment's results follow the analysis closely except for at 1s/50s fault rate. This could stem from the analysis simplifying the fault rate to a percentage, which might weaken the analysis for high fault rates.

5.5.3 Comparison to Traditional RPC

A final experiment was made to compare the performance of Ratatoskr to a traditional RPC call without other forms of reliability than ACK/retry, and to evaluate the effect of spatial and temporal redundancy on the failure models used. Here, performance refers to the ability to tolerate network faults and end-to-end delay over a faulty network. Experiment runs for no redundancy, 4 temporally redundant sends, spatial redundancy and the combination of the two were performed over 1% omission fault rate, 1 second fault duration every 10000 seconds, the combination of the two and no failures. The early success rates are shown in figure 5.5. The average duration with

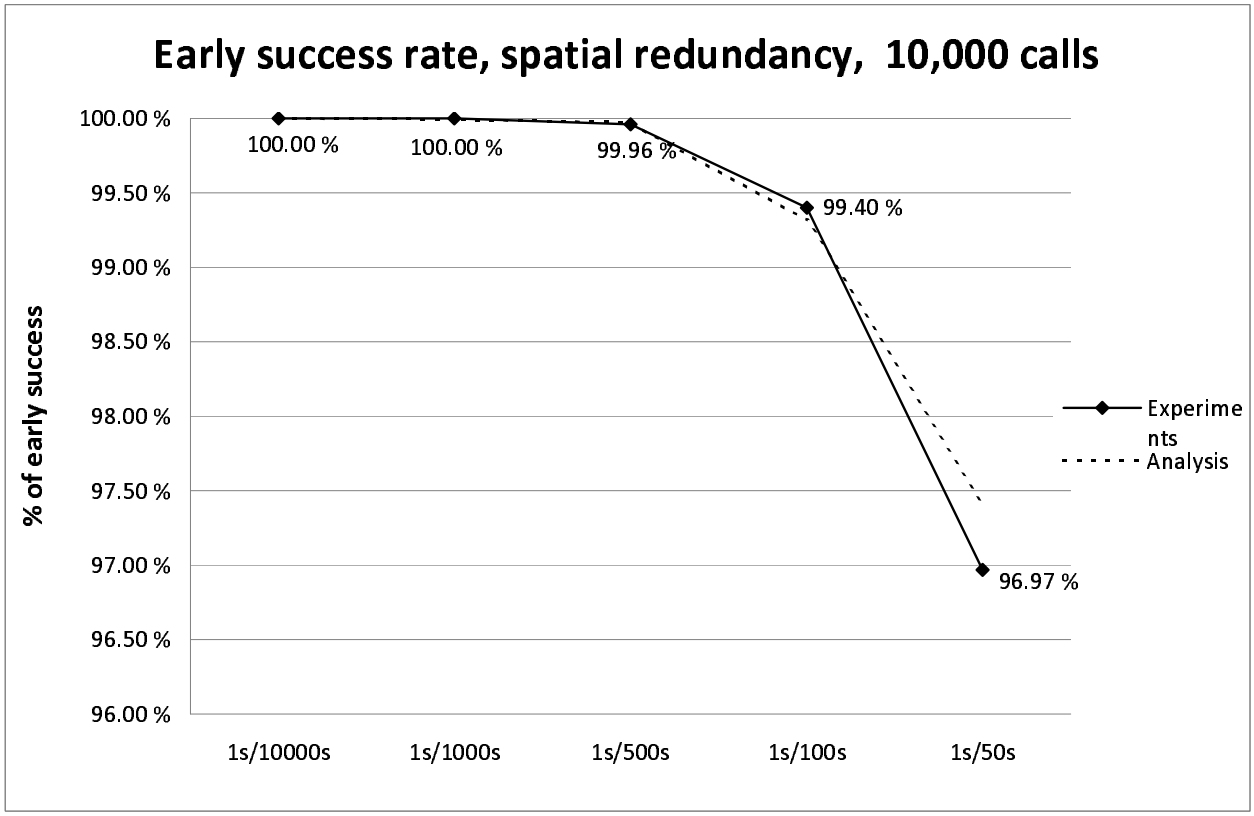


Figure 5.4: Early Success for Spatial Redundancy over Varying Duration Faults

standard deviation for all redundancy levels with both fault models combined is shown in figure 5.6. Cumulative distributions of timeouts experienced before call success for all redundancy levels for the combination of fault models is shown in figure 5.7 (note that the percentage range on the graphs are different to allow visible details). The highest calltimes measured with the combination of losses are: 939 milliseconds for no redundancy, 91 ms for spatial redundancy, 977 ms for temporal redundancy and 24 ms for the combination of the redundancy techniques.

From figure 5.5, it is seen that for 1% omission loss, temporal redundancy experiences very few timeouts, but the spatial redundancy does not provide enough reliability for the fault rate level and 0.81% of the calls experience timeouts. Without any redundancy, less than 90% of the calls achieve early success, which makes for very unstable calltimes. The 1s/10000s duration fault setting does not affect the early success rate of any of the redundancy levels too much, while the combination of fault models has a pattern very similar to that of 1% omission failure.

In figure 5.6, the effects of duration loss becomes apparent. Temporal redundancy, which is ineffective against duration faults, has a considerably much higher standard deviation of end-to-end delays than spatial redundancy. Together with the low early success rate, this signifies that a few calls must retry several times before success is achieved, even with four redundant sends. The experiment run with no redundancy follows a similar pattern with high standard deviation of end-to-end delays, and also the average end-to-end delay is higher. This is most likely from the high percentage of calls that timed out. The end-to-end delay with both forms of redundancy retain the same average as spatial and temporal redundancy, and with a small standard deviation. The standard deviation patterns are reflected in figure 5.7. 99.85% all of the calls made with temporal redundancy incurred no timeouts, but the distribution has a long tail, and 0.05% of the calls incurred over 10 retries. The highest end-to-end delay measured for temporal redundancy was 977 milliseconds, and the highest number of retries was 22. The cumulative distribution for spatial redundancy compared to the temporal calls reveals that while a relatively high number of

calls experience timeouts (over 0.8%), only a single call experiences the highest number of timeouts, 2, and 91 milliseconds was the highest measured end-to-end delay. With both redundancy techniques, no timeouts were experienced and the highest measured end-to-end delay was 24 milliseconds. Without redundancy, over 12% of the calls experienced timeouts and the highest number of timeouts was 25, with the measured end-to-end delay 939 milliseconds. This is a higher number of timeouts for a shorter measured end-to-end delay compared to the temporal redundancy call, as the timeout is set higher for temporal redundancy due to the 2 millisecond wait between redundant sends. From this we can conclude that even if a control mechanism over a single-path network uses a transport protocol with temporal redundancy, it may still experience very high call durations with longer-term failures in a single component on the path. Compared to RPC calls without the redundancy measures found in Ratatoskr, spatial redundancy greatly lowers the worst-case end-to-end call duration, temporal redundancy improves the average call duration considerably, and the combination improves reliability greatly.

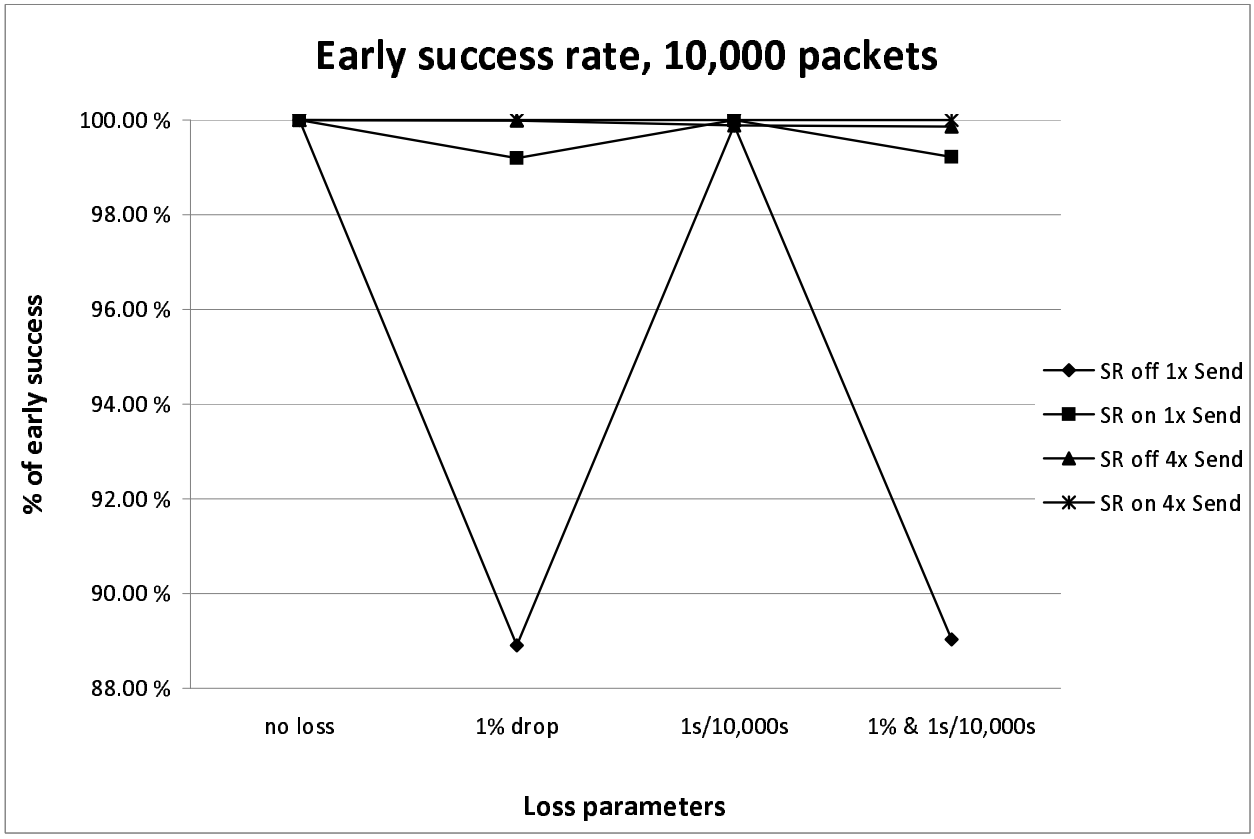


Figure 5.5: Early Success for Varying Redundancy and Loss

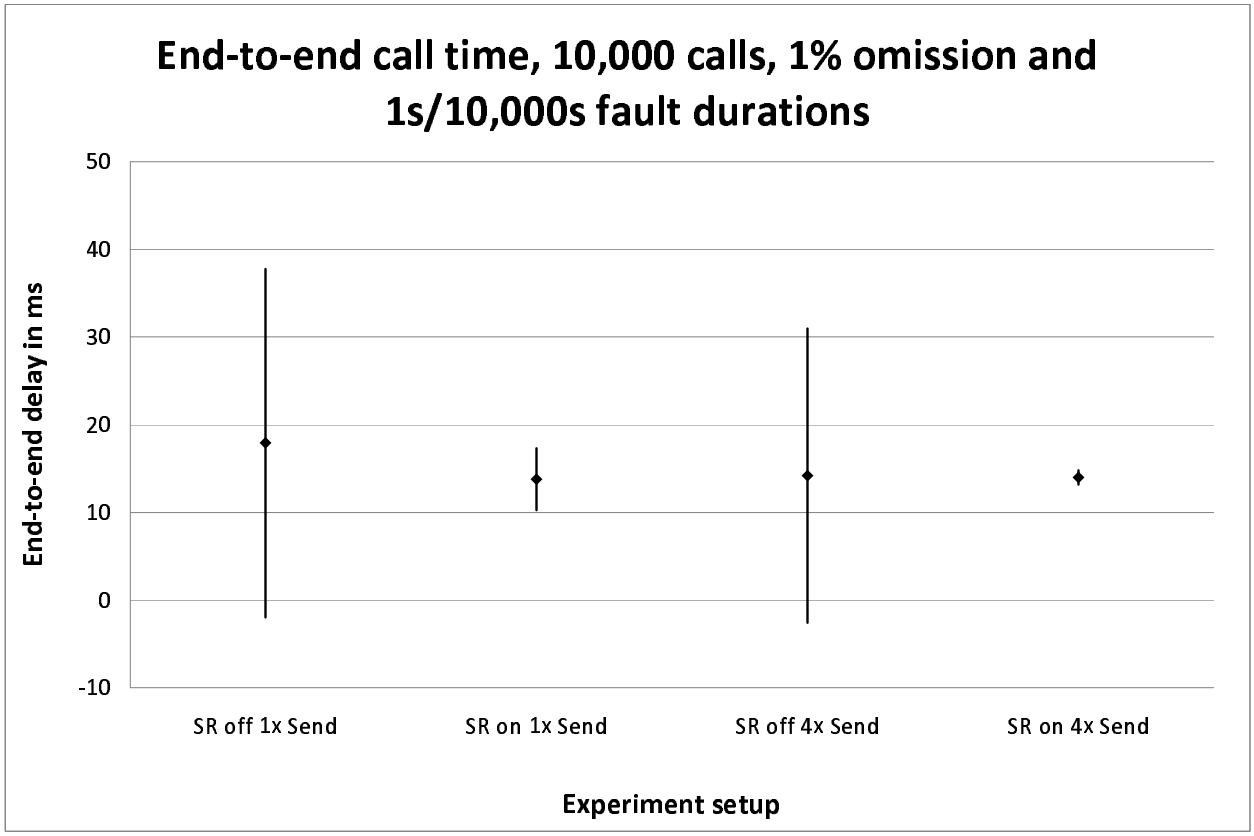


Figure 5.6: Average Calltimes for Various Redundancy with Full Loss

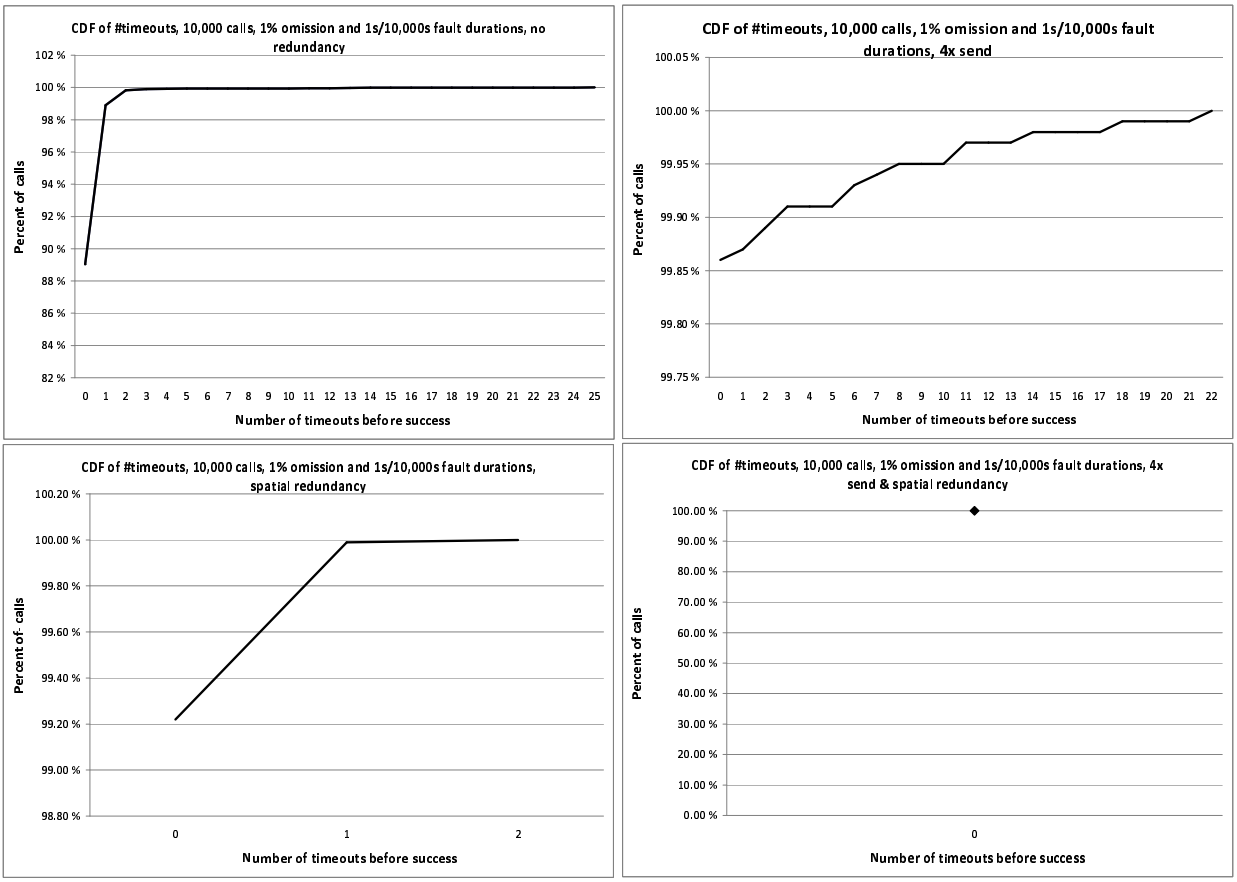


Figure 5.7: Cumulative distributions of number of timeouts per call

CHAPTER SIX

CONCLUSION AND FUTURE WORK

6.1 Concluding Remarks

Existing power-grid control mechanisms are inflexible, incompatible across vendors, and designed for highly centralized environments, and cannot provide for the next generation of power grid communication. While much work is done in using web services and other existing control mechanisms, such control mechanisms built directly on top of existing network protocols will not overcome the incompatibility issues with proprietary protocols and may have to compromise safety and timeliness depending on the underlying network.

This thesis provides the architecture, design and evaluation of Ratatoskr, an extension to the GridStat status dissemination system with the design and implementation of a timely and reliable remote procedure call mechanism running on top of two one-way GridStat subscription paths. This extension complements GridStat's capabilities as a middleware data acquisition system with the ability to send control messages to power-grid actuators.

Ratatoskr is split into two subsystems, a transport protocol for two-way communication over the GridStat network, the 2WoPS protocol, and an RPC mechanism built on top of the 2WoPS protocol, the Ratatoskr RPC mechanism. The 2WoPS protocol utilizes the QoS semantics provided by the GridStat network to offer maximum delivery delay guarantees and spatially redundant network paths for sends. Further, two additional redundancy techniques are used; temporally redundant sends and ACK/retires. While the 2WoPS protocol was implemented specifically for the RPC mechanism, it is also used in another GridStat project that gained from the timeliness and reliability provided by the protocol. The Ratatoskr RPC mechanism provides extensive customization of the redundancy levels for each call, providing a tradeoff space between timeliness, use of network resources, and reliability. Futher, pre- and post conditions built into the call semantics provides

additional safety mechanisms for application designers, and creates a building block for a uniform middleware safety framework for inter-vendor operations.

The evaluation explores the effectiveness of the redundancy techniques in the face of two fault-models, omission faults with uniform drop probability and durations of total link failure. The experiments show that both spatial and temporally redundant resends provide a polynomial reduction in end-to-end fault occurrence rates for omission faults, with a a temporal redundancy of two providing a square reduction, a temporal redundancy of three providing cube reduction and so on. The spatial redundancy shows a similar pattern with two redundant paths providing a square reduction, but as the current implementation of GridStat does not support more than two redundant paths, a conclusion on the developments of the pattern with more paths could not be established beyond the mathematical models. Temporally redundant sends proved of little value in the face of link failure durations, but spatial redundancy provides a square reduction also here. The experimental results match the expected results from the analysis closely. A comparison between Ratatoskr and a simulated traditional RPC mechanism without temporal and spatial redundancy, shows that Ratatoskr profits from the redundancy with a lower average for end-to-end call times and a considerably tighter call time distribution.

6.2 Future Work

6.2.1 Long Term Connections

GridStat is designed with a relatively stable network topology in mind, where most data is produced by permanent, physical devices and consumed in a long term monitoring fashion. GridStat uses the stability to trade longer connection setup times for better connection QoS, and the connection time is likely to be high and unpredictable. For Ratatoskr, this implies that connections must be established well ahead of performing delay-sensitive calls. This further implies that many RPC connections will be long term, and it is likely that permanent connections will be established as RPC channels between locations that frequently issue control commands and field sites.

For many such longer term connections, utilization is likely to be relatively infrequent, and traffic levels relatively bursty. Currently, the routing techniques used in GridStat are optimized for rate based traffic. It is undesirable to force bursty RPC traffic to follow a rate based traffic pattern, as the rate would either have to be high enough to accommodate traffic bursts, which would waste bandwidth between bursts, or the RPC call would have to send packets at a low rate which could cause delays. The handling of control and router scheduling for bursty and rate based traffic in GridStat requires further research.

No matter the solution chosen for handling bursty traffic in GridStat, some sort of restrictions is likely to be placed on the traffic patterns used in Ratatoskr. The RPC mechanism must adhere to these restraints to avoid packet loss due to traffic policing. This will delay sends and thus affect delivery deadlines. Further work is needed to reflect the effects on deadlines into call semantics and syntax.

6.2.1.1 Shared Connection

One strategy that could reduce some of the impact of traffic policing would be to establish a bulk channel between client and server sites used by several control processes issue commands. One example of this would be several protection schemes in a control center sharing a single RPC connection to a substation, with a single server in the substation forwarding commands to individual actuators. Such a bulk channel could reduce the maximum bandwidth provisioned for bursts when compared to having separate RPC connections for each control process, based on the expectancy that it is unlikely that all processes will invoke RPCs at the same time. If such a scheme is used, some kind of prioritization scheme is required to handle the cases where so many RPCs are invoked at the same time that the burst bandwidth limitations are exceeded.

6.2.2 Fault Tolerance Level Calculation

In the prototype implementation, the levels of fault tolerance for each call must be set by the application using Ratatoskr. This is cumbersome, as it requires that the programmer has some

knowledge of the network loss properties of the GridStat path to the server. Further, hard-coded fault-tolerance parameters could become outdated with changes to network topology. Given abstract network loss properties of the path and fault tolerance requirements for the call, Ratatoskr would be able to periodically calculate the appropriate level of redundancy, balance this to the network resource consumption and adjust fault-tolerance parameters. Network loss properties could be obtained by regular measuring, performed either by Ratatoskr itself or by the GridStat management hierarchy.

6.2.3 Extensions to the 2WoPS Protocol

While the 2WoPS protocol was designed especially for use with the Ratatoskr RPC, there could be several other uses for two-way traffic over a GridStat deployment. This section lays out some of the improvements that could support such other uses.

6.2.3.1 Packet Size

Packet sizes in GridStat are currently limited to the largest size supported by the underlying network. Ratatoskr was designed and implemented as a prototype for evaluation and experimental use, and no steps were taken to increase the packet size. This could affect the usability of the final version of Ratatoskr, and increased packet sizes in the 2WoPS protocol could open up new uses. Larger packet sizes can be achieved by splitting TSDUs over several TPDUs, using sequence numbers to identify the position of each TPDU in the TSDU and buffering each TSDU until all corresponding TPDUs are received at the server. Another solution would be to implement TSDUs as streams of data where each section of the stream is immediately delivered to the application given that the previous section of the stream has been delivered before, but this approach is undesirable for RPC calls because of the mechanism's message-oriented nature and should be provided as an addition to the existing message service.

6.2.3.2 Group Communication

GridStat supports delay-bounded multicast from a single publisher to multiple subscribers using spatially redundant paths to each receiver, [14]. This may be used to provide highly reliable group communication in the 2WoPS protocol.

6.2.4 Extensions to the RPC Mechanism

Ratatoskr was implemented as a prototype for evaluation and experimentation, and many of the features of a final version were omitted to limit the scope of the thesis. This section details some of these omissions.

6.2.4.1 Peer Failure Handling

GridStat does not take note of failed peers, and the subscription paths for such peers will remain open. This should be remedied by having 2WoPS peers ping peers they share a connection with, and close a connection if the other peer is registered as failed for a long period. Further, failed RPC peers that are restarted should be able to re-establish connections that were open before the peer failed. To reuse existing GridStat paths, variable and subscription IDs must be stored to stable storage and loaded at restart, together with packet filtering data to retain the at-most once delivery guarantees. A handshake protocol to establish the next sequence number and any undelivered calls should be used. Further, the server call repository would have to be stored so remote calls would still be accessible upon reconnection.

6.2.4.2 Replicated actuators

Replication of servers in the client/server model is a common technique for achieving fault tolerance against a variety of failures, including server system failures. There are several approaches towards how a service may be replicated and how failures are handled. The two main distinctions are active replication where calls are carried out by all replicas, and passive replication where a single server actively carries out calls while replicated servers remain passive until a failure in the master is detected and a passive replica is activated as the new master. The approaches to these

schemes and several hybrid schemes provide a tradeoff space between performance, flexibility and communication requirements.

As actuators in the grid are mechanical devices, one redundancy approach might not necessarily fit all applications. For example, if replicating a transformer mechanism as two transformer actuators in series, active replication would make both transformer actuators adjust the voltage, and the actual voltage adjustment would be twice the intended. Some sort of passive replication would correctly adjust a single transformer actuator. If replicating a set of protective circuit breakers on the same line, with passive replication the time to detect a failure in the master and do a failure passover to a replica may be too long for protective schemes, and so active replication might be the only replication-style that allows for a quick enough response. To accommodate a variety of such situations, replication in Ratatoskr should provide several replication techniques. Replication would gain considerably from the reliable group-communication provided by GridStat, but the delivery guarantees provided may have to be strengthened further. Research must be done into how to strengthen guarantees while retaining real-time properties, and which replication schemes are appropriate for grid operation.

6.2.4.3 Caller replication

There is an increasing trend among electric utilities to set up backup control center facilities for maintaining operation in the cases where disrupting events affect the main control center, [13]. Future work might include researching how replication of caller applications affect the RPC system, how to synchronize call state, and how to handle failure passover in replicated caller programs during call execution.

6.2.4.4 Extensions to Pre- and Post-conditions

The pre- and post condition semantics implemented in the prototype were designed to serve as a proof-of-concept for the advantages of designing such functionality into the RPC semantics. Several extensions are needed to provide the full potential of these safety measures. One extension

that would greatly enhance their semantics would be shared state between the predicate modules, across pre- and post conditions and across client and server modules. The need for such semantics and their design is still a matter of future research. Another extension that would enhance modularity and reusability would be to provide a mechanism for a libraries of conditional expressions, where each expression contained a map of input variables with predefined roles in the expression that could be connected to specific published GridStat variables by application designers. An example of this would be a pre-condition for determining whether a transformer will overheat if it is energized, with a standard input variable for sensor readings of the internal temperature of the transformer. In addition to enhancing reuse, such libraries could provide a standard set of conditional modules that would ensure uniform safety semantics across vendor and power utility.

6.2.5 Security

The security requirements for Ratatoskr is outside the scope of this thesis, but it is essential to provide a high degree of security in a control mechanism used in a critical infrastructure such as the power-grid, and security measures must be addressed before a deployment. Ongoing work on securing GridStat will eventually, among other features, supply data-plane communication paths with integrity and confidentiality. These security mechanisms will be administered from the management plane, and will require that entities accessing the security management mechanism authenticate themselves with keys pre-loaded by GridStat maintenance personnel. While these properties will greatly help towards securing Ratatoskr, communication-level security will not protect against misuse from compromised devices with pre-loaded keys. At the very least, some sort of authentication mechanism should be built into the RPC mechanism. Such an authentication mechanism could provide group and user policies on access to RPC calls. Further, semantics could be added for differentiating which pre- and post-conditions are executed for each call for users or groups. This would enable utilities to retain emergency accounts that override safety mechanisms

that could interfere with critical operations that go outside normal usage patterns. Finally, the prototype was designed without considerations for byzantine behavior. This must be addressed before a final version, as at the very least malformed messages or sequence number manipulations could lead to unexpected behavior.

6.2.6 Future Evaluations

The current GridStat implementation is a prototype under constant development. Future versions will extend current functionality used by Ratatoskr. These extensions, in addition to access to new computing resources, would allow a more extensive evaluation of the capabilities of the fault-tolerance techniques used in Ratatoskr. In addition to evaluating a wider range of fault model parameters, an extension to the maximum number of redundant paths, which currently is limited to two, would be able to verify the expected results put forth in this thesis. Further, additional network fault models could be introduced, such as modeling drop rates on each link as a Markov chain where the state of the link decides a uniform drop rate. This model could represent periods of interference for wireless links, while using a single state for multiple links could represent common mode failures such as high levels of network traffic during DDoS attacks on the network and following congestion in bottleneck links.

BIBLIOGRAPHY

[1] S. F. Abelsen. Adaptive gridstat information flow mechanisms and management for power grid contingencies. Master's thesis, Washington State University, Pullman, Washington, USA, August 2007.

[2] D. E. Bakken, C. H. Hauser, H. Gjermundrød, and A. Bose. Towards more flexible and robust data delivery for monitoring and control of the electric power grid. Technical report, School of Electrical Engineering and Computer Science, Washington State University, 2007.

[3] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39 59, 1984.

[5] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design, fourth edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[6] EPRI/CEIDS. The integrated energy and communication systems architecture, volms i-iv, July 2004.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[8] K. H. Gjermundrød. *Flexible QoS-managed status dissemination middleware framework for the electric power grid*. PhD thesis, Washington State University, Pullman, Washington, USA, August 2006.

[9] O. M. Group. The common object request broker: Architecture and specification. Technical report.

[10] O. M. Group. Fault tolerant corba specification v1.0. Technical report, Object Management Group, April 2000.

[11] O. M. Group. Realtime corba specification 1.2. Technical report, Object Management Group, January 2005.

[12] C. Hauser, D. E. Bakken, and A. Bose. A failure to communicate: next generation communication requirements, technologies, and architecture for the electric power grid. *Power and Energy Magazine, IEEE*, 3:47–55, March-April 2005.

[13] T. Heidrick, J. Mossing, and G. Ashfaq. Calling for backup, the importance of, and key design issues for, backup control centers in maintaining power system reliability. *Power and Energy Magazine, IEEE*, 2(1):114–131, 2004.

[14] J. N. Helkey. *Low-cost delay-constrained multicast routing heuristics and their evaluation*. PhD thesis, Washington State University, Pullman, Washington, USA, August 2006.

[15] J. N. Helkey. Achieving end-to-end delay bounds in a real-time status dissemination network. Master's thesis, Washington State University, Pullman, Washington, USA, May 2007.

[16] G. Iannaccone, C.-n. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an ip backbone. *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, page 237 242, 2002.

[17] IEC. Iec 61850 communication networks and systems in substations.

[18] S. Iren, P. D. Amer, and P. T. Conrad. The transport layer: tutorial and survey. *ACM Computing Surveys*, 31(4):360–404, 1999.

[19] S. Lee, Y. Yu, S. Nelakuditi, Z.-L. Zhang, and C.-N. Chuah. Proactive vs reactive approaches to failure resilient routing. *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE INFOCOM*, 1:176 185, 2004.

[20] S. Maffeis. Adding group communication and fault-tolerance to corba. In *COOTS'95: Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 10–10, Berkeley, CA, USA, 1995. USENIX Association.

[21] R. Marasli, P. D. Amer, and P. T. Conrad. Retransmission-based partially reliable transport service: An analytic model. In *INFOCOM (2)*, pages 621–629, 1996.

[22] A. D. McKinnon. *Supporting fine-grained configurability with multiple quality of service properties in middleware for embedded systems*. PhD thesis, Washington State University, Pullman, Washington, USA, December 2003.

[23] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the eternal system. *Theor. Pract. Object Syst.*, 4(2):81–92, 1998.

[24] T. Nakajima. Dynamic transport protocol selection in a corba system. In *Object-Oriented Real-Time Distributed Computing, 2000. (ISORC 2000)*, pages 42 – 51, march 2000.

[25] Y. J. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri. Aqua: An adaptive architecture that provides dependable distributed objects. *IEEE Trans. Comput.*, 52(1):31–50, 2003.

[26] D. C. Schmidt and F. Kuhns. An overview of the real-time CORBA specification. *Computer*, 33(6):56–63, 2000.

[27] D. C. Schmidt, D. Levine, and S. Mungee. The design and performance of real-time object request brokers. *Computer Communications*, 21(4), 1998.

[28] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):246–260, 1982.

[29] A. S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[30] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, November 1994.