Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Maria Peltola

Implementing reliability and redundancy in a time critical telecommunication system

Master's Thesis
Espoo, November 16, 2015

Supervisor: Professor Heikki Saikkonen
Advisor: Miia Valtonen M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Maria Peltola

Title:
Implementing reliability and redundancy in a time critical telecommunication system

Date: November 16, 2015 **Pages:** vii + 108

Major: Software technology **Code:** T-106

Supervisor: Professor Heikki Saikkonen

Advisor: Miia Valtonen M.Sc. (Tech.)

Different distributed systems have different requirements for reliability. One way for increasing reliability is fault tolerance. Methods for increasing fault tolerance have been studied for decades, starting from early hardware level fault tolerance until more recent studies for peer-to-peer and cloud computing fault tolerance.

In this paper, fault tolerance of an existing telecommunication service platform is studied and improved. Even in the case of a failure of a single server, called a call handler, any data should not be lost. Three sub-problems with different expectations are presented: failure detection, data dissemination and takeover. The implemented failure detection protocol is based on a basic gossip-style heartbeat protocol. The data dissemination protocol is a gossip-style dissemination protocol which, unlike traditional gossip algorithms, does not select its targets fully randomly. Instead, known data about which call handlers have received and might have received a message are used when selecting targets. On top of both, a simple takeover protocol is implemented.

Testing was done in a closed environment both for the failure detection and for the data dissemination, as well as for whole the system. The results show that failure detection protocol is able to provide adequate detection times and accuracy. The nature of the implemented data dissemination algorithm is very spamming by the results. The amount of sent messages can, however, be greatly decreased with design decisions.

Test results for whole the system indicate that the system is able to provide over 99 % reliability even with large server crash probabilities at least up to call handler amount 16. Unfortunately, resources in the testing environment were limited and that is why memory problems started to occur affecting the test results starting from 17 call handlers. That is why larger call handler amounts could not be tested.

Keywords: gossip, failure detection, data dissemination, heartbeat algorithm, fault tolerance, reliability

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Maria Peltola		
Työn nimi:	Luotettavuuden ja toisteisuuden toteutus aikakriittiseen televiestintäjärjestelmään		
Päiväys:	16. marraskuuta 2015	**Sivumäärä:**	vii + 108
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Heikki Saikkonen		
Ohjaaja:	Diplomi-insinööri Miia Valtonen		

Erilaisilla hajautetuilla järjestelmillä on eri vaatimuksia luotettavuudelle. Eräs tapa parantaa luotettavuutta on vikasietoisuuden kasvattaminen. Erilaisia tapoja parantaa vikasietoisuutta on tutkittu jo vuosikymmeniä. Aikaisimmat tutkimukset keskittyivät pääasiallisesti laitteistotason vikasietoisuuteen, mutta viime aikoina päämielenkiinnonkohde on siirtynyt vertaisverkkojen ja pilviteknologioiden vikasietoisuuden tutkimiseen.

Tämän opinnäytetyön tarkoituksena on tutkia ja parantaa olemassaolevan televiestintäjärjestelmän luotettavuutta vikasietoisuudella. Päämääränä on toteuttaa ratkaisu, joka pitää huolta ettei tietoa katoa puhelunkäsittelijöiden vikatilanteissa. Varsinainen ongelma on jaettu osa-alueisiin: virheen havaitsemiseen, tiedon levittämiseen ja haltuunottoon. Virheenhavaitsemisprotokolla pohjautuu juorutyyliseen sydämenlyöntiprotokollaan. Tiedon levittäminen on toteutettu juorutyylisellä tiedonlevitysprotokollalla, joka ei valitse lähetyskohteitaan täysin satunnaisesti, toisin kuin perinteinen juoruprotokolla. Sen sijaan valinnassa käytetään hyväksi tietoa verkon jäsenistä, jotka ovat jo vastaanottaneet viestin tai ovat mahdollisesti vastaanottaneet viestin. Puheluiden haltuunottoprotokolla on toteutettu hyödyntäen edellisiä protokollia.

Testaus toteutettiin suljetussa ympäristössä. Erilliset testit suoritettiin virheenhavaitsemis- ja tiedon levittämisprotokollille. Lisäksi tehtiin koko järjestelmätason testejä. Virheenhavaitsemisprotokolla näyttää tulosten perusteella tarjoavan riittävän hyvän havaitsemisajan ja -tarkkuuden. Tulosten perusteella tiedonlevittämisprotokolla luo paljon viestejä, mutta viestimääriin pystyy vaikuttamaan protokollasuunnittelulla.

Koko järjestelmän testit viittaisivat siihen, että yli 99 % luotettavuus saavutetaan jopa suurilla kaatumistodennäköisyyksillä ainakin 16 puhelunkäsittelijään asti. Testiympäristön resurssit olivat rajoitetut, mistä johtuen jo 17 puhelunkäsittelijän testeissä esiintyi muistiongelmia, jotka vaikuttivat tuloksiin. Suurempia jäsenmääriä ei siksi pystytty testaamaan.

Asiasanat:	juorualgoritmi, virheenhavaitseminen, tiedonlevitys, sydämenlyöntialgoritmi, vikasietoisuus, luotettavuus
Kieli:	Englanti

Acknowledgements

First of all, I want to thank Aalto University and the supervisor of this master's thesis, professor Heikki Saikkonen, for providing the opportunity to do the thesis.

I am also thankful to NSF Telecom Ab for providing the base for the interesting subject.

Finally, a special thank you goes to my wonderful fiancé, who has provided mental support during the writing process.

Espoo, November 16, 2015

Maria Peltola

Contents

Chapter 1

Introduction

In today's world, more and more devices are becoming a part of a growing network of smart devices. This trend means that while communicating cars, fridges or coffee makers were only science fiction not too many years ago, nowadays smart devices are very much feasible and growing amount of product development is done for creating them. However, intelligence, let it be human or device intelligence, often requires information outside from what the intelligent object itself can sense. For fulfilling the need of external information, methods and protocols for communication are needed.

But there are many cases in which it is not enough to know only methods of communication. Capability to know if messages have reached their target or to use another service when one is unavailable might be essential for a system to fulfill its purpose. This is the case especially in systems of one growing market: telecare services. If considering, for example, a system in which a machine sends to a center information about a person's heartbeats, it is crucial to be sure that the information reaches the target, or else the lost messages might be a matter of life and death, literally. This is where reliability studies step in.

Reliability has been studied for decades on several levels for different kind of computing systems. Earliest studies mainly featured hardware level and low-level network fault tolerance, and the most recent studies have largely been examining cloud systems and peer-to-peer algorithms. There are numerous different kind of faults that can be disastrous in systems. While the systems are still becoming more and more complex, the amount of possible failures raises together with the system complexity. Furthermore, the diversity of distributed systems is wide. Systems have their own specifications requiring tolerance of certain kind of failures and certain kind of behaviour in the case of failures. Therefore, a single "one-solution-fits-all" is difficult, if not possible, to find. Instead, solutions are usually requirement specific.

1.1 Objectives

The main objective of this paper is to study how reliability of an asynchronous, time critical system can be improved by increasing fault tolerance of the system. The base method for increased fault tolerance lies in replication of processes. This work examines three major questions:

1. How network member failures are detected, and which components should be responsible of failure detection?
2. Which structures and protocols provide the expected reliability for data spreading?
3. How a system should behave in the case of failures?

An already-existing telecommunication service platform is used for examining these questions. The system provides the actual system specifications and expectations through which possible fault tolerance methods are evaluated. A solution is then implemented and tested for the example system.

1.2 Structure

Chapter 2 introduces the bases for this work. The most relevant terms are defined, the example system is described and the objectives are stated. Chapter 3 presents some of studies about fault tolerance techniques in general, and different failure detection techniques are viewed in the Chapter 4. Fault tolerance of the example system is then improved, and the solutions is explained on a theoretical level in Chapter 5. Practical implementation of protocols and the structure of the implementation are then presented in Chapter 6. Implementation is tested both with whole system tests and with tests for data dissemination and failure detection protocols separately. Test results are presented in Chapter 7 for the protocol specific tests and in Chapter 8 for the system tests. Finally, the test results and the used mechanisms are analyzed in the Chapter 9.

Chapter 2

Current system and methodology

This chapter first defines some of the most important terms. Later, the example system is introduced by shortly explaining its purpose and the current architecture. Then it is examined what are the biggest problems with the current system architecture, and how to try to solve them.

2.1 Definitions

This section defines terms as they are used in this paper. The relationship between different terms can be seen in Figure 2.1. Two main terms covering the subject of this paper are *correctness* and *dependability*. They both consist, among others, of *reliability* and *availability*, which are the main goals in this paper. The figure presents correctness and dependability as unconnected properties, however, correctness could be considered to be a property of dependability. Undesired circumstances which should be avoided or handled are *faults*, *errors* and *failures*. Means for handling these circumstances include validation, meaning how to reach confidence; and procurement, meaning how to provide the ability to deliver a service correctly. [41]

Fault tolerance provides methods for a service to continue working by its specification even in the presence of faults. *Fault avoidance* consists of methods for preventing faults by construction. *Error removal* strives to remove errors with verification if they occur. *Error forecast* aims to estimate the presence or consequences of error occurrences. It is noted in [41] that dependability is only achieved with a combination of these means. All faults are not predictable and preparing for faults in fault avoidance requires that all possible faults are known in advance. Also, all errors can not simply be

removed, because in some cases an action is required from the system administrator, for example in hardware problems. Faults cannot be tolerated or removed if they are never detected by evaluation. [41]

The two means for reliability examined in this paper are replication as a method for fault tolerance, and failure detection as a method of error forecasting.

2.1.1 Dependability and reliability

Dependability is a collective term which defines the quality of service. The criteria for the quality is that the service should work by its specifications and process required actions for a defined duration [35, 41]. Dependability can be described and quantified through several sub-factors, and there are different ways for classifying the sub-factors. In International Electrotechnical Commission Technical Committee (IEC TC) 56 standards [36] the sub-factors are defined to be reliability, *maintainability* and *maintenance support*. Along [41], reliability and availability are the two main measures of dependability. Reliability is presented as a measure for continuity of a service; the system continuously processes correctly for the expected duration. Availability is described as a presentation of readiness for correct service. In addition to these two, [4] presents some more factors: *safety*, meaning there are no catastrophic consequences from actions or environments; *integrity*, meaning there are no improper system alterations; and maintainability, as capability and readiness for modifications and repairs.

Reliability has been defined in Online Electrotechnical Vocabulary of International Electrotechnical Commission [35] to mean the ability of some system to perform required actions correctly for a given time interval.

2.1.2 Time criticality

In [37] the most important characteristics of a time critical system are ability to satisfy the time constraints and capability to guard the system against faulty execution. In this paper the time constraints define that a response to any message is expected to be received in a moderate time. Time that it takes to produce a response should not take more than several seconds, maximum. Exact time limits do not exist, however, the longer it takes, the more probable it is that an end user gets tired of waiting and thus the system has failed to serve its purpose.

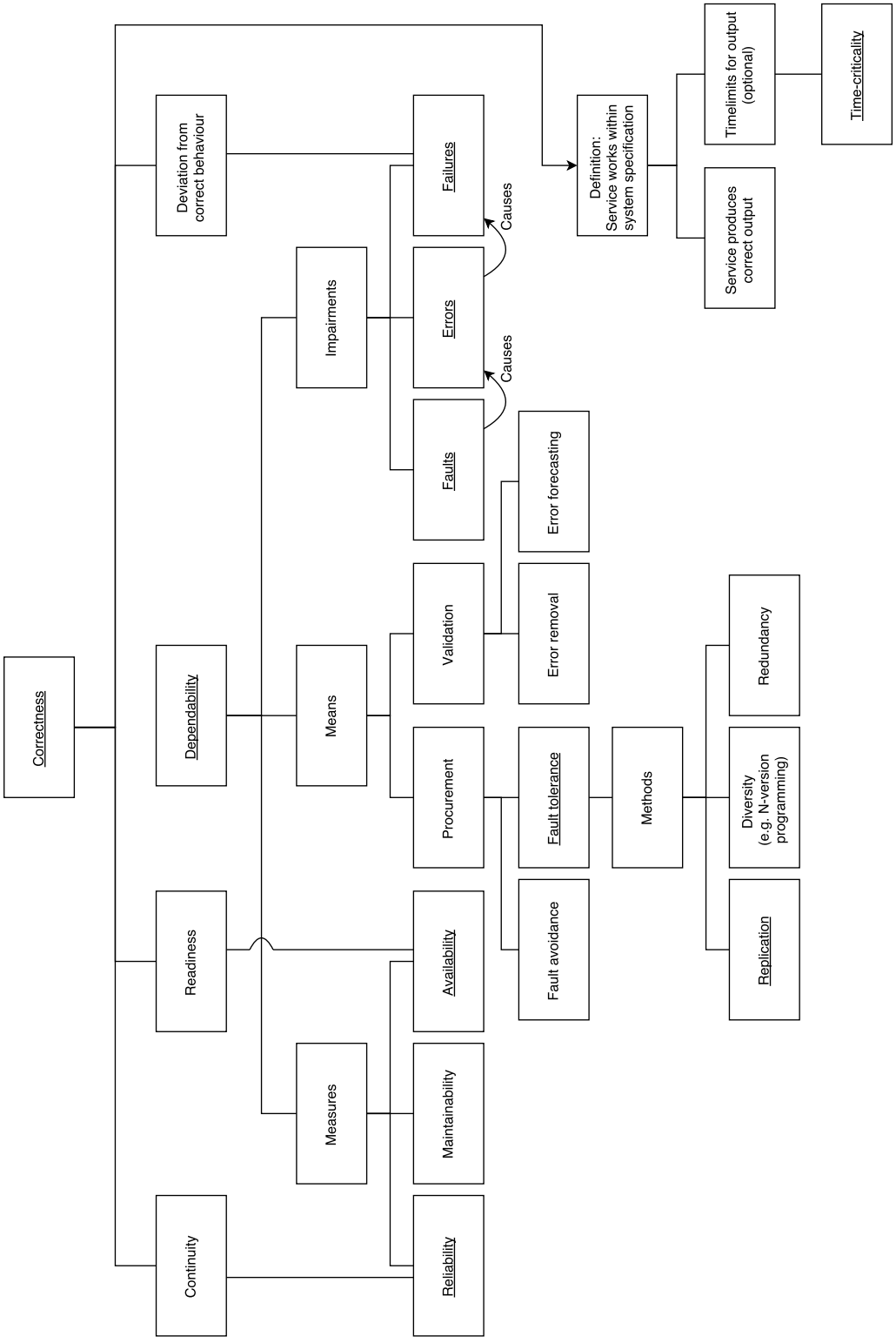

Figure 2.1: Relationships of used terms. Each term defined in this chapter is underlined. The tree is based on dependability tree presented in [41].

2.1.3 Correctness, failures and faults

In [13] correctness is used to describe that a system behaves along its service specification. A system is correct as long as its system specification is followed. Terms fault and failure can then be defined through correctness to be actions, happenings or changes in an environment such that without any means of specific fault handling, the system is not correct anymore.

The definitions of terms fault and failure are partially related. In [22] they are presented as that fault leads to an incorrect state by specification, and failure "restricts the system to perform its required functions". In [41] the difference between failures, faults and errors is presented explaining the relation between the terms: failures are events that affect services, errors are system internal undesired events, and faults are undesired properties or events that might raise errors and later failures. Also, an example of the difference is given: designer's mistake is one kind of a fault, which then causes an error (i.e. erroneous computation) which results in failure (i.e. wrong calculation result is used, which leads to unexpected results). Definitions by IEC 60050, electrotechnical vocabulary [35] define a failure as a termination of correct performance of required functions, and a fault as a state in which a service is unable to perform required functions.

There are a huge amount of different kind of faults and failures. That is why classifying them helps in defining which kind of faults and failures are expected to occur. In researches, different kind of classifications are presented. In [25] and [41] they represent many factors through which faults can be classified according to different view-points. The most relevant ones for this paper are divisions between physical, design and interaction faults, and whether it is a permanent fault or a temporary fault.

Faults can also be classified based on what are the consequences, that is, which kind of failure it causes. In [22] a simple classification into two different types of failures is done. They present *fail-stop* failures in which the processing ends or becomes prevented, and *byzantine* failures, in which processing might produce incorrect output.

Another classification for failures is presented in [5]: *crash*, *send-omission*, *general omission* and *arbitrary* failures. A crash failure is an event in which after the first failure, the system fails to produce output indefinitely, until the system is restarted or the root cause for the fault is otherwise fixed [13]. Omission failures are problems in communication. In send-omission failures, a system fails to produce and send a response, and in general omission failures the system fails to read, and thus also to output the expected result [5]. The most varying failures are arbitrary failures, also called Byzantine failures, in which the behaviour is unexpected and incorrect, but it might not be seen by

an outside observer because messages might not get lost. The system might, for example, due to some calculation errors, produce unexpected output. [5]

2.1.4 Fault tolerance

Fault tolerance means that the system is able to tolerate some certain specific classes of failures so that when a fault is observed, the system works as its failure specification describes. Fault tolerance is one of four methods for achieving dependability, the other three being fault-avoidance, as how to prevent faults; error-removal, as how to minimize the presence of occurred errors; and error-forecasting, which tries to estimate occurrences and consequences of errors. [41]

In [3] different types of fault-tolerance are introduced based on how a system should behave in the presence of faults: *masking*, *non-masking* and *fail-safe* fault-tolerance. Masking fault-tolerance is the strictest: a process should continue working by its specifications. In non-masking fault tolerance, in the case of a failure, a process might be perturbed and for a while it is not correct, however it then recovers back to correct. Fail-safe fault-tolerance is the least strict: when a failure occurs, a process is still correct as per the minimal safety specification, but it might enter into a state in which it is not correct as per whole the system's specifications. The process might, for example, lose liveliness. [3]

In [45] fault-tolerance is divided into three categories following the division of a system into a three-level hierarchy: application level, operating system level and hardware level. Hardware level fault tolerance most often is achieved by pure redundancy: using several of same kind of components. Operating system fault-tolerance might be used for example for masking disk controller failures. Application level is the highest abstraction level of the presented, and there are several ways for implementing toleration also for lower level programs: redundancy, masking, atomicity, fault-tolerant algorithms etc. In [22] even higher system level is mentioned: architectural level. While application level examines one process and errors and exceptions happening in the scope of the process, architectural level examines several processes and their interaction.

2.1.5 Replication

Replication is a fault tolerance method in which one or more replicates of a process are created in a multi-process system. All replicates of a node know the same data so they all are able to do the same action with the same output [23].

Viewpoint	**Perspective**	**Explanation and examples**
Phenomenological cause	Physical	Some hardware part in a system stops working, for example broken processor
	Human-made: design	Failures caused by programmers, such as bad choices in algorithms, some special cases are not considered
	Human-made: interaction	Failures caused by users and programmers, user does something which is not accepted and which is not taken into account by programmers
Persistence	Permanent	Action is required by admin or other processes in order to recover from the failure, for example broken hardware
	Temporary	The failure might disappear without anyone or anything intervening, for example temporary congestion in network causing packet drops
Consequences	Crash failures	Process stops working altogether, for example caused by too long power outage
	Send-omission failures	Process fails to send a response, for example caused by route to the target not found
	General omission failures	Process fails to receive, and thus also to respond, for example caused by connection breakage while reading
	Arbitrary failures	Any failure which might not cause crash or omission, for example because of some bit errors, the result of calculations and thus the response from a process is wrong and unexpected, or a message of wrong form

Table 2.1: Possible faults classes, combination of the most important viewpoints of Laprie [41] together with the failure definitions presented by Bazzi&Neiger [5]

There are several replication techniques which can be classified in different ways. Homogeneous and heterogeneous replications are presented in [23]. In heterogeneous replication, replicates' implementations differ, however, they have been implemented using the same specification, thus they are functionally the same. Replicates in homogeneous replication are identical copies of each others at the beginning of the system execution. Replication techniques can, additionally, be classified depending on how the replication happens and what are the roles of replicates.

- **Active replication**, also called state-machine approach, is a technique in which all replicates are equal in a hierarchy level, playing the same role without centralized control. All replicates are synchronized, and they process requests parallel. All of the replicates also return responses. [29, 65]

- **Passive replication**, also called primary-backup strategy, means replication in which only one of the replicates is working as the primary. The primary receives requests and processes them. Then the primary sends update messages to other replicates, called backups. Backups return an ACK message to the primary, and after having received ACKs from all living backups, the primary returns the response. [29, 65]

- **Semi-active replication**, also called leader/follower replication, is a term used for replication technique in which replicates have different roles: there is one primary and others are backups. However, when receiving a request, all replicates process them independently, but only the primary returns a response to the request sender. [65]

- **Semi-passive replication** is a technique for lowering the reconfiguration costs of passive replication in the case of the failure of the primary replicate. In semi-passive replication, there is one primary replicate, others being backups. The primary receives requests and updates backups, which then send ACKs as responses back to the primary. A major difference between semi-passive and passive replication is that the primary is selected again for each client request, based on rotating coordinator paradigm. [17]

2.2 Introducing the case: a telecommunication service platform

The base for the architecture and protocol selection in this paper is an existing telecommunication service platform. This section presents the system and its architecture. Additionally, the objectives of this paper are explicated.

2.2.1 High level - Basic functionality

The starting point for this work is a telecommunication service platform called Tempo, developed by a telecommunication company named NSF Telecom Ab. The system offers a wide amount of different communication channels:

- Voice calls
- Short message service (SMS) message sending
- Unstructured supplementary service data (USSD) message sending and receiving
- System integration over simple object access protocol (SOAP) and Representational State Transfer (REST)

Event data record (EDR) creation for these communication channels is provided for monitoring and billing purposes. Some of the most important features of the system are presented in Table 2.2.

The aim for the recent development has been creating a system for critical communication, which creates more importance for the system reliability and availability.

2.2.2 Architecture of the example system

The system consists of different architectural components presented in Figure 2.2 each having a different function. All persistent data needed by the system is saved in a replicated database, to which all components have an access.

Media gateway is an element which is connected to the outer world: public switched telephone network (PSTN), mobile network and voice over internet protocol (Voice over IP, VoIP) trunks. Communication to outer world is done using session initialization protocol (SIP) for signaling and Real-time Transport Protocol (RTP) for transferring voice or video data. Additionally,

Feature	Description
SMS, USSD, Voice channels	Different channels for transferring data/voice
Call screening and barring	Which calls are allowed and how different call cases should be handled
Speed dial calls	Short numbers which work within a defined group as aliases for other, usually longer, phone numbers
Call routing, forward and transfer	Calls can be forwarded or transferred to another number than the original called number either calendar based, as a result of unanswered call, as a result of do not disturb feature or during a call
Calling line identification restriction and presentation	Anonymous caller numbers, and showing other (allowed) number other than the actual number of the calling phone
Group calls	One or more numbers behind one group number are called parallel or sequentially, when the group number is called
Interactive Voice Response (IVR)	Automatic menu with announcements which allows the caller to select the next action using DTMF
Call queues and attendant	Music on hold is played for a caller until an attendant picks up the call from a queue for answer or forward
Intelligent machine-to-machine message delivery	Message queues with retry policy in the case of message delivery fails
Provisioning	Records from each call and message delivery are created for provisioning purposes
Priority calls	Allowing high priority calls to go through even if otherwise the maximum allowed amount of concurrent calls is achieved

Table 2.2: Some of the most important features of Tempo

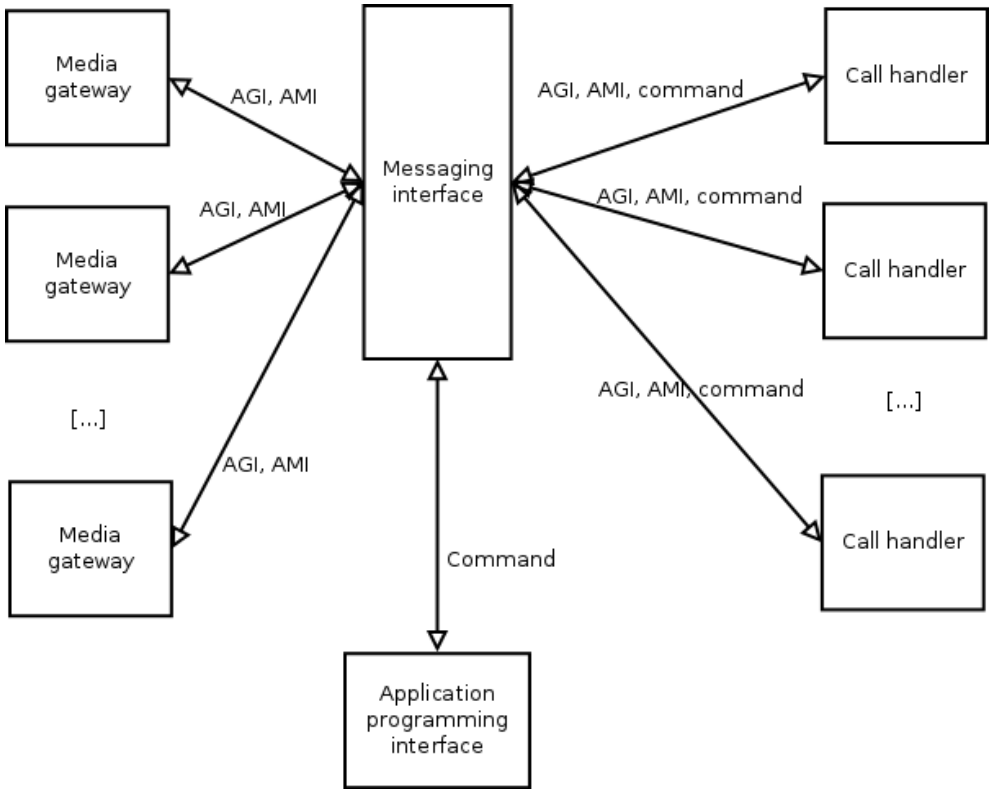

Figure 2.2: The component of the example system, showing $N \geq 3$ call handlers. Communication between call handlers is not presented.

a media gateway provides a VoIP proxy, which handles registrations for VoIP clients known by the system. Media gateways also take care of channel actions of all voice calls. One channel is created for each end party (i.e. caller and callee). Channel actions include, among other things, connecting caller and callee channels, playing announcements and music-on-hold or reading dual-tone multi-frequency signaling (DTMF) inputted by the call parties. The data of calls in RTP messages is also handled by a media gateway.

The brains of the system is called *call handler*. A call handler maintains information about all currently active or waiting actions on different communication channels in memory. When receiving notifications about state changes for state-based communication channels, a call handler analyzes the notifications and decides what to do in which case. For example, when receiving a notification related to a call telling that a DTMF selection is done by the caller, a call handler then analyzes the received information. Based on the call case, it then decides what should be done: should a call be started,

forwarded, rejected, hanged up et cetera.

The inner architecture of a call handler is module based. Handling of different tasks has been divided into separate modules, which are all part of the same process. Modules can be dynamically loaded and unloaded. The modules are able to communicate with each other using internal messages.

Application programming interface exists for providing a way for fetching information from the system using representational state transfer (REST). It provides a restricted possibility to affect call flows from applications such a web interface. The possibility is needed for example for dispatcher calls.

Messaging interface is a component easing communication between heterogeneous components of the system. It forwards notifications about changes in communication channels to call handlers, and responses from them back to the original information sender. Messages are delivered to call handlers using Transmission Control Protocol (TCP) and so the messaging interface knows if forwarding a message to a call handler succeeded. Thus the messaging interface is able to make sure that a message is never sent to a crashed call handler. Also, if a response to a message is not received from a call handler within a defined timeout, the messaging interface sends the message to another call handler.

In this paper, an abstract version of the messaging interface is used for better examining the behaviour of call handlers. In the abstraction, the messaging interface selects randomly a call handler to which a message is sent. The selection is done by randomly picking one call handler from all active call handlers. When getting a response to a message, the messaging interface forwards the response back to the correct component. The point of interest of this paper is in communication between call handlers and thus media gateways, command interfaces and messaging interfaces can be considered to be a "black box" from which messages are received, and to which responses need to be returned.

Worth noting is that even if messaging interface appears as a single point of failure in the Figure 2.2, the actual implementation of messaging interface is redundant. In this paper, the messaging interface is only presented as one component for removing the need for presenting how communication with duplicated messaging interfaces is implemented.

In addition to already presented components, there are other components which take care of other communication channels. Short Message Service (SMS) gateway allows sending text messages. Unstructured Supplementary Service Data (USSD) gateway enables both sending and receiving USSD messages. Call handlers provide an interface for SMS and USSD sending, and take care of creating event data records of sent and received SMS and USSD messages.

While the example system is able to handle different messaging channels, this paper mostly concentrates on calls. The reason for this selection of focus is the difference in the nature of the messaging ways. Calls are more difficult for the system to handle because of their real-time session-like nature: there are different states a call can have, and the state can change during a call. As opposed to calls, SMS and USSD messages are only one-time actions from the point of call handlers. While USSD is real-time connection, too, the connection is handled by the USSD gateway, and it is thus out of the scope of this paper. SMS messages also are not time dependent: when sending a SMS, one cannot be sure when the SMS will be received.

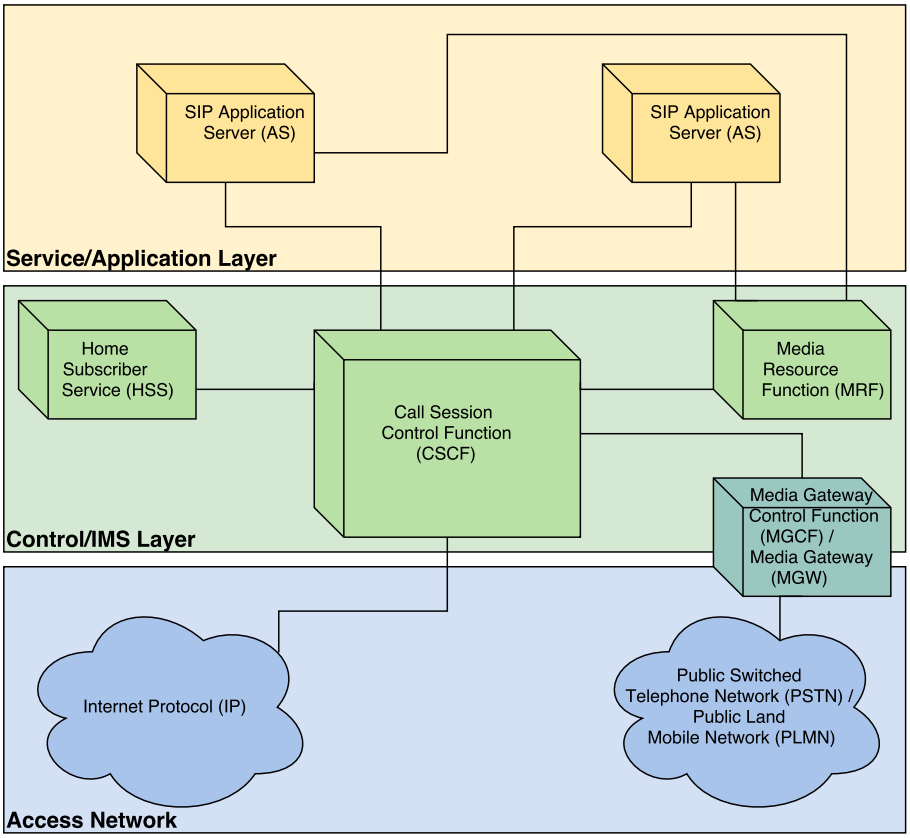

Figure 2.3: A simplified presentation of IMS layers by 3GPP specification.

The example system can be presented through the architecture of Internet Protocol Multimedia Subsystem (IMS) from 3rd Generation Partnership Project (3GPP) [1], of which a simplified presentation can be seen in Figure 2.3. Call handlers are working on the service/application layer, while

the components of black box, apart from the messaging interface, are mainly components of the control layer.

2.2.3 Initial state of fault-tolerance in call handlers

Currently, reliability in the system is achieved with two identical call handlers residing on different servers. One of them is the designated main call handler, which handles all calls and processes all messages as long as it is running. The other call handler is a warm backup, meaning that the process is running and ready for handling calls, but it does not contain copy of the data known by the main call handler. The call handlers are unable to communicate with each other.

Lack of communication between the main and the backup call handlers causes that no data is shared and the duplicates are not in a consistent states. As a result, if backup has to take over handling messages, it starts from empty state. Knowledge on all possibly ongoing calls at the time of crash is lost. In the best case, depending on its case and its state, a call itself is not necessarily interrupted, because RTP and channel handling is processed in a media gateway. However, even in the best case, the event data records about the calls are lost, and thus there might not be any traces of the calls once they end.

A missing record is problematic for three reasons. First of all, operators cannot charge for calls for which a record is not found. In some countries, records are also required by law and by other requirements of telecommunication authorities to be created and preserved. Thirdly, especially in critical systems it is crucial to know what happened during a call: was the call answered or not.

2.2.4 Signaling in calls between components

An example is presented to shed a little bit light on how exactly the system is behaving during a call and in which cases call status is changing and messages are exchanged between the black box and a call handler. Call party A calls an Interactive voice response (IVR) number. The call is answered and announcement is played asking A to select something with DTMF. A then presses '5'. The call is forwarded to a call party B, who answers the call. After a while, B hangs up.

For the sake of simplicity, A is calling from a VoIP phone and B is answering in a VoIP phone. That way there is not need to include any trunks or other elements in the image. Figure 2.4 shows the messages going from

phones to media gateway and vice versa and from media gateway to call handler and vice versa. Signaling between phones and media gateway uses standard session initialization protocol (SIP) as presented in the standard [55].

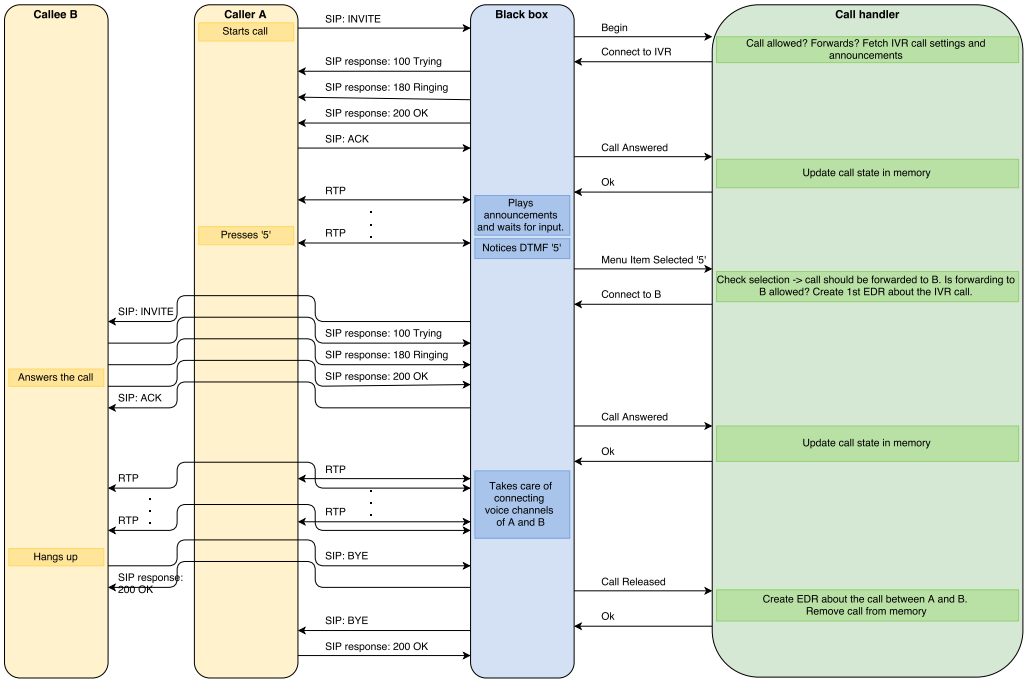

Figure 2.4: Message passing between components and phones and a simplified presentation of the inner functionality of the components in the example call case. SIP signals in the figure follow SIP standard [55].

2.3 Methodology

Defining fault specification is needed for improving fault-tolerance. The specification should answer at least questions that which failures are expected to occur and what level of fault-tolerance is expected. Additionally, the expected system properties need to be defined. The expected system properties play an important role as criteria when protocols for implementing fault-tolerance are decided and later evaluated.

2.3.1 Simple failure specification

The four failure classes as presented in [5] are crash, send-omission, general omission and Byzantine failures. For the example system, the expected failures to happen are crash and omission failures.

In the example system, crash failures mean that either a call handler has stopped unexpectedly, or a server has stopped working. Permanence of the failures depends on a case. In cases such as hardware failures, the failure might be permanent; they are not fixed on their own and interaction is required from a system administrator. On the other hand, in simple server restarts, for example as a result of temporary power cut, the failure is only temporary. Crash failures are caused by both physical and design faults.

Send-omission and general omission failures mean that one call handler is unable to send or receive messages from any other component. Root causes of omission failures might be either permanent, for example in the case of a broken network card, or temporary, for example if connection is temporarily lost.

Byzantine failures are excluded in this paper. Detecting and surviving them would require message data validation and process tracking, which are out of the scope of this paper. However, since most of the messages are heavily dependent on existing information about ongoing calls, some arbitrary failures in which message data is changed, are noticed when a message is being processed by a call handler. In this case, the message is just ignored. Also, handling arbitrary or byzantine failures reliably would require a slightly different approach. Researches have shown that tolerating k byzantine failures requires $(2k+1)$ nodes [22]. Because of leaving Byzantine failures out of the scope, from now on the term "failure" is used to mean only crash failures and omission failures.

This paper mainly concentrates on architectural level of the system. Thus it is assumed that media gateways and messaging interfaces, "the black box", always work correctly. Their fault tolerance is a subject for another study. It is also assumed that any fault on other levels (i.e. hardware, operating system and software/node levels) will cause symptoms similar to crash, omission or arbitrary failures on the architectural level.

Expected behaviour for the example system is that even in the presence of failures, end users should hardly notice any difference in the service. The example system as a whole should continuously satisfy the system specification. This is achieved with masking fault tolerance. If failures are faced, the system should still be able to hide the failures from end users.

2.3.2 System requirements

Using the example system in critical communication creates high expectancy for how much reliance can be put on the system. That is why two the most important properties for the system are reliability and availability, the main factors of dependability. Reliability for the example system means that after a call starts, it should be correctly handled until the end of the call. Thus call duration is the time frame for the inspection of reliability. Even in the case of a call handler crash, no call should be lost. Availability prospect expects that when a new call is to be started, there should not be a situation in which there are not any call handlers available for handling the call.

A part of reliability is the timeliness of the system. Because of time criticalness, not responding sufficiently fast violates the system specification. For the end users, slow responses from call handlers appear as delay. It may result in users getting tired of waiting. The maximum allowed time within which a response from call handlers should be received can be assumed to be some seconds, or a minute in maximum. The expected response time, however, is also a message-dependent attribute. For example, when a call is beginning or ending, the response time can be a bit longer than in the case of call answer, because they appear in a different way for a user.

Another important property for the architecture is limited scalability with continuous supported node amount. The architecture should allow any call handler amount [2, N], in which N is some tens, maximum. Supporting larger amounts of call handlers is not necessary, because the point of this paper is to increase fault-tolerance, not to increase processing capacity. Also, any need for large call handler amounts is not expected.

Other call handlers have to be able to continue where another call handler was at the point of its crash. Thus data consistency in node memories is a major property. A problem arises from the fact that the system is unsynchronized. Implementing strict data consistency could cause major decrease in the efficiency of the system. That is why only loose, eventually consistent system can be expected. Data in the memory of call handlers does not need to be in a consistent state at each moment, as long as it eventually is. Also, because of time criticalness, ”eventually” should not be a long time.

Another desired property is efficiency. The selected methods should not cause excessive overhead to the system. Additionally, for the sake of simplicity, the network is assumed to be equal between any node: delay caused by network in message transfer is assumed to be equal. The assumption zones out the need to consider networking distances between call handlers.

2.3.3 The goal

A short recapitulate about the starting point: There are call handlers in an asynchronized system. Call handlers are located on separate servers and thus they do not share the same computing resources. They are also connected to each other via an unweighted network. At the current state of the system, fault-tolerance is provided by a warm backup call handler.

The desired goal is to find a way to create replicated system which allows using any call handler amount up to some small N, and which allows parallel processing of different calls on different call handlers. Additionally, on top of the replication, a protocol for call takeover in the case of a call handler failure is needed. For a complete system with the desired properties, three sub-problems have to be solved:

1. **Architecture of the system and messaging protocols.**
   How data is disseminated between call handlers?

2. **Failure detection.**
   How failures of single call handlers are detected?

3. **Failure handling.**
   On top of the other two questions, the third question is that when data has been disseminated and a failure is detected, how the other call handlers should react to the detection? Who takes over the control of the calls of a failed call handler?

Different topologies and protocols for creating fault-tolerance in distributed systems have been studied a lot for decades from different point of views and for different kind of systems. Having already-defined expected properties in mind helps in examining through the studies for finding suitable solutions.

One important note to keep in mind when examining topologies and protocols is that for better reliability, single points of failure should be avoided at any time. Single points of failure would not only cause unnecessary risks which are undesirable in systems which heavily require reliability, but also could possibly be bottlenecks in communication.

Chapter 3

Fault tolerant data spreading

A problem in improving reliability of a system is that there are numerous ways for doing it. Methods for improving reliability of a single process consist mostly of algorithm design and preparing for different failures. However, it can only be done up to a certain point. After that, reliability can mostly be increased by replicating the processes, which allows access to the service even if one member process of it is completely unavailable.

In a replicated system, there are more than one member which are able to provide the same service. If one of the members becomes unavailable, others or one of the others is able to continue providing the service, and downtime does not occur or only occurs for a really short time. A major challenge in replicated systems is how to spread information between members reliably. Without data dissemination, different members would have different knowledge on the current situation. If the provided services depend on the current situation, the results might differ depending on which member is contacted.

3.1 Studies about fault tolerance

The earliest studies are mostly oriented toward studying hardware level fault tolerance and failure tolerance of network topologies in the case of reliable communication in multiple processor systems. Computers were not as evolved as they are now a couple of decades ago, from which time a lot of studies about network topologies are found. The selection of a network topology had a lot of importance in communication, when computers' capability to connect outside was restricted by the amount of Input/Output ports. A wide spectrum of different properties are used in different studies for evaluating the solution: connection amount, path lengths, messaging complexity, time restrictions, performance, fault tolerance, maintainability and extensibility,

to name some.

One of the most researched network topologies is based on De Bruijn graph, discovered in 1946 by Nicolaas De Bruijn, originally called the graph T-nets [16]. The graph is based on P_n-cycle, $n \geq 2$, which is an ordered cycle of 2^n bits. The bits are ordered such that any permutation of n bits can be found from the P_n-cycle exactly once. For example, when $n = 2$, possible bit sequences are 00, 01, 10, 11, and the only sequence containing all of them exactly once is 0011. Since the bits are in an ordered cycle, 0011 is exactly the same sequence than 0110, 1100 or 1001. De Bruijn graph is then formed by first setting all possible permutation of n length bit series as the nodes of the graph. The nodes are then connected with *unidirectional* edges following the rule: for two nodes A and B, if the last $n-1$ bits of A are the same as the first $n-1$ bits of B, then edge is added from A to B. [16]

The graph can be extended by allowing the use of other digits in addition to zeros and ones in a cycle. The amount of accepted digits is presented with r. In the extended case, the total amount of nodes in a De Bruijn graph is r^n, in which both r and n are positive integers. De Bruijn graph can then be presented as D(r, n). As a result of how the graph is formed, there are self-loops: nodes from which there is an edge back to the node itself. There are always r self-loops in a graph with r^n nodes. [58]

In many topology studies based on de Bruijn graph, the self loops are removed or otherwise ignored. Varying fault tolerances for different versions of De Bruijn graphs are achieved. One such a topology based on De Bruijn graph is examined in [21] for node connectivity: shift and replace graph (SRG). Directions of edges are removed and thus the edges become bidirectional. Also, self-loops and multiple edges between same nodes are eliminated. The approach achieves fault tolerance of $(2r-3)$ for node amount r^n. Other proposed solutions based on De Bruijn graph include load balancing technologies for multiprocessor networks [50, 57], distributed hash tables [15, 43] and networks-on-chips [32, 33].

In addition to De Bruijn based topologies, other regular graphs have been studied for fault-tolerant communication, as well. The topologies include different kind of cubes, hypercubes, Cayley digraphs, and other regular digraphs. One such a regular digraph network topology is presented in [51]. It provides near-optimal fault tolerance. When r is the amount of accepted digits and n is the amount of digits in a permutation, the number of nodes in the digraph is r^n. A path between any two nodes can always be found, and the maximum length of the path is $(2n-1)$. The number of connections per a node is fixed, however, depending on the value of r, the amount might be either r or $r+1$. Fault tolerance achieved by the topology is $(r-1)$ or r. [51]

Different variations of hypercubes have been studied. Typically, hypercubes require node amount of $N = W^D$, in which W and D are positive integers, D is the amount of dimensions and W is the amount of nodes in each dimension (width of a dimension) [7]. One such a hypercube is Boolean n-cube, in which there are $N = 2^n$ processors placed one in each of the corners. Two hypercube presentations are presented in [7]: *generalized hypercube* and *generalized hyperbus*. An advantage in them is that the proposed structures allow almost any number N. The structures are also highly fault tolerant and have small average message distances. Generalized hyperbus structure has N communication buses, and each node is connected to two adjoining buses. The worst case distance between nodes depends on the amount of buses. Having only two connections per node makes the generalized hyperbus cheap when it comes to network costs, but fail tolerance is fairly bad: a node is out of function if both of its buses are out. Generalized hypercube provides better fault tolerance. In the generalized hypercube, the total number of processors is presented as a product of m_i's: $N = m_r \times m_{r-1} \times ... * m_1$, when $m_i > 1$ for $1 \leq i \leq r$. The total number of connections per a node is $L = \sum_{i=1}^{r}(m_i - 1)$, and the total amount of connections in the structure is $\frac{N}{2} \times L$. Fault tolerance of the structure is $(L-1)$ and the worst-case distance between any two nodes is $(r + 1)$. [7, 11] Another proposition for improving fault tolerance in hypercubes is made in [11]. A k-safe hypercube has been developed to tolerate maximum of $(2^k(W - k) - 1)$ faulty nodes, in which W is the dimension of the cube and k presents k-safeness: each not-faulty node must have at least k not-faulty neighbours. [7, 11]

Combinations of De Bruijn and hypercubes, named Hyper-De Bruijn, have been proposed, too, for finding the ideal solution for fault tolerance and shortest path routing. De Bruijn graph is extended to high degree and combined with hypercube by first connecting nodes using De Bruijn graph constructing rules, then using hypercube constructing rules. One such a graph is Hyper-De Bruijn-Aster presented in [46]. If De Bruijn graph is presented as $D(d_{hd}, m_{hd} + n_{hd})$, as opposed to the previously presented $D(r, n)$, and a hypercube is presented as $H(m_{hd} + n_{hd})$, then Hyper-De Bruijn-Aster is presented as $HD*(m_{hd}, d_{hd}, n_{hd})$. Total amount of nodes is then $2^{m_{hd}} d_{hd}^{n_{hd}}$, and the achieved fault tolerance is $m_{hd} + 2d_{hd} + 2$. [46]

A special case of topologies is a fully connected topology in which all nodes can communicate directly with each other. Fault tolerance of fully connected networks is easy to estimate: since there are n nodes, and $(n-1)$ connections between nodes, maximum of $(n - 2)$ failures are tolerated. However, the performance in fully connected networks might be bad for one node.

Also hierarchical topologies have been studied, most commonly tree topologies. Fault tolerance of binary-tree-networks is examined in [52]. In [40], two

typical tree solutions are examined: *star* and *non-homogeneous tree*. Star is a two-level tree with homogeneous nodes: one root node and all other nodes are leaves. In non-homogeneous tree, nodes on different levels of the tree are not of the same type, for example the nodes in different levels may present main CPU, I/O controllers and I/O devices. Also, designing optimal edge fault-tolerant supergraphs of these two is investigated. Presented solutions provide near-minimum overhead, but they might not work in all cases. Also, for achieving k-edge fault tolerance in non-homogeneous tree, there must be $k+1$ edges from each node. For a simple k-edge fault-tolerant supergraph, if there is only one root node, there must be $2k+1$ global nodes for providing fault tolerance of k. Global nodes are connected to all the other nodes in the graph. [40]

3.2 Middleware solutions

Middleware can be used as solution for a wide range or problems: message passing, message spreading, load balancing or fault tolerance. An advantage in middleware is that it might ease implementing interconnection between nodes, especially between heterogeneous nodes.

There are different paradigms for middleware: message queues [66], invocation-based request/reply and event-based publish/subscribe [49]. The functionality of message queues is to forward messages to one or more clients with some specific rule, without caring if the receiver is interested about the messages [66]. In request/reply middleware, clients send requests to middleware and receive responses, thus the communication is only one-to-one [49]. Publish/subscribe middleware provides solutions for both one-to-one connections and unnecessary messages sending: clients subscribe to the middleware and announce what kind of messages they are listening. Whenever receiving a message, middleware publishes it to all interested subscribers. Thus publish/subscribe offers many-to-many communication [49, 66].

One of the much used standards in general purpose middleware is Common Object Request Broker Architecture, CORBA. "Traditional" CORBA itself does not guarantee message passing within time limits or fault tolerance, but fault-tolerant version of CORBA has been designed: Fault Tolerant CORBA. The object of Fault Tolerant CORBA is to provide high level of reliability with the means of redundancy, fault detection and recovery, and the standard requires that there are no single point of failures and it offers several fault tolerance strategies. The standard also defines requirements, limitations and basic fault tolerant mechanisms that must be implemented in a Fault Tolerant CORBA system. [47]

A lot of generally used, fault tolerance providing middleware exist. One is Apache ActiveMQ, by The Apache Software Foundation [61]. It is a cross language Java Message Service provider which offers a lot of features. Apache ActiveMQ has a failover transport layer which can be used for providing several addresses, which are then used if the main address is out of reach. ActiveMQ provides the core functionality for JBoss A-MQ, previously Fuse Message Broker, by Red Hat Inc. [54]. JBoss A-MQ is able to notice if some nodes are out of reach. Thus it works as an outside monitor for fault tolerance on node failures. JBoss A-MQ also provides fault tolerance methods to hande the cases of a message broker failures [38]: a fault tolerance protocol for selecting a new broker, and master/slave groups.

CORBA complying middleware, Apache ActiveMQ, JBoss A-MQ along with many other general purpose middleware are usually mainly targeted for message passing or load balance. However, middleware primarily designed for providing fault tolerance have been studied as well. Distributed systems have their own expectations about properties and requirements, so also studies about fault tolerance providing middleware have different premises about the target usage.

Adaptive middleware for fault tolerance in satellite distributed computing and in other space applications is presented in [22]. Each node, on-board computers in this case, consist of one on-board unit and one adaptive middleware. They form a master/slave type of a group, with one master and one or more slaves, master being the on-board computer with the smallest identification number. On-board units communicate with each other through a multicast communication bus, as well as middlewares communicate with each other through another multicast communication bus. Also, each on-board unit communicates with the middleware which locates on the same computer. Middleware detects any unit faults and sends fault messages to other middleware when noticing a failure. They take care of isolating the faulty unit and reconfiguring the other group members. Then middlewares of not-faulty units inform their associated units about the changes in group. Now the master on-board computer redistributes tasks between not-faulty members. Estimations about the achieved fault tolerance have not been presented. [22]

An event-based middleware architecture for large-scale distributed systems is presented in [49]. There are two types of components in the basic case: event clients and event brokers. Clients are nodes that can publish messages and that can subscribe to listen some kind of messages. Brokers are the actual presented middleware part of the system. Brokers are connected with each other in an arbitrary topology, and clients can subscribe to any of them as a publisher or as a subscriber. Any publication is transformed into

a message which is passed through the network using some event dissemination tree for finding all relevant interested subscribers. An architecture is proposed which adds rendezvous nodes to the network. Rendezvous nodes are special brokers which are known to all publishers and to all subscribers, and they handle advertisements and subscriptions. At least one rendezvous node exists for each event type, and they are replicated throughout the network. [49]

In [68], a robust middleware for adding fault tolerance to parallel and distributed systems while achieving high efficiency is presented. The solution is based on a dynamic robust embedding/allocation middleware scheme. The idea of the solution is that at least most of the healthy processes could be used for computation, and the used processes are searched for again when new failures occur. Some opportunistic strategies for selecting healthy processors, that are used in computing in case of faulty processors, are presented. The middleware often finds some submesh of the original system, and in the proposed solutions, the sub-mesh usually is a grid mesh. [68]

Another middleware for fault tolerance has been proposed for complex distributed simulation applications in [63]. The goal of the system is to maintain availability and reliability. The system is based on Fault Tolerant CORBA, and so its structure is largely similar to it. The system can dynamically allocate needed resources for achieving desired reliability. [63]

Having high availability and low overhead as the goal, [34] presents a middleware with deterministic algorithm for clouds. In the case of node failures, the middleware automatically allocates fault-free backup nodes for the system. The proposed solution uses active replication, thus all replica nodes have the same role. Middleware monitors the system, and when it notices that utilization of resources exceed some predefined threshold, it adds a new node from existing nodes to serve the resource. [34]

Another middleware with low latency designed to provide fault tolerance in cloud systems is presented in [69]. It protects the system against crash and timing failures by replicating the application processes using leader/follower replication. Each process is a part of a process group, and there is a virtual connection between groups. Virtual connection is presented as a full-duplex, many-to-many communication, in which all members of a group listen to the same virtual port. Special Low Latency Messaging Protocol is used for the actual message passing between groups, in which ACK message is expected within time limit, else timeout occurs and retransmission is tried. Middleware ranks the members with 1, 2, 3, ... and takes care that each member in a group knows their group members and knows which is the leader. The rank also determines the timeout for detecting failures in preceding leader/followers: the larger the rank, the longer the timeout. In implementa-

tion and tests, the middleware produced overhead from 15 % to 55 %. Fault tolerance of the system is not further studied. [69]

Also, middleware solutions for heterogeneous networks are designed, for example in [12, 64], and for cases in which nodes cannot interact with each other easily [22]. Because of their preparation for heterogeneous systems, middleware solutions often offer only loose coupling for different elements in a network. Also, generally message queues do not necessarily provide persistence of messages [66].

3.3 Protocol level fault tolerance

During recent years, cloud and peer-to-peer services have become more typical types of services. A problem especially in peer-to-peer networks is that members might pop in and out at a fast rate. That is why finding interconnection architecture which would be dynamic, scalable and reliable has become even more important.

A number of different applications for protocols which allow dynamic connections have been studied, for example: replicated database management [18], scalable services architecture [44], protocol for group multicasting [8, 30, 31], and failure detection or threshold crossing detection [20, 42, 67]. One category of programs for which dynamic, fault tolerant and time critical protocols are developed is multi-player games. They often require that all members of a network must know the current situation. As an example, the design architecture of one peer-to-peer solution for synchronizing game clocks in a network of unsynchronized members is explained in [6]. Other such studies are done for example in [56], [24] and [14].

A widely studied method containing a huge amount of protocols is *gossip protocol*, also called *epidemic protocol*. The names come from how the protocol spreads messages in a network. Messages can be thought to be rumours which then spread from one node to another as if the nodes were gossiping. Another way to think about it is to consider how epidemics are spreading in population: it infects new people (at least seemingly) randomly. [28, 31]

An abstraction named "Flat Gossip" protocol is presented in [31] for examining general properties of gossip protocols. In flat gossip, there are N members in a network, and one rumour is delivered for $log(N)$ rounds. Each round, group members send rumours to b other, randomly selected, members. Thus each member of a network has a probability of $P = 1 - \frac{1}{N^b} * (1 + o(1))$ of receiving the rumour, when N is amount of nodes and b is the amount of nodes to which the message is sent. [31]

Gossip protocols, in general, provide good fault tolerance, because of

their probabilistic nature. Fault tolerance is achieved because randomization of message targets increases the probability that a message finds another route around a possibly faulty connection [31]. However, gossip protocols with biased target selection exist, too. Usually the advantage of the biased selection is that it guarantees that a message is received by all members within bounded time, while the basic gossip only promises that with a high probability all members receive all rumours. One biased way for determining the target node is round robin. In round robin protocol, the target is selected using equation $TargetID = SourceID + r$, $1 \leq r < N$, in which r is the current round and N is the node amount [53]. A variation of round robin is binary round robin, which eliminates redundant gossiping. Then the target node is selected with equation $TargetID = SourceID + 2^{r-1}$, $1 \leq r < \log_2(N)$ [53].

Efficiency of gossip protocols is another subject worth examination. In flat gossip, messages are sent to random nodes for r rounds, so it is very likely that some nodes receive the same messages more than once. Transferring these duplicate messages requires resources without giving additional advantage to the forwarding nodes. That is why one problem in gossip protocols is the increased network overhead and messaging complexity. Also, research is most often done on gossip protocols in synchronous systems, and asynchrony of system might cause the protocols to be even less efficient or reliable [28]. Gossip protocols without fixed round amount exist, too.

Two gossip protocols for multicasting are described in [30] and [31]: *hierarchical gossiping algorithm* and *adaptive dissemination*. Both are based on leaf box hierarchy in which each node is a member of a leaf box. Both also decrease network overhead, however while adaptive dissemination guarantees high reliability, hierarchical gossiping decreases reliability slightly compared to flat gossip. In the hierarchical gossiping algorithm, the messages are passed for $log_2 N$ rounds in leaf box hierarchy, in which N is an estimated amount of members in a group. The probability that all members receive a rumour is $\pi(log_K(N) - 1)$, in which K is a small constant and $\pi(j)$ presents the probability that there is a direct path from any member s to all other members in its subtree of height j, using only archs of the subtree. Adaptive dissemination is a combination of leaf box hierarchy and some basic gossiping protocol, for example flat gossip. In the adaptive dissemination, multicasting node first sends a message to itself and a hop count is included to the message. The count is increased every time the message is forwarded. Targets are selected by taking one random node from each leaf box. Messages are forwarded for $log_K N - 1$ rounds, and receiver always sends an acknowledgement. When the final round has been done, or if acknowledgement is not received, a gossip message is sent and it is forwarded using the selected gossip protocol, such as

flat gossip. The achieved fault tolerance is at least as good as the reliability in the used basic gossiping protocol. [30, 31]

Typical systems which need reliable data dissemination are replicated databases, for which data dissemination methods have been studied for decades. Use of gossip-based solutions for maintaining consistency in replicated database is examined in [18]. The study presents two gossip-based protocols for replication: *anti-entropy* and *rumour mongering*. Anti-entropy can be viewed as a version of flat gossip with $b = 1$. Each site selects regularly another site with which it shares its whole contents. If differences occur, they are solved. Rumour mongering allows update-based replications: new updates are the subjects of rumours. The study also presents a randomized anti-entropy algorithm which provides performance improvements. [18]

Study [8] presents bimodal multicast: a gossip protocol named *pbcast*. It contains two steps: unreliable, hierarchical broadcast and anti-entropy protocol. First, processes send a multicast to other processes using an unreliable multicast method. During the second step, some anti-entropy protocol is used for the duration of some unsynchronized rounds for correcting losses of messages in the first step. [8]

Karp et al. in [39] present a push-pull protocol for rumour spreading. Each round, every node selects randomly another node to which it connects. These two nodes can exchange information to both direction. Every round, nodes have to decide whether they push (tell about new rumours) or pull (ask for new rumours). Lower limits for needed forward rounds are thus achieved, compared to traditionally push-type gossip algoritms, such as flat gossip. However, the nodes should be producing messages sufficiently often, otherwise pull-operations are only waste of messaging capacity. [39]

Another protocol for rumour spreading is presented in [19]. Each member in a network has some identification (ID) in a cyclic ID space. When a member receives a new rumour that it did not know yet, it selects the next target node randomly. Two-way communication is allowed. The receiver member uses it for acknowledging its readiness to receive rumour and for informing if it already knew the rumour. After that, the sender nodes selects the next target for the rumour using the following rule: If the previous receiver knew the rumour already, the rumour is next sent to the successor of the previous receiver. Otherwise, the next target is selected randomly. The rumour is sent until there have been $R + 1$ members which already knew the rumour. R is a parameter which can be a function of the total node amount. The presented protocol is not very robust against adversarial node failures: if many consecutive nodes fail (at least $R + 1$), the protocol might fail to spread the rumour for all nodes. A solution for the problem is presented: when successful and unsuccessful connections to other nodes are

differentiated, a node first finds a random node to which it successfully sends the rumour. Only after that it continues sending the rumour as defined. Nodes also keep track on how many unsuccessful calls were made. [19]

Three gossip protocols specifically for asynchronous systems are presented in [28]: Epidemic Asynchronous Rumor Spreading (EARS), Spamming Epidemic Asynchronous Rumor Spreading (SEARS) and Two-hop Epidemic Asynchronous Rumor Spreading (TEARS). In EARS, processes maintain Informed-list which contains that which processes have been informed about which rumours. After having gathered rumours, a process selects randomly another process to which all rumours are sent. Then the target process is added to the Informed-list, and the known informed processes are recorded in the message for telling that they have already received the rumour. When the original sender process detects that all other processes have received all rumours, it enters into a shut-down state. In the shut-down state, it receives and sends messages normally for $\Theta(\frac{n}{n-f} \log n)$ rounds, when n is the amount of nodes and f is amount of possible failures. During the shut-down phase, it sends information that all processes have already received the rumours. If new rumours are not received during the shut-down phase, the process stops sending messages and becomes quiescent. If noticing a new rumour at any time, the process exits shut-down or quiescent state. Achieved time complexity is $O(\frac{n}{n-f}\log^2 n(d+\delta))$ and message complexity is $O(n \log^3 n(d+\delta))$. δ is the relative process speed and d is communication delay. [28]

SEARS is a constant-time gossip protocol and a variant of EARS. In SEARS, each process selects a large set of other processes to which they send all the rumours. Also, shut-down state is shortened. A counter is added to each rumour, such that rumour initiating process sets it to zero. On every step, a process increases the counter when forwarding the message. After the counter of a rumour exceeds some threshold, the rumour can be ignored, and processes in quiescent state do not wake for sending the rumour anymore. Achieved message complexity is $O(\frac{n^{2+\epsilon}}{\epsilon(n-f)}\log n(d+\delta))$ for any constant $\epsilon < 1$ when $f < n/2$. [28]

The basic idea behind another constant-time protocol TEARS is that it is enough if only majority of processes receive a rumour; this gossip type is also called majority-gossip. There are two stages. The first stage is that each process sends its rumour to about $\Theta(\sqrt[2]{n} \log n)$ other random processes. During the second stage, processes forward the messages, received during the first stage, again to $\Theta(\sqrt[2]{n} \log n)$ other, randomly selected processes as second-level messages. Determining when to send these second-level messages is based on two rules. The rules are based on how many of the messages from the first stage are received. Achieved time complex is $O(d+\delta)$ time and

message complexity is $O(n^{7/4}\log^2 n)$ with high probability, when $f < n/2$. [28]

Use of gossip protocols for data dissemination between players in online games has been studied recently, too. One of the studies is done by Ferretti et al. [24], in which use of three different push-type gossip protocols are studied. In the first protocol, the probability of dissemination is fixed, and it is thus called fixed probability gossip. In the second protocol, the amount of neighbours to which to send the rumour is fixed. The third protocol is a probabilistic broadcast. They conclude that the use of gossip protocols allows fast dissemination and high responsiveness, however, when using low gossip probabilities, all members might not receive all rumours. [24]

Another study about usage of adaptive gossip in unstructured network in multiplayer online games is done in [14]. Three different gossip protocols are examined. The first protocol is simple: when receiving a new rumour, a member sends it to another member n with probability of v_n, for each member n except itself and the one from which the rumour was received. A method for calculating v_n is given. The second presented protocol is an adaptive dissemination threshold protocol. Each member maintains a dissemination threshold value γ_s for other members. The third protocol is based on the second protocol. A difference is that instead of only one threshold value for each node, a set of arrays, one array for each neighbour, is maintained. It is noted with simulations that adaptive strategies achieve better results in coverage and in delay of dissemination compared to non-adaptive ones, such as fixed probability or probabilistic broadcast. [14]

Chapter 4

Failure detection

Two main properties of failure detectors are specified in [10] to be *completeness* and *accuracy*. Completeness expects that any crashed process is eventually suspected of being faulty. Accuracy defines that not-faulty processes should not suspected of being faulty. However, it is noted that fulfilling both of these properties at the same time is impossible for failure detectors in an asynchronous systems. The reason is that distinguishing a slow process from a failed process is impossible. The problem of differentiating the two cases causes that perfect accuracy cannot be achieved. [10]

In addition to using middleware as failure detectors, also algorithmic methods can be used. One method is using heartbeats. The basic idea of heartbeat protocols is that processes send periodically heartbeat messages which tell others that the sender is still alive. The simplest version of heartbeat is a protocol, in which a heartbeat message is sent periodically to neighbours. Each process then keeps track that when one was the last heartbeat received. If long enough time has passed since the last heartbeat, the corresponding process is marked as suspected. This is a timeout based heartbeat.

A heartbeat protocol without timeouts is presented in [2]. Again, a heartbeat message is sent periodically for telling that the sender is still alive. Processes maintain a counter for each other processes, and when receiving a heartbeat message, the counter corresponding to the sender is increased. If a process crashes, its counter will eventually stop increasing. [2]

Because of the general properties of gossip protocols, one much studied application for them is failure detection. Gossip protocols are resilient and do not dependent on the state of a single processes. That is why they are also able of routing around faulty connections between any two processes, and most often provide high probability for complete distribution. Gossip protocols are also highly scalable, and they do not make assumptions about

networks. [59] These properties are favourable for spreading heartbeat messages.

One such a gossip-based, heartbeat utilizing, failure detection protocol is presented in [9]. The protocol is called piggyback protocol because it allows sending rumour information in application-generated messages. Another heartbeat-based failure detector gossip protocol is presented in [20]. It is designed to detect crash failures in wireless ad-hoc and mesh networks. In the proposed protocol, detection time is adapted from the times of previous heartbeat messages, and estimations about the arrival of next heartbeat messages are done. Adaptability is also used in [60], in which adaptive timeouts are used. Adaptive timeouts are an extended version of heartbeats, in which each process has a list of adaptive timeouts for their neighbours. If a process A falsely suspects the process B of being faulty, but then receives a heartbeat from B within timeout period, A increases the expected timeout for B. Thus the protocol is able to adapt to different network properties between different processes. [60]

Two gossiping heartbeat protocols, basic and hierarchical version, are proposed in [62]. Both versions achieve fairly good completeness and accuracy. The hierarchical version is especially designed for large systems. It contains two levels: groups, within which basic version protocol is used; and cross-group level, for which a modified version of the protocol is used. During every round, on average, one process of each group gossips with a process belonging to another group. [62]

In the basic version, each process has one heartbeat counter which they increase periodically. Also, they maintain a list containing an entry for all known processes. An entry contains two heartbeat values: the newest received heartbeat value of the other process and the heartbeat value of the list maintainer, which tells when the newest heartbeat value was received. After increasing their own counter, processes select randomly another process to which they send their own heartbeat value and the full list of entries. When receiving a heartbeat message, the list within the message is merged with the receiver process' own list. Occasionally, each process broadcasts their list for being discovered. A timeout T_{fail} is set such that if a process A notices that new heartbeat value for another process B has not been received for T_{fail}, then B is considered to be faulty. From that moment, if a new heartbeat value is not received for B for further $T_{cleanup}$ time, B is forgotten and its entry is removed. The basic version protocol is only capable of noticing if a process becomes fully out-of-reach for all others, not if there is a communication breach between any two processes. [62]

There are also alternatives to heartbeats. Other methods for failure detection are shortly presented in [60]. *Pinging* is a method in which processes

ask their neighbours if they are alive. If a response is not received to the pinging, the neighbour is suspected of being faulty. *Leases* is a mechanism which is used especially when processes are sleeping most of the time. Processes send messages to neighbours telling that it is alive, and then asks for lease for some duration. After that, the process goes back to sleep. As long as it sends a new message within the lease time, it is not suspected of being faulty. [60]

Chapter 5

Theory behind implementation

Hardware restrictions in computers are nowadays not as strict as they were a few decades ago. The increased capability allows that some design factors are not as critical in design as they were previously. One such a factor is the amount of networking connections one processor can make, which was part of the reason for importance of network topology studies earlier.

The implementation for all protocols is done on top of a fully connected network topology. One great advantage of it is that reconfiguration is not needed in case of crashes. Also, routes around failed connection links can be found if the network is not split. Call handlers are equal in a network: there are not different roles. Additionally, they do not depend on each other: an active call handler is fully able to continue working even if another call handler crashes. Thus a need for a certain network topology is not created by the system properties, either.

Middleware could provide solutions both for failure detection and for data spreading. It could maintain a list of active call handlers either by noticing when responses are not received from call handlers or by checking regularly if there is an active connection to call handlers. As an external detector, middleware is also independent of the actual processes, and thus failures in call handler processes would not cause other unnoticed failures or incorrect detections. In addition to taking care of failure detection, middleware can also handle spreading messages to relevant call handlers. However, middleware has its problems. The system should not have single points of failures, and thus middleware should be redundant as well. Also, takeover processing is difficult for middleware to take care of: how to tell a call handler that it should start processing calls of another call handler? Current state of the calls should be known by the new call handler. Thus all messages should be sent to all call handlers, or another method for data sharing should be found. However, it creates a possible problem of data integrity. If a message

from messaging interface is sent to more than one call handler, how to take care that they calculate the same result? Another drawback of middleware is that the amount of needed configuration would increase.

As a result, selection of network topology is slightly outdated way for improving fault tolerance in the case of the example system because of available computing resources. It does not provide solutions for the problems. Middleware, on the other hand, would bring additional unanswered problems regarding the need for takeover and data integrity. That is why protocol level solutions are used for the example system.

Failure detection and data dissemination differ in one major aspect. In a failure detection protocol, all call handlers do not need to receive all heartbeat messages as long as they receive any new message within a set time limit. On the contrary, the selected data dissemination protocol should be able to disseminate data fast and reliably to all call handlers: each call handler must receive all messages. That is why two separate protocols are used for the solutions.

On top of the two protocols, a takeover protocol is created for responding the question about what to do in the case of a failure. The takeover protocol defines how call handlers select who should be taking over the calls of a failed call handler, how takeover is done, and how other call handlers should be informed about the takeover. The takeover protocol uses the failure detection protocol for detecting when takeovers should occur, and uses the data dissemination protocol for spreading information about takeovers.

In the example system, calls are session-like data: it is very likely that more than only one message per a call is received. That is why ongoing calls must be kept in memory until they end. For evading possible problems between call handlers and other components, it is desirable that each call is handled always only by one call handler at a time. Different call handlers can handle different calls at the same time. Thus a kind of semi-passive replication is needed. The solution should not be address-oblivious; identifying call handlers should be possible. For this, an addressing system is needed. Also, for making takeover protocol easier, as will be seen in the relevant Section 5.3, the identification space should be cyclic.

The rest of this chapter presents all three protocols. Theory of the selected failure detection protocol is explained in Section 5.1. Then data dissemination at theory level is presented in Section 5.2. Finally, in Section 5.3 takeover is presented.

Data name	Type	Description
Node ID	Integer	Unique ID of the node
Heartbeat	Integer	The most recent received heartbeat count for the node
Last seen	Integer	The heartbeat count of the list owner call handler telling the point when the most recent heartbeat count for the another call handler was received
Suspected	Boolean	True, if the node is assumed by the list owner node to be failed

Table 5.1: Data contained in an entry about a node.

5.1 Detecting failures

A heartbeat approach for the problem is selected over the other two methods presented in Chapter 4 because of its properties. In pinging protocols, which is push-pull type of protocol, two messages are sent for each live check and thus pinging burdens network more than heartbeat, which is only push type of protocol. Additionally, heartbeat protocol is selected over leasing for simplicity reasons. Leasing would require additional protocols and safety checks for taking care that failures are noticed fast enough and for overcoming the problem of unsynchronized system in which clock times of servers cannot be trusted to be the same.

The selected protocol is based on the basic heartbeat case of [62] with message spreading happening as in the flat gossip presented in [31]. In the protocol, the lack of global clock and time is solved by using a *heartbeat counter*. The heartbeat counter is updated periodically and the new value is reported. Spreading heartbeat values using a gossip protocol provides probabilistic certainty that heartbeat values spread to all call handlers. Since the connections between all call handlers are assumed to be equal in this paper, there is not need to take adaptability into account. Also, call handlers are equal, so hierarchical solutions are not needed.

The implemented protocol for call handlers contains the same information than what is presented in [62]. In addition to maintaining the heartbeat counter, each call handler maintains a table which contains an entry for all other call handlers in the network. In this paper, the table is called *status table*. All fields of an entry and a short description about them is presented in Table 5.1.

In the example system, steps of failure detection protocol are executed in local *rounds*. One round of a call handler is done every T_{round} seconds. The first step of a round is increasing the heartbeat counter by one. On the next step, the call handler collects and checks all rumours it has received since last round. A list is constructed containing call handler and heartbeat counter pairs, one for each call handler, selecting the newest heartbeat value for it. The next step is updating step. The list is iterated through, and if newer heartbeat value, than currently known in the status table, is noticed, the status table is updated accordingly for the relevant call handler.

After updating, a failure detection check is done. A call handler checks if there are call handlers about which rumours containing a new heartbeat count have not been received for at least t_{fail} local rounds. If such call handlers are found, they are marked as "suspected" in the status table and takeover protocol is performed. The takeover protocol is explained later in Section 5.3.

The final step of a round is a forward step, which starts by creating a new heartbeat rumour message. A call handler creates a message and adds to it the newest known heartbeat counts of those call handlers about which it heard any rumour during this round. Also the new heartbeat count of the call handler itself is added to the rumour message. Always, the newest known heartbeat value is added, even if the received rumours contained older information than what was already known in the status table. The call handler then selects maximum of b_{hb} other not suspected call handlers. The newly created message is then sent to the b_{hb} selected call handlers.

There are some differences between the implemented failure detection protocol and the basic protocol presented in [62]. A major difference is that instead of forwarding and comparing full status tables, only the rumours which were gathered during the step two of a round are forwarded in the example system, however, always with the newest known heartbeat counts. The decision brings minor savings in rumour message sizes, while the accuracy seems to stay adequate, as the tests described in Section 7.1 show. Another difference is that call handlers never broadcast their full list. In the example system, the locations of call handlers are always known, so there is not need for call handlers to advertise themselves with broadcasts. Also, when spreading heartbeat rumours, b_{hb} call handlers are selected to which the rumour message is sent, instead of selecting only one. This decision helps new heartbeat counts to spread faster.

The main focus of the protocol is to detect if a call handler becomes unavailable by all call handlers. Thus the most important kind of failures to detect are crash failures. Expected detection of omission failures depends on the case. Because gossiping is able to route around connection breaks,

the failure detection protocol cannot detect communication breaches between two call handlers. Only if a call handler becomes unavailable altogether, the protocol should be able to notice the problem. Thus single omission failures are not causing problems to the system, and neither are long lasting consecutive omission failures between any two call handlers, if there is at least some route from all call handlers to others. However, if continuous, consecutive omission failures are occurring for a call handler in all connections, the result is that the call handler appears as crashed, and it should be detected. Byzantine failures are out of the scope of this paper.

The simplified process flow of the heartbeat protocol is presented later in Figure 6.2.

5.1.1 Maintaining message order

A typical problem in asynchronous systems is message ordering. When there are several call handlers sending rumours independently of each other, how to take care that receiving old information late does not affect the system and break completeness and accuracy?

The problem of message orders is evaded with how the heartbeat counters work. Each call handler updates a heartbeat counter independently from others, and when forwarding a value, it is never manipulated. Thus the heartbeat values of a call handler are directly comparable.

Apart from byzantine failures, the only time when received heartbeat values of a call handler are not directly comparable is when a call handler has been restarted and it restarts counting from zero. This is a special case which is more closely examined in Section 5.1.4.

5.1.2 When to stop

An important question in gossip protocols is when to stop forwarding a rumour. Typical solutions include counters in messages or time calculations. In the implemented failure detection protocol, such solutions are not needed.

During every local round, a call handler creates and sends a new rumour which includes the call handler's newest heartbeat value. This new value invalidates the old value which is most likely still spreading in the system. It was noted earlier that all call handlers do not need to get all heartbeat rumours, as long as they receive at least any newer value within t_{fail}. Therefore it is justifiable for a call handler to discard any old information and only forward the newest known heartbeat value, when receiving several different heartbeat values for a same call handler within one round. Stopping rule for the implemented failure detection can then be presented as: when receiving

a rumour containing a heartbeat value older than another heartbeat value received during the same round or value already known in the status table, the old rumour is not forwarded anymore.

5.1.3 Selecting timeout for detection

A significant part of the used failure detection protocol is failure detection timeout, presented with t_{fail}. If a call handler does not receive a new heartbeat value about another call handler within t_{fail} rounds, the call handler assumes that the other part is unavailable and flags it as suspected. The base unit of t_{fail} is one local round r, which again is presented in seconds.

Selecting a proper t_{fail} is a major problem in the failure detection protocol. The problem arises from two contradicting expectations: detecting failed call handlers should happen as fast as possible, however, as little false detections as possible should occur. The smaller the timeout value is, the faster failures are detected but the more false detections occur, and vise versa.

One approach for selecting a good failure detection timeout limit is examining probabilities of one call handler receiving any new heartbeat before the detection timeout occurs. However, examining the exact probabilities is difficult, because the system is asynchronous. As such, it does not have a global clock and nothing can be assumed about delays or temporary slowness of call handlers. Therefore, for calculating approximate, directional values for t_{fail}, probabilities in a similar, but semi-synchronous system are examined. In the semi-synchronous system, local rounds of all call handlers are expected to happen always at the same time, and thus the concept of a *global round* can be used. It is also assumed that any message sent on a global round r_{global} is always received by the rumour's targeted receivers on the global round $r_{global} + 1$.

By examining the worst case in the semi-synchronous system, a probability P_i can be calculated. P_i presents the probability of a call handler of receiving any newer heartbeat value within i global rounds *if message omissions and call handler failures do not occur*. Approximate values for t_{fail} can then be calculated from the equation by giving some expected value for P_i. Calculations for finding the probability, as well as some calculations for t_{fail}, are presented in Appendix A.

5.1.4 Call handler restart

In real-life systems, it might not be desirable that once crashed server never restarts, or could not rejoin the system after restart. Some kind of restart

handling is needed. That is why some rules for detecting and accepting restarted call handlers is needed.

When a call handler restarts, it sets its heartbeat counter to zero. If heartbeat values are compared only by checking which value is bigger, restarted call handlers would go unnoticed until they reach the heartbeat value of the moment of the crash. That is why a special handling for restarts is needed, and a new *restart detection range* from zero to $t_{restart}$ is introduced as a special range of heartbeat values which indicate about recent start of a call handler.

When a call handler receives a heartbeat value within the range from zero to t_{fail}, the subject call handler of the rumour is flagged as restarted, and restart detection actions are done. After that, until the first heartbeat value larger than $t_{restart}$ is received for the first time, received heartbeat values of the restarted call handler are handled normally except for one exception: values less than t_{fail} are not considered as a sign of restart. When $t_{restart}$ is reached, the restart flag of the rumour target call handler is emptied and fully normal handling of heartbeat values continues. Heartbeat values less than t_{fail} are considered again as signs of restart. Restart detection range works thus as a buffer which takes care that heartbeat values are correctly handled even after a restart.

Selection of $t_{restart}$ must be done carefully for avoiding problems. If it is too large, quick and consecutive restarts are likely to go unnoticed. If the selected value is too small, some call handlers might fail to continue normal heartbeat counting and thus will erroneously think that the restarted call handler has crashed again. A good value for $t_{restart}$ is two times t_{fail}, when t_{fail} is selected such that there is a huge probability that all call handlers have received any new heartbeat until then. If so, there is a huge probability that all call handlers have received at least t_{fail} after $t_{restart}$ rounds, and thus problems should not occur.

5.2 Data dissemination

The purpose of the data dissemination protocol is to ensure that all call handlers know all changes in call statuses. The ideal solution would thus be one that forwards rumours fast and reliably to all members.

The implemented data dissemination protocol is loosely based on Epidemic Asynchronous Rumour Spreading, EARS, presented in [28]. The amount of call handlers is small and call handlers are equal, thus there is no need for hierarchical protocols. Also, when the network between call handlers is assumed to be equal, adaptive timeouts are not needed. A ma-

jor advantage of EARS is its less general gossip kind of nature. Instead of probabilistic certainty that all processes receive a rumour, the maintained informed-list helps to keep track on which processes have not received the message yet. Also, the receiver information is told to all processes when including the information in a message.

However, there are some minor disadvantages in EARS for the example system. First of all, the protocol is based on local steps, while the ideal case for the time critical example system is to forward new information as soon as required. Also, the original sender only sends rumours to one other process each step. This slows down the dissemination, and creates a possible single point of failure. That is why modifications to EARS are done.

The first modification is that instead of only one other process q, a rumour is sent to b_{data} other call handlers. From this part, the protocol leans slightly towards the spamming variation of EARS: SEARS [28]. Instead of recording to which processes a rumour has been **sent**, it is now recorded that which processes have *confirmed* that they have **received** the rumour. When forwarding a message, a call handler adds its own ID to a *received array* and adds the IDs of selected addressee call handlers to a *sent array*. The sent array tells to which call handlers the message has been sent but which have not confirmed the message as received, yet. The received array tells which call handlers have also confirmed the message as received. These arrays are then included in the rumour message.

A unique message identification number (message ID) is given for each data message when one is created for the first time. The same message ID is preserved in all created copies when forwarding the message. Additionally, each call handler separately maintains a table similar to informed-list of EARS in [28]. The table is called *received table* in this paper. It contains pairs of a message identification number and a list of call handler IDs. When receiving a message, a call handler adds the content of the message's received array to the received table.

The data message protocol is always initiated by a call handler only when there is a need to disseminate any data. A call handler creates a data message containing some data, adds itself to the received array and creates a unique ID for the message. Then it selects b_{data} other call handlers from the currently not suspected call handlers at random. IDs of the b_{data} selected call handlers are now added to the sent array of the message. Then the rumour is sent to the selected call handlers.

When receiving a data message, a call handler updates its received table with the information contained in the message's received array. If an entry for the message ID does not exist yet, the call handler knows it has not received the message yet, and so it also processes the message before creating an entry.

Now the call handler selects the next maximum of b_{data} call handlers with the following algorithm:

At first, maximum of b_{data} targets are randomly selected from the call handlers which do not appear in the received array of the message, nor in the received table for the message ID, nor in the sent array of the message. If the total amount of selected targets after the first step is less than b_{data}, additional targets are randomly selected from the sent array of the message until either b_{data} targets is achieved, or until there are not any not selected call handlers in the sent array anymore. Thus the total amount of targets might be less than b_{data}. The IDs of the selected call handlers are added to the sent array of the message. Also, the ID of the call handler and the content of the received table for the message are added to the received array in the message. The message is then forwarded to the selected targets.

The received table is used for reducing total message amounts. It is likely that most call handlers receive the same data message more than once when $b_{data} > 1$. When a call handler sends a data message to b_{data} others, none of the b_{data} call handlers is sure if other $b_{data} - 1$ targets actually received the message or not. Without received table, the received array in the copies of the original message would be growing only by one ID per forwarding. The received table brings additional information about receiver confirmations for messages. When several copies of a message are received by a call handler, it is able to combine information from received arrays of several messages, and eventually it might be able to notice that all call handlers have confirmed having received a message, even if the received array of a single copy does not include all possible call handlers yet.

Notable savings in total message amounts are achieved with the received table, as can be seen in test results presented in Section 7.2. A drawback of the table is that it requires additional memory. However, the content of the table is not large when call handler amount is not huge, so the additional memory requirement is tolerable.

A sent array is added to a message for faster spreading to all call handlers. Sent array keeps track on which call handlers have had the possibility to receive a message. It leads call handlers to forward a message primarily to those which did not even have the possibility yet. Downside of a sent array is that the size of a rumour message grows, but the growth is tolerable when the amount of call handlers is not huge. Slight differences in the speed of spreading are achieved with sent arrays, as can be seen in the test results in Section 7.2.

A special characteristic of the implemented data dissemination protocol is that the algorithm for selecting next targets is not fully random. Instead, information about those which have already received the message and which

might have received the message is used in the selection. It is both an advantage and a disadvantage. The semi-random selection algorithm provides certainty in dissemination, which normally lacks in gossip protocols: with the received array and received table it is taken care of that all call handlers will receive the message, at least as long as problems do not occur. A disadvantage is that never sending the message to a confirmed receiver reduces the amount of alternative paths that can be taken for delivering a message. For example, let us examine a case in which a call handler A is only able to communicate with a call handler B. If B initiates data dissemination by sending a message to b_{data} other call handlers, of which none is A, then A will never receive the message, because the message will never spread back to B. It is a major drawback, but the probability for this case to occur is tiny enough when the communication happens in a fully connected network. It is a case, however, of which it is good to be aware.

Another minor disadvantage is somewhat spamming nature of the protocol. Some of the call handlers which receive a message quite late will receive multiple copies of the same message. Thus they get burdened more than those which received the message earlier. However, the use of the received table somewhat reduces the effects of this disadvantage by lowering the total amount of messages significantly.

5.2.1 When to stop

Again, a question about when to stop forwarding a message is relevant. Call handlers are always added to a received array by themselves, never by another call handler. That is why it can be trusted that the confirmed receivers in the received array are correct, when message content changing byzantine failures do not occur. Thus also the information in received tables of call handlers can be trusted.

Now it can be defined that a message is forwarded only if there is at least one active call handler which does not occur either on the received array or on the received table for the message. That way a call handler may decide to not to forward a message anymore even if the received array of the message does not contain some of active call handlers, as long as all active call handlers are either in a received array or in the received table.

5.2.2 Ordering of data messages

In data messages, the order of messages regarding a same event has a great importance when processing them. Messages about different calls may be handled at any order.

In general, there are two different approaches to the message order problem. The first approach is to take care that messages are always delivered in a correct order. However, avoiding the incorrect message orders completely would require a strongly synchronized system, but that is not possible for the example system. The another possibility is to have a way to detect and handle cases in which messages have arrived in a wrong order.

A solution for checking the right message order is to use time stamps in messages. While it can be assumed that clocks of all call handlers are about the same, it cannot be relied much enough for reliable message ordering. Thus clock independent solutions are needed.

Some solutions for the message ordering problem would be acknowledging and voting. In acknowledgment, all members, call handlers in this case, have to accept the message before the changes caused by a message actually take place [26]. In voting algorithms, the receivers vote if a message was received on time, and the change only takes place if majority of receivers voted for it [27]. A detriment in both of the solutions is that delays between receiving the first message from messaging interface and finally producing a response back to it grow, if all call handlers would need to react. Thus neither of the solutions is suitable for the example system.

The implemented solution is based on counter values transferred within each message. The counter values must be selected such that the values in messages regarding same events must be comparable with each other. There are two data messages in which orders matter: call state change messages and takeover messages, which both are further presented in Section 6.3. Takeover messages contain both the heartbeat of a yielder and the heartbeat of an overtaker. Depending on the case, either one is used for comparison.

A problem with call state messages is that the owner call handler of a call might change if takeovers occur. Because heartbeat values of different call handlers are not comparable, the heartbeat counter of the owner call handler cannot be used. That is why a *session counter* for calls is introduced. When a call begins, the session counter is set to one. When a state change for the call occurs, the session counter is increased by one. Call state changes are updated only by the current call owner, that is why session counter can be trusted to be reliable enough. Now it is easy to determine, if a message is new or not: if a session counter in the message is bigger than the known session counter of the call, the message is new and the state of the call should be updated. Otherwise the message is ignored.

Ordering general message forward request, which are presented later in Section 6.3, is the most difficult case of all data messages. They can originally be sent by different call handlers, and they can be sent at the same time. One way for decreasing the amount of wrong orders is to send the message

always to the owner of the call for which the message it is meant, in addition to sending it to $b_{data} - 1$ others as well. However, the problem is still not eliminated. A reliable way for ordering them has not been found and is left for further studies.

5.3 Takeover protocol

The first problem of takeover protocol is to determine the rule along which the actual overtaker of calls of a crashed call handler is selected. Concepts of *monitor* and *monitored* are taken into use for the solution. Monitor is the call handler which will take over the calls of the monitored call handler if the monitored crashes. For example, if B is the monitor of A, then A is the monitored of B. Monitoring relationships are always maintained between active call handlers, which means that if crashes occur, monitoring relationships must be updated. As a result, when there are at least two active call handlers, each call handler has *exactly one* monitor and *at least one* monitored, of which only one can be active. If all call handlers in the system are active, each call handler has only one monitored, but if crashes have occurred, some call handlers might have more than one monitored.

Monitoring relations are decided using unique identification numbers (ID) of call handlers. Earlier the IDs space was defined to be circular, which is utilized in the selection. It is defined that the next value in the circle is the monitor, and the previous value is the monitored of any call handler.

When a crash of a monitored is noticed, a takeover occurs, during which the monitor takes over the calls of its monitored. Monitor becomes thus an *overtaker* and the monitored becomes a *yielder*.

The change in monitoring relationships caused by takeover last until the crashed call handler restarts. That is why call handlers maintain a table called as *takeover table* which contains the information about the takeovers it has heard about. The data contained in an takeover table entry is presented in Table 5.2.

Takeover protocol is initiated by a call handler B when it detects that it has not heard any new information about its monitored A for t_{fail} rounds. B first finds which is the new monitored after the crash, and notes how many crashed call handlers there are between the new monitored and the call handler B itself. All the crashed call handlers in between are then taken over by B and are now considered as inactive monitored call handlers of B. B takes the ownership of known active calls of all the monitored call handlers, and starts responding to the messages regarding these calls. Then B creates a takeover message, which contains IDs of all the call handlers which are

Data name	Type	Description
Renouncer	Integer	Unique ID of the renouncer call handler
Renouncer heartbeat value	Integer	The most recent received heartbeat value
Overtaker	Integer	Unique ID of the overtaker call handler
Overtaker heartbeat value	Integer	The overtaker's heartbeat value at which it took over the calls
Restarted	Boolean	True, if the renouncer node has been restarted after this entry was created.

Table 5.2: Entry in the takeover table.

now monitored by B. This message is forwarded to other active call handlers using the data dissemination protocol presented in Section 5.2.

When receiving a takeover message, a call handler first has to find out which call handler will most likely be the actual overtaker. This check is needed for avoiding problems caused by receiving messages in a wrong order because of asynchrony of the system. The receiver call handler checks its takeover table for an entry telling if there are takeovers for the original sender of the takeover message. If such an entry exists, the heartbeat values in the message and in the table entry are compared. If an entry does not exist or the heartbeat counter in the entry is older than the heartbeat counter of the message, the original sender of the message is considered as the actual overtaker. Otherwise, the overtaker found in the table entry is considered as the actual overtaker.

Additionally, it is checked if there are entries in which the yielder of the message works as the takeover: has the now crashed call handler previously taken over other call handlers. If such entries are found, the overtaker in the entries is updated to be the actual overtaker of the new crash case. Also, if the overtaker is updated, also the heartbeat counter in the entry is updated to the heartbeat count of the actual overtaker.

When processing a received takeover message, a call handler never checks if yielders have been marked as suspected in its own status table. Neither it never sets suspected flags as a result of takeover messages. Obeying takeover messages even if it is contradicting the call handler's own knowledge is a crucial property for avoiding conflicts with the failure detection protocol. It also helps preserving consensus between call handlers about the knowledge

on call owners. When a takeover message is received, the actual takeover has already happened: the original sender of the message already considers the calls as its own. If takeover messages were ignored by some call handlers due to conflicting information about call handler statuses, different views on call owners would occur, which would complicate the proper takeover of calls.

The submissive nature of handling received takeover messages is especially important in the cases of false failure detections. At any point, any call handler might falsely detect that its monitored has crashed. In that case, for a short period of time, there are two call handlers handling the same calls. In the case of false detection, it is probable that the monitored receives the takeover message, too, at some point. When receiving a takeover message telling that it itself is a yielder, the falsely detected call handler knows to stop handling its calls which had started before the time of the takeover. It then lets the overtaker continue handling them, even if there had not been any need for the takeover.

Chapter 6

Implementation

The implementation of the protocols is made as an inner module of a call handler. The module is called a *rumour module*.

The circular identification number (ID) space of call handlers is defined to be presented as integers. In the integer identification number space, the next ID is the next bigger integer, and the previous ID is the previous smaller integer. The next value from max integer is min integer, and the previous value from min integer is max integer.

The design of a rumour module and the most relevant inner components are presented in Figure 6.1. Red components are related to the data dissemination protocol, call handling and takeover. Components in green are related to failure detection. The components in yellow are general components needed by all protocols.

6.1 Components of the rumour module

ModuleMain component takes care of starting and controlled stopping of the rumour module. At the start, it creates other components and starts them as soon as everything has been properly created. At the time of a controlled stop, it tells other components to releases resources and to stop communication.

Server and *MessageParser* work closely together. Server listens a specified port for User Datagram Protocol (UDP) messages. When receiving a message, it just passes the read bytes to a queue in MessageParser. This way the server is ready for listening for more messages as fast as possible. MessageParser waits until read bytes are available in a queue. It examines the read bytes to see which message the bytes are presenting, and then converts the bytes to the object presentation of a message. Now it forwards

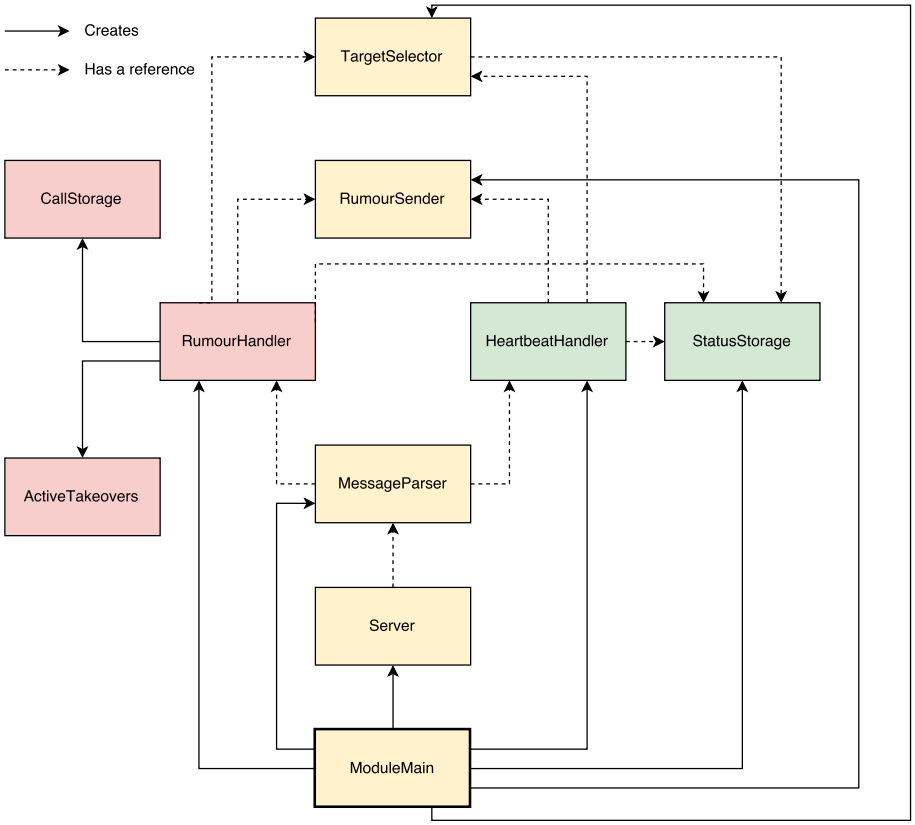

Figure 6.1: Rough inner design of a rumour module. Failure detection handling related inner components are presented in green, while data dissemination and takeover related are red. Components used by both protocols are presented in yellow.

the message to the relevant component: heartbeat messages are added to a queue in HeartbeatHandler, and any other message is added to a queue in RumourHandler.

TargetSelector is a component relevant for both failure detection and data dissemination. It selects addressed receivers for messages. The addressees for data messages are selected using the selection rules of data dissemination as described in Section 5.2, and the addressees for heartbeat messages are selected along the rules presented in Section 5.1.

For heartbeat messages, b_{hb} addressees are simply selected from among all active call handlers fetched from the *StatusStorage*. In the case of a data message, the selection process is longer. TargetSelector first checks the re-

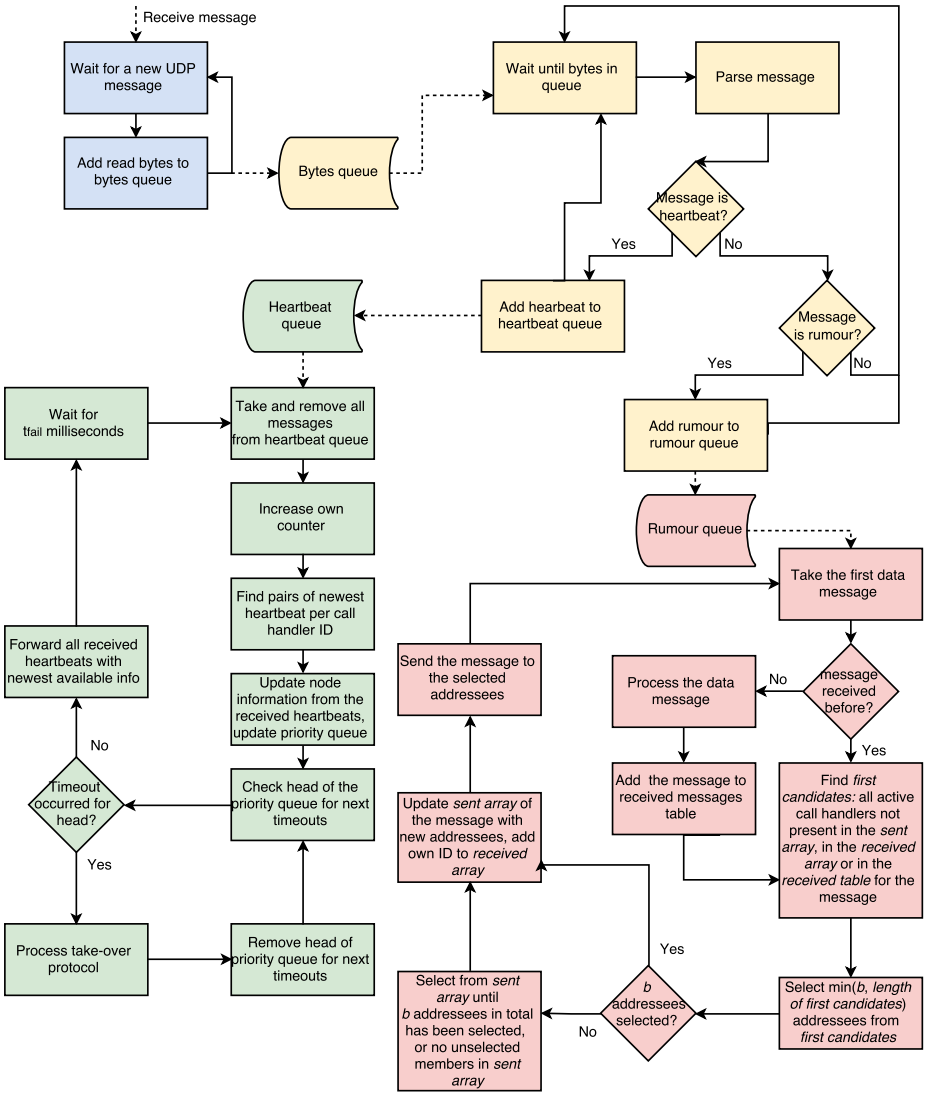

Figure 6.2: Inner function of a rumour module. Dashed arrows present how a message is transferred between components. Failure detection processing is presented in green, data dissemination handling in red, Server functionality in blue and MessageParser funtionality in yellow.

ceived array of the data message and the received table of RumourHandler for finding the call handlers which have not confirmed having received the message yet, at least as far as the call handler knows. The status of each potential addressee is then checked, and the call handlers flagged as suspected, are discarded. The remaining call handlers are divided into two categories:

potential addressees which show up in the sent array, and primary candidates which are not in the sent array and have not confirmed the message as received. At first, b_{data} addressee call handlers are selected from the latter category. If the category contained less than b_{data} members, the remaining addressees are selected from the sent array of the message. After the selection is done, a message and the selected addressees are sent to *RumourSender*. RumourSender translates the message to transferable bytes and sends them to the targeted receivers using User Datagram Protocol (UDP).

The main component in failure detection handling is *HeartbeatHandler*. It takes care of performing the steps of a round every T_{round} seconds, as presented in Section 5.1. HeartbeatHandler is the only only component which updates StatusStorage, even if other components have access to it as well. StatusStorage is the component in which known status of each call handler is saved, and which thus allows access to the status table.

RumourHandler handles all messages which are spread using the data dissemination protocol. Unlike HeartbeatHandler which works periodically, RumourHandler works only when a need arises. It sleeps until receiving a message to its queue from MessageParser, after which it processes the received message. RumourHandler creates and maintains two storages: *CallStorage* and *ActiveTakeovers*. CallStorage contains the minimum amount of required information about all ongoing calls. ActiveTakeovers contains the active takeovers table as presented in Section 5.3. In addition to the storages, RumourHandler maintains the received table needed in the data dissemination protocol. Another table handled by RumourHandler is an *unanswered messages table* which contains forwarded messages from outside of call handlers. The importance of the table is explained in Section 6.5.1.

The functionality of the module is presented in Figure 6.2. The functionality of RumourHandler and other data dissemination components is coloured in red. The functionality of HeartbeatHandler is coloured in green. Blue boxes present the functionality of Server, and the work flow of MessageParser is in yellow.

6.2 Implementation of failure detection

The presentation of heartbeat count is done with 32-bit integers. Integers provide large enough heartbeat range, having 2^{32} unique value. If, for example, T_{round} is one second, integers provide values for heartbeats for over 136 years. T_{round} should not be too small, because it creates a lot of additional overhead to the network if heartbeat messages are sent too often. On the other hand, having too large T_{round} slows down failure detection. That is

why a good value for T_{round} for the example system is one second.

After the start, a round is run every T_{round} second by HeartbeatHandler. A round is started by fetching all heartbeat messages from the queue and by emptying the queue immediately afterwards. The messages have been added to the queue by MessageParser. The heartbeat messages are iterated through and compared with each other. During the iteration, HeartbeatHandler updates a list of newest heartbeats. The list contains the heartbeats which are the newest received counts for each call handler. As a result, a list of call handler and heartbeat pairs are formed, one entry for each call handler about which rumours were received. The heartbeats in the list are compared to the current heartbeats in the status table, and the status table is updated when new heartbeat counts are noticed.

The check for detecting possibly crashed call handlers is implemented using a priority queue. In the queue, call handler IDs are ordered by how many local rounds there have been since receiving a new heartbeat count last time. When iterating through the list of received rumours, also the priority queue is updated when a newer heartbeat than in the status table is noticed. Updating the priority queue is done by removing an entry of the call handler in question and re-adding it to the tail. Remove operation executes in linear time and adding is done in $O(log(n))$ time [48]. As a results, the head of the priority queue contains call handlers with the longest time since their new heartbeats were received last time.

When the step of finding possibly crashed call handlers comes, the HeartbeatHandler checks the head of the queue. If the most recent heartbeat count of the head has been received more than t_{fail} rounds ago, the call handler at the head of the queue is considered as crashed. If the newly found crashed call handler was monitored, takeover is executed. The head is removed from the priority queue. Then new head of the queue is checked, and handled respectively. The same procedure is repeated until the new head of the queue should not be suspected. When such a head is faced, or the queue becomes empty if all other call handlers have crashed, HeartbeatHandler continues to the next step: creating a new heartbeat message. The implementation of the priority queue provides constant time for checking the head of the queue, and $O(log(n))$ time for removing the head of the queue [48].

Now HeartbeatHandler creates a new heartbeat message adding its own heartbeat counter to it. Also newest known heartbeat values for all call handler about which rumours were noticed during this round are added to the message. HeartbeatHandler requests TargetSelector to select b_{hb} targets for which the heartbeat message should be sent. RumourSender is then called for sending the message to the selected targets.

6.3 Data messages and data dissemination

There are several different kind of messages which are all processed by RumourHandler. They can be received through two different paths: from other call handlers through Server, or as internal messages from other modules of a call handler. The actual processing of a message depends on the type of the message. In total, there are four different kind of messages:

- State change messages
  Received from other call handlers through Server or internally from other modules. Presented in Section 6.3.1.
- Takeover message
  Received from other call handlers. Presented in Section 6.3.2.
- General message forward requests
  Received internally from other modules if the general message is coming from the messaging interface, or from other call handlers through Server. Presented in Section 6.3.3.
- Update calls notification
  Received from other call handlers. Presented in Section 6.3.4.

Further explanations about messages and about how RumourHandler handles them are presented in subsections. The bit-wise presentations of messages are presented in Appendix B.

6.3.1 State change message

The most common messages are *state change messages* which tell that a change in the state of a call has happened and which contain the new state presentation of the call. They are received either from the other modules inside the same call handler, if the call in question is currently owned by the call handler, or from other call handlers through Server, if the call is owned by another call handler. No matter which case it is, the message is processed in the same way.

State change messages for a call can only be created by the current owner call handler of the call. This is a major safety rule which is needed for message ordering and for avoiding conflicting messages related to a call being sent by different call handlers.

There are three types for a state change message: *begin*, *state change* and *release*. *Begin* indicates that the call has recently been restarted. When

receiving a *begin*, RumourHandler checks if a call with the given call identification number is known yet. If not, it adds the call to CallStorage. *State change* type indicates that the call has previously been started, but its state has changes somehow. When receiving a state change message of type *state change*, RumourHandler finds the call with the given call ID and checks if the received message contains new information about the call. If the message presented a new change in state, the state of the call in memory is updated. Otherwise the message is ignored. Determining if a state change message contains new information is explained in 5.2.2. *Release* type is used for telling that a call has ended and should be removed from memory. Also, the ID of the call is added to a constant size round buffer named *released calls array*. The purpose of the release calls buffer is explained in Subsection 6.5.2.

No matter what type the state change message is or if it was received from other call handlers or modules, the next step for RumourHandler is to forward it using the data dissemination protocol presented in Chapter 5.2. RumourHandler adds itself to the received array of the message, asks TargetSelector to select the addressees, and then lets RumourSender to send the message to the selected call handlers.

Change state messages always contain the full presentation of a call state. While it makes messages larger than if they contained only new changes, it helps taking care of that call handlers are unanimous about call states. If the messages contained only changes and each call handler had to compute the new state by themselves, problems could occur if receiving messages in a wrong order. Computing the changes in wrong order could lead to wrong results. Alternatively, a rule should be found out on how to handle the cases in which it is noticed that some messages are missing in between. Both of the problems are solved by sending the full state presentation. For example, if a call handler gets a call state change message with session counter larger than the next from the previously known counter, it is known that some state change is missing from the between. However, the update of the call in memory can be done, because of getting the full state in the message. If receiving the missing call state message later, the message can just be ignored, as well as if receiving a message with smaller session counter than the most recently known.

6.3.2 Takeover message

Takeover messages can only be received from other call handlers. Their purpose is to inform other call handlers that the sender of the message has taken over calls of another call handler, and is now the owner of the calls. When receiving a takeover message, a RumourHandler first checks if the

information is new. Determining if the information is new, is a problem caused by message orders, and it is explained in Section 5.2.2. If the message contains new takeover information, the takeover is handled as described in Section 5.3. In either case, the message is then forwarded using the data dissemination protocol.

6.3.3 General message forward request

General message forward requests are special kind of wrapper messages. General messages are originally received from the black box through the messaging interface. Each call is only owned by one call handler at a time, and thus each general message has an intended target call handler. However, if the message is sent by the messaging interface to a call handler to which it does not belong, the call handler forwards the message to others. Thus general messages can either be received from other call handlers, or from other modules inside the same call handler. A general message forward request is a data message which contains one general message from outside the network of call handlers.

When receiving a general message from the messaging interface for a call of another call handler, the other modules of the receiver call handler forward the message to the rumour module. RumourHandler then tries to find a call from CallStorage matching the information of the message. If a call is found, the ID of the call is set to the general message forward request for helping other call handlers to identify the call for which the message is meant for. Then the ID of the call handler is added to the received array, and the message is forwarded using the data dissemination protocol.

When receiving a general message forward request from another call handler, RumourHandler first tries to find the call for which the message is meant. If the call is found, and the call is owned by the call handler itself, RumourHandler forwards internally the general message inside the general message forward request to other modules which are responsible of handling such messages. In that case, the general message forward request is not forwarded to other call handlers anymore. However, if a call is not found or it is owned by another call handler, the general message forward request is forwarded using data dissemination protocol. The inner message is also saved in memory, purpose of which is explained later in Section 6.5.1.

6.3.4 Update calls notification

When a call handler restarts, it has lost all data of the ongoing calls. For fixing the situation, *update calls notifications* are used. When a call handler

notices that any other call handler has restarted, it creates update call notifications from all ongoing calls it knows and sends the message directly to the recently restarted call handler.

When receiving an update calls notification, RumourHandler updates its call storage with the calls from the messages. If the call storage already contained some of the received calls, the call information with newest call session counter is selected as the current state of the call. The update calls notification is never forwarded after that.

6.4 Takeover

When HeartbeatHandler notices that any new heartbeat count about the monitored has not been received for at least t_{fail} rounds, the takeover protocol is started. The status of the monitored is flagged as suspected, and the information about monitored crash is forwarded to RumourHandler. RumourHandler finds the new monitored, and takes over calls of all call handlers between the new monitored and itself. An internal message is created containing data of all calls that were taken over at this step. The message is then sent internally to other modules, which are take care of call processing. Entries of all the takeovers are created or updated in the takeover table.

Then a takeover message is created and the IDs of all call handlers which were taken over by the call handler as a result of the monitored crash are added to the message. The takeover message is then forwarded using the data dissemination protocol.

6.5 Special cases

There are some cases which require special handling. They are mostly caused by asynchronicity of the system, or preparing for real-world situations. Some of the cases are presented in this section.

6.5.1 Crash before responding to the messaging interface

Some of the messages received from the messaging interface require a response. The responding call handler is most often the one which handles the call about which the message is. A response is sent before a state change message is created and disseminated. If crash of a call handler happens at the moment when a response is expected, but the call handler did not have time

to send it, the overtaker of the call should send the response instead. For that reason, a table named *unanswered messages* is maintained. When forwarding a general message forward request, the general message is also saved in the unanswered messages table, along with the current session counter of the call to which the message is related.

When detecting a crash, the monitor of the newly crashed call handler performs the takeover protocol. After starting handling the calls, the unanswered messages table is checked for all calls that are taken over. If the most recent session counter of a call is newer than the session counter of the unanswered messages table for the call, all the messages have been responded, and the overtaker does not need to process the messages in the table. However, if the session counter of a call is equal to or less than the session counter in the unanswered messages table, additional processing is needed. The overtaker then processes the unanswered messages in the same way than when receiving a general messages for own calls. As a result, responses are sent.

6.5.2 Receiving messages after release

A special case of a wrong message orders is when state change messages are received after a release message. As a result of a release message, a call and all related information is removed from memory. Consequently, if receiving a state change message for a call after the call has already been released, it appears for a call handler as if there is a new call for which begin message was not received. That is why additional information about already processed and ended calls is needed.

A simple solution is used. A circular buffer of fixed-size s called *released calls array* is used for saving s most recent call IDs of released calls. When receiving a state change message for an unknown call, the released calls array is checked. If the call ID of the state change message is found in the buffer, the state change message is about an already released call and the message can be ignored. If the call ID is not found in the buffer, the state change message is probably about a new call. This solution radically decreases the amount of re-adding already released calls, but it does not fully eliminate the cases.

6.5.3 Controlled stopping of a call handler

In real-life systems, it is good to be prepared for maintenance work which might require manual restart. While the failure detection protocol could be let to take care of manual restarts, too, a better solution would be to provide

a way for controlled shutdown. In the case of call handlers, the controlled shutdown would allow faster takeover of active calls.

For this purpose a *unloading notification message* is sent by a call handler when a controlled shutdown is requested for it. The message is only a simple notification message, which will be sent once to other call handlers and which will not be forwarded anymore by the receivers. When receiving an unloading notification message for the monitored call handler, the message is handled by RumourHandler in the similar way to the normal monitored crash, and thus the control of calls is moved faster to the monitor call handler.

Chapter 7

Protocol testing

Tests are run separately both for the whole functionality of the system and for data dissemination and failure detection protocols of the system. This chapter presents the test cases for two of the protocols: failure detection and data dissemination.

7.1 Testing failure detection

Tests of the solution for the failure detection problem cover heartbeat sending and failure detection algorithms.

7.1.1 Test setup

Having a test system with an access to PSTN would be both expensive and irresponsible, because of possible risk of congesting the network. That is why a closed environment with virtual machines for call handlers is used in testing. Due to lack of PSTN connection, only SIP user agents are used for calls. Also, it is not possible to deploy call handlers geographically dispersed. Consequently, the tests do not examine how the solution works for geographically distributed system.

Servers and virtual machines and their purpose in the test setup is presented in Figure 7.1. A rack server with limited resources is working as a virtualization platform which hosts virtual machines used for call handlers. It has CentOS 6.6 as its operating system, and it has four 2400 MHz central processing units (CPU) and 4 GB of memory available. It runs OpenNode Linux 6.6, with which ten identical virtual servers have been created. Each virtual server has been installed with CentOS 6.5, they have 300 MB of memory reserved, and they share the host server's CPUs. Virtual servers

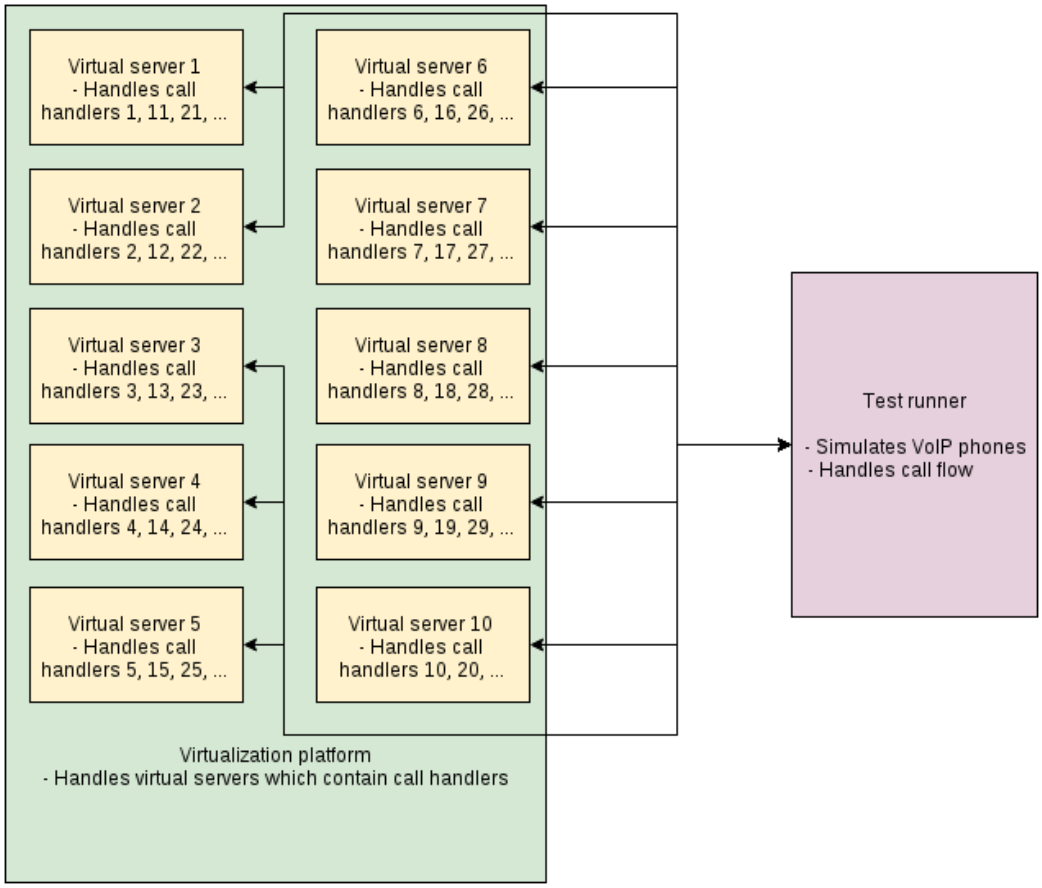

Figure 7.1: Servers of the test setup.

use 64-bit OpenJDK 1.8.0 for running call handlers. Depending on the total call handler amount in a test case, each virtual server provides resources for one or more call handlers. Servers are behind the same network router, thus communication between call handlers and test runner happens only within the network.

Test runner is a virtual server with CentOS 6.5 as its operating system, 500 MB of memory and access to one 2500 MHz central processing units (CPU) of its host server. It takes care of running the components of "black box" as presented in Chapter 2. Test runner also commands SIP client handler, test robot and test scripts. The SIP client handler is a C-based program which takes care of registering VoIP numbers and handling SIP signaling for the clients. The test robot is a Ruby-based program taking care of which call cases should be done, when and how many. It commands SIP client handler to start, answer or release a call. The last level of the

test runner consists of Bash scripts. These scripts change automatically test parameters for test cases, configure call handlers, define how long tests should be running, and starts the test calls by calling the test robot. After each test case, the scripts gather logs from all test environment components. The scripts also take care of restarting call handlers and other test components at the beginning of each test case for avoiding test cases potentially affecting each other. Test cases are run sequentially, never parallel.

The base unit of test cases is one *round*. One round is a time during which in total 10 concurrent calls are made, and the total time of a round is about one minute. For the first 10 seconds, one call is started every second, ten calls in total. Then the calls are answered every 2nd second, and then hanged up every 2nd second. Thus the time of an answered call is 30 seconds. The time of one test case is set to be about two hours. When the base unit of tests is a round which lasts for one minute and during which 10 calls are made, the expected total amount of calls for each test case is $2*60\ minutes*10\ \frac{calls}{minute} = 1200\ calls$. However, calls are only done in the whole system test cases presented in Chapter 8.

Failures are simulated by killing immediately a call handler that is due to face a failure. Killing the process is done by sending a "SIGKILL" signal to the call handler process using a Linux command "kill". The result of the signal is that a process ends immediately without terminating correctly.

A huge amount of log data containing every action of test components is created during test cases. In the larger amounts of call handlers, one test case could create log data up to 5 GB. That is why also several parser programs have been written for parsing the relevant information from the logs.

7.1.2 Test parameters

Test parameters are needed for examining dependence of test results on different properties of a system. Each test case consists of different set of parameters. For the example system, the most relevant test parameters are:

- Call handler related parameters
  - Call handler amount, n
  - Data dissemination connections, b_{data}
  - HB connections, b_{hb}
  - Heartbeat failure detection limit, t_{fail}
    Base unit is one local round T_{round}, which is presented in seconds
- Testing system related parameters

- Failure probability, f
  Means the probability of $\frac{1/f}{s}$, thus base unit is one per second, $\frac{1}{s}$. Smaller f means larger probability, and vice versa.
- Restart probability, f
  Means the probability of $\frac{1/f}{s}$, thus base unit is one per second, $\frac{1}{s}$. Smaller f means larger probability, and vice versa.

Call handler related parameters are some of the most important configurations of the example system: how many call handlers there are in total, to how many call handlers heartbeat and data messages are sent, and what is the timeout for failure detection. Testing system related parameters, again, provide simplified information about the real world system: how probable it is that one call handler crashes independently of others, and with which probability it is automatically restarted after a crash. For getting more meaningful results in test, failure probability is exaggerated.

The amount of call handlers, n, can be presented with a positive integer and values of data and heartbeat connection with an integer between [1, n-1]. The value of failure detection timeout, t_{fail}, can be presented with a positive integer, which tells the amount of local rounds. Time of one local round, T_{round}, in all test cases is 1 second, thus the base unit of t_{fail}, in practice, is one second.

Positive integer can be used for f for both failure and restart probability. The value means that every second each call handler has a $1/f$ probability of crashing or restarting, if it had previously been crashed.

As a result of the large amount of possible values for each parameter, there are numerous different combinations of test cases, and they would take huge amount of time to run. Furthermore, running tests for all possible values for failure and restart probabilities would not bring much additional information; there is not much difference if each call handler has a $\frac{1}{2000}$ or $\frac{1}{2001}$ probability to crash every second. That is why a good set of test parameters are selected beforehand.

From the nature of data dissemination algorithm it can be noted that data connections amount should be small; otherwise message amount grows largely. That is why the value is set to 2: data messages are always forwarded to only two other call handlers.

Estimates about ideal values for heartbeat connection amount and failure detection timeout can be calculated for each call handler amount. For that, the calculations presented in Appendix A are used. Worth noting is that the calculations are done for a semi-synchronized system in which all local rounds happen at the same time. That is why the real ideal values likely are something else than the values used in tests.

It was noticed in initial tests that larger probabilities for a failure are more likely to cause problems. The effect of restart probability to reliability, however, appeared to be indirect: the faster a call handler is restarted after a crash, the faster it could crash again. That is why restart value is set to an arbitrary constant smallish value 300.

With these decisions, the amount of configurable parameters can be reduced to only two: the amount of call handlers and the failure probability. Other parameters are constant (data connection amount and restart probability) or are dependent on the call handler amount (detection timeout and heartbeat connection amount).

One important testing system related parameter is left out: probability that a single message gets lost in network. Because of lack of it, tests cover only crash failures, but do not cover omission failures. Testing omission failures would have required even more complex testing system and more time. Thus it is assumed in the tests that almost all of the messages are correctly delivered.

7.1.3 Running tests

Run time for test cases for failure detection is about 2.5 hours. A constant, smallish value for failure probability is selected, because it should only affect on the amount of expected crashes and restarts, not to the actual detection processing. That is why the failure probability is set to 300, which means that each second each call handler has a 1/300 probability to crash.

Initial tests showed that the virtualization environment cannot handle correctly large amounts of call handlers. That is why tested call handler amounts are all values between [3,10] and all even values between [12,20].

Testing environment and call handlers separately log every action they take during test cases. Time stamps of the logs are used for ordering the events for creating a full timeline of crashes, restarts and detections. The virtual servers of call handlers and the test runner are not perfectly synchronized, but their clocks are very close to the same time: there is less than one second difference, maximum, between any two servers. Because the smallest significant time unit in logs is second, time stamps in logs can be viewed as reliable enough for testing purposes. Missing detections and faulty detections can then be found from the timeline.

In some cases, missing detections are left out of the final results. These are borderline cases, in which it is not sure if not detecting an event should be viewed as violation of completeness or as normal behaviour. A missing detection is not considered as a failure of call handler B to detect the crash of a call handler A if:

- B does not notice the crash of A when A is restarted less than t_{fail} time after the crash.
  If a call handler A is crashed for less than the detection limit time, it is expected that other call handlers do not have enough time for noticing the crash; it is normal behaviour. However, if call handlers fail to notice the restart of A, it is considered as a failure. Call handlers should always notice restart of the heartbeat counter of another call handler even if they did not know that the call handler had been inactive.

- B does not notice the crash of A and B crashes within detection limit time from the crash time of A.
  In this case B does not have enough time to detect the crash of A.

- B does not notice that A has been offline, if B is restarted and within less than detection limit time also A restarts.
  When B restarts, it assumes that all known call handlers are active. If A is restarted before B reaches heartbeat counter value of detection limit time, B could not have known that A was not available.

7.1.4 Criteria

Main properties of failure detectors are completeness and accuracy [10]. Completeness, in this case, means that not only each crashed call handler, but also restarted call handler, is eventually detected. Likewise, accuracy for the example system means that no faulty start detection nor faulty crash detection occurs. Thus examining the results from the point of completeness and accuracy brings valuable information about how well the algorithm works.

Another expected property of failure detection in the example system is that detections occur fast enough: "eventually" of completeness should be short time. Thus the third criterion is the time that it takes before a call handler detects a crash or restart.

Differences in importance of the three criteria for the example system exist. Completeness together with time should be as good as possible, because otherwise takeover of calls might not occur, and data will be lost. Faulty detections, on the contrary, most probably would not cause data losses, because of call handlers staying active and because of how takeover protocol works. Faulty detections also have been taken into account already in implementation.

7.1.5 Test results

Detection times and amounts of undetected events are counted from the timeline, and the exact counts and average times are presented in Table 7.1. Average detection times are used because more interesting than to examine how fast a single call handler detects a crash or restart, is to examine how fast in general call handlers are able to notice events. Detections are always independent actions, which do not depend on detections of other call handlers. The values in the table are shown per call handler amount and the detection timeout is included for a reminder.

The values presented for stop detection times are *average stop detection times* ("Avg stop detection time" in the table) and *stop detection time after restart* of the detector ("Avg stop (restart)" in the table). Stop detection time after detector restarts tells that how fast after its restart a call handler detects another call handler, which was not active at the time of restart. Stop detection times do not include the detection times after restart. These two values are not directly comparable between different call handler amounts because the ideal stop detection time depends on the timeout value, and the timeout is different for different call handler amounts. That is why two additional values, which are dependent on the detection timeout, are used. *StopDT/t* presents the average stop detection time compared to the detection timeout value. Respectively, *StopRDT/t* presents the average stop detection time after the detector has been restarted recently compared to the detection timeout value. The need for separately tracking detection time after restart is because the test environment writes down the time stamp when it called start for a call handler process, but it takes some seconds before the process actually is up and running. The time that it takes before a call handler is fully functional since restart is included in StopRDT/t times.

Restart detection times are presented by *average start detection times* ("Avg start detection time" in the table). Unlike in ideal stop detections, ideal start detection time does not depend on detection timeout limit, because restart should be detected as soon as a restarted call handler sends out the very first heartbeat value. That is why the ideal value would be 1 and the values can be directly compared between different call handler amounts. However, the values presented in the table also include restart time of a call handler, which usually varies from 5 to 10 seconds.

StopDT/t, StopRDT/t and average start detection time are the most relevant values for examining detection times because of the values being comparable. Figure 7.2 shows the change of the values depending on the call handler amount. Even if steady change is not directly visible from the values, a slowly growing trend seems to exists in all three values. It would

Call handlers	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Timeout	5	12	8	11	15	18	12
Avg start detection time	6.48	6.77	6.95	6.54	9.93	7.64	6.79
Avg stop detection time	6.19	13.24	9.39	12.30	16.79	19.41	13.30
Avg stop (restart)	9.90	17.33	14.02	16.78	20.94	24.48	19.86
StopDT/t	0.95	1.96	1.35	1.88	1.69	2.54	1.96
StopRDT/t	1.60	1.31	1.49	1.36	1.25	1.26	1.49
Detections	89	201	336	465	694	927	1310
Undetected	0	0	0	0	3	2	0
Faulty Detection	0	0	1	0	0	0	0
Call handlers	**10**	**12**	**14**	**16**	**18**	**20**	
Timeout	14	11	14	16	13	11	
Avg start detection time	7.49	6.98	8.20	8.17	7.81	7.76	
Avg stop detection time	15.37	12.59	15.48	17.65	14.49	12.50	
Avg stop (restart)	22.26	19.09	24.18	24.97	21.94	19.81	
StopDT/t	2.05	1.80	1.89	2.16	1.86	1.61	
StopRDT/t	1.45	1.52	1.56	1.42	1.51	1.59	
Detections	1683	2187	3075	4369	4836	6511	
Undetected	0	0	11	5	1	0	
Faulty Detection	0	3	2	7	1	3	

Table 7.1: Average detection times for start detections, stop detections, and stop detection when the detector has recently restarted.

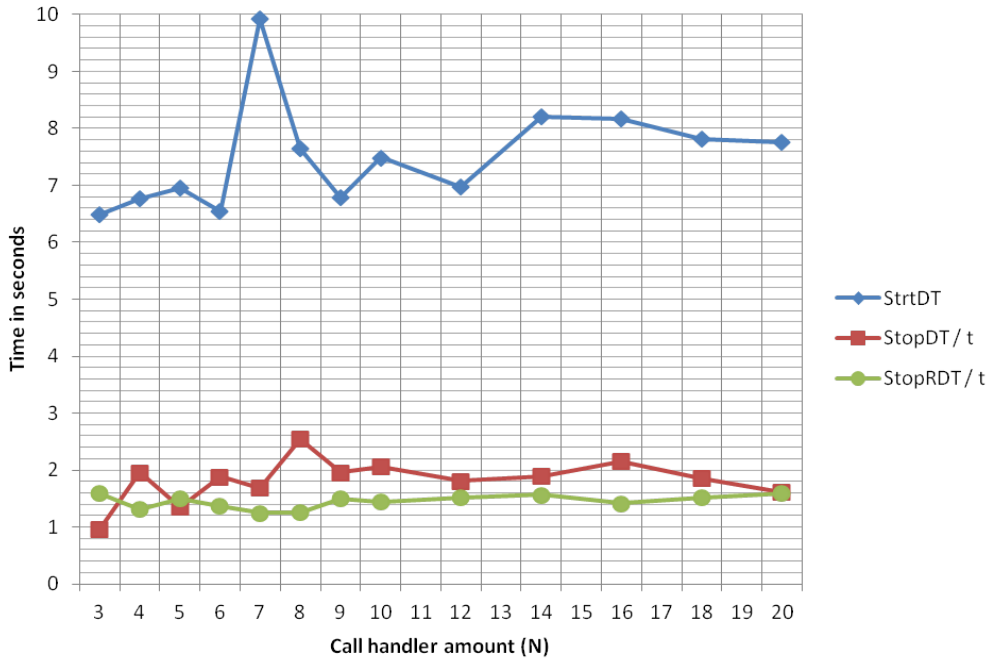

Figure 7.2: Values of start detection times (StrtDT), and stop detection time and stop detection time after recent restart related to timeout (StopDT/t and StopRDT/t). Time is presented in seconds.

seem that the more call handlers there are, the later events are detected on average.

Reasons behind detection getting slower when call handler amount goes up lies in the heartbeat algorithm. At each point of time, there are most probably several different heartbeat values for a call handler circulating between call handlers. As a result, some of the call handlers receive some information later than others, as long as the amount of heartbeat connections is less than the total amount of active call handler minus one $b_{hb} < N_{active} - 1$. The more call handlers there are, and the smaller b_{hb} is compared to the call handler amount, the more time spreading the newest information most likely takes. Growth of stop detection after restart time is gentler than that of stop and start detection times. The reason for it is that B does not have information about the stopped call handler A at the timer of its restart. When the first heartbeat value about A is received, it will restart counting for the detection timeout. If only short time has passed since the stop of A, there might be different heartbeat values for A still circulating between call handlers, and B might receive an old value, which slows the detections when new information

is later received by B. However, if long time has passed since the crash of A, all active call handlers most likely already have up-to-date heartbeat value of A. In that case, when B restarts, it receives the correct crash time of A right away, and thus it can detect the crash as soon as it reaches own heartbeat count t_{fail}+(heartbeat count when it first time received information about A).

With 7 and 10+ call handlers average detection times are affected by a bug in the restart detection implementation. Most notable result of the bug is seen in the call handler amount 7, in which start detection time suddenly makes a spike up. With only 7 call handlers, the total amount of start detections is low for the test cases, and thus four large times (179 s) caused by the bug raise the average largely. When counting the average without the four values the result is 6.79 s. In other test cases, the bug causes from 0.03 s (in stopDT/t for 18 N) to 1.96 s (in stopRDT/t for 14 N), however most often less than 1 s, higher values. In stop values the larger average is not as clearly visible, because of comparing it to the timeout, but in start values results from the bug cause additional variance.

The bug is caused as follows. Call handlers have to be able to notice restarts even if they did not notice any crash. It is a needed functionality for the real life system: manual system restarts must be handled correctly, and it is a safety feature for faulty crash detections. For detecting these restart cases, restart limit range $[0, t_{restartlimit}]$ was introduced in Section 5.1.4. When a call handler B first time receives a heartbeat value within restart range from call handler A, B assumes that A has been restarted. If A, however, crashes again in t_{crash} when $t_{crash} >= 0, t_{crash} <= t_{restartlimit}$, problems might occur, because of incorrectly not setting suspected flag again. When A restarts again, B will not notice the new restart because it had not marked A as suspected, and it will also start updating heartbeat value only when the received new value is larger than t_{crash}. If t_{crash} was larger than detection time limit, A will not be accepted as a working call handler before another recently restarted call handler happens to send a heartbeat message to A. The bug was fixed only after tests were done.

"Detections" row in Table 7.1 shows the total amount of correct detections done by all call handlers. The value is calculated by adding together the amounts of start detections, stop detections and stop detections after restart. Row "undetected" shows the amount of cases in which a call handler did not log about noticing restart or stop of another call handler. For example, there are 20 active call handlers and A stops. Now it would be expected that 19 detections about the case are done. If only 17 call handlers noticed the stop, the amount of undetected is increased by 2. Row "faulty detections" in the table shows the total amount of faulty stop, stop detection after detector

restart and start detections. A typical case is that a faulty crash detection is done. As a result, when the detector most likely receives a new heartbeat value soon, it will also detect restart. These cases are only counted once for faulty detections, because restart detection is a wanted property in this case. Uncertain cases, in which test log parser could not reliably classify are added to faulty detections, too.

From the point of completeness, the implementation is not perfect, but it is working well enough. Even if undetected amounts vary a lot between different test cases, some sort of growing trend can be seen. It means that when the amount of call handlers grows, so grows the amount of unnoticed crashes and restarts.

Accuracy by the test results is not perfect, but adequate. In smaller amounts of call handlers, which is up to 10 call handlers, faulty detections practically did not occur during tests. The likelihood of faulty detections occurring seemed to grow slightly with the amount of call handlers. However, since uncertain cases were classified in faulty detections by the test log parser, the actual amount might be less than in the results. The uncertain cases contain cases in which at some points of time on a timeline there were so many events in unconventional order that the test log parser was not certain about which detection is for which event. In some cases, it might not have been clear even if checking the cases manually. Thus they were counted as faulty detections just in case. During all the test cases only two certain faulty detections occurred: both happened for call handler amount of 20, and in both of the cases restart detection was done only one second after the stop detection.

7.2 Testing data dissemination

The second major sub-problem of the example system is sharing information between call handlers. For that purpose a data dissemination protocol is used. The function of it is to take care of disseminating data messages so that all call handlers will eventually receive the message with high enough probability.

7.2.1 Test setup

Tests setup of data dissemination tests differs from other tests. Instead of testing in a full system, only the data dissemination algorithm part is extracted and run separately on much lighter setup. Instead several call handlers on different servers, several threads are used, one thread pretending

to be one call handler. This way testing of data dissemination can be done on a single machine and failures of the testing environment itself are less likely to occur.

Test machine is a desktop computer with 3.6 GB memory, 2 times 800 MHz processors and 64-bit Debian 7.8. Dissemination tests are written in Java and run with Oracle's Java version 1.8.0_25.

First all threads are started. Then one message is sent to one randomly selected call handler thread, at which point the start time for tests is set. When a call handler thread forwards the message, a counter is increased by one. The counter also takes care of checking if all call handler threads have already received the message, and at which point the last call handler received it. The test for each test case is run 100 times, and average values of the 100 runs are calculated.

Tests are run for four different versions of the algorithm for showing the concrete importance of design decisions presented in Chapter 5.2: forwarding within the message a sent array tracking call handlers to which the message has been sent, and maintaining received table within all call handlers separately. These two designs are called **additions** in these tests. Four different versions of the algorithm are:

- Basic algorithm: not using either of the additions ("Neither" in figures)
- Using only sent array which tracks call handler to which the message has been sent ("Sent array" in figures)
- Using only the received table ("Received table" in figures)
- Using both the sent array and the received table ("Received table and sent array" in figures)

7.2.2 Criteria

Dissemination should happen fast enough, because a response to messages is always expected to be received within a short, but undetermined, time. That is why one main property for tests in the data dissemination algorithm is time: how long it takes before all call handlers have received data at least once.

Reliability for the algorithm is achieved in the algorithm by forwarding the message until all call handlers have signed it as received. This creates numerous copies of the message and causes that the same message is handled several times by some call handlers. However, copies of a message create additional overhead to network, which is something that should be avoided as

much as possible. So, the total amount of sent message per one data message should be as small as possible. Thus the second criterion for dissemination algorithm is the amount of sent messages.

A major property of dissemination algorithm, completeness, meaning that all call handlers should eventually get each message, is not specifically tested in this section. Reason is that completeness is taken care by the algorithm design as long as crashes and message omissions do not occur. Of course in real world failures do occur, but in that case completeness of the data dissemination algorithm is heavily affected by how fast and how well failure detection algorithm works. Consequently, it would be difficult to test data dissemination protocol reliability separately from other algorithms. Instead, it is tested as a part of the whole system test presented in Chapter 8. Additionally, if some call handler should not receive a message, it is detected by tests.

7.2.3 Test results

The effect of change in b_{data} was studied with tests for call handler amount of 20 and data connection amounts from 1 to 19. The average amounts of sent messages until stop, which is when none of the call handlers decides to forward the message anymore, are shown in Figure 7.3. Starting already from data connection amount of 2, a huge difference in message amounts is present between using neither of the addition or only sent array, and using either received table or both received table and sent array. That is why closer look on same data for latter two cases is done in Figure 7.4. The effect of a call handler amount in average sent message amounts was tested with call handler amounts from 3 to 10 using constant data connection amount 2. Results of the tests are plotted in Figure 7.5.

Increasing data connection amount is largely affecting the message amounts. Remarkably rapid growth is present in the basic algorithm and in the algorithm which uses only sent array. For example, when data connection amount is 3, the amount of sent messages is about 250 000 sent messages for spreading one message across a system, and when using data connection amount 4, the amount is already around 350 000 messages. These amounts would be unbearable for a time critical real-life system. A similar growing effect of data connection amount can also been seen in algorithms using either a received table or both received table and sent array, but only in much smaller scale. While the need to send on average about 650 message for spreading one message across 20 call handler when b_{data} is 3, is still a lot, it is, nevertheless, within bearable limits. The results show that raising data connection amount will create a lot of overhead to a network, and that is why small

values are recommended.

When keeping b_{data} constant, call handler amounts affect less on the sent message amounts. Nevertheless, a growing difference between using received table about how not using work it is present starting from call handler amount 4.

Reasons for the differences can be explained with how the algorithm works. The additions provide additional information which is effectively used for reducing message amounts. Received table alone is able to bring message amounts down. Received table causes call handlers to remember which call handlers have already earlier confirmed having received a message, thus it allows combining receiver information from several messages. Thus also information about confirmations spreads faster in a network.

Sent array alone does not provide additional information about which call handlers have already received a message, thus it alone does not reduce message amounts. But when combining it to received table, results are even better than when only using received array. This is the case especially for data connection amounts larger than 2. The reasons behind this effect are in how sent array controls to which messages are sent next. A message is primarily sent to those who are not present in the sent array of a message, and it helps the messages spread to all call handlers faster, as will be pointed out later. The later in the chain of receivers a call handler is when first time receiving a message, the more likely it receives several copies of the same message early on. These copies are likely to contain different confirmed receivers, and thus the call handler is able to gather receiver information from the messages and forward larger lists of confirmed receivers.

Running tests on one machine and within one process caused a problem when testing for dissemination times: messages delivery times were negligible, while there is always some delay in the real system when transferring a message between servers. That is why artificial delay was added. When a call handler sent a message to another call handler, constant time of 100 milliseconds[1] was added to the delivery time in time tests.

Figure 7.6 presents tests results for average time in milliseconds before all call handlers have received a message when the call handler amount is 20 and b_{data} is from 2 to 7. Differences between different versions of the algorithms are not as drastic as in the case of total message amounts. The versions using a sent array are performing slightly better, but the differences are not large. More visible difference in times can be seen in Figure 7.8, in which call handler amount varies from 3 to 10 and data connection amount is constant

[1]Time might have varied slightly depending of the CPU time allocation, however, the variance is insignificant in this case.

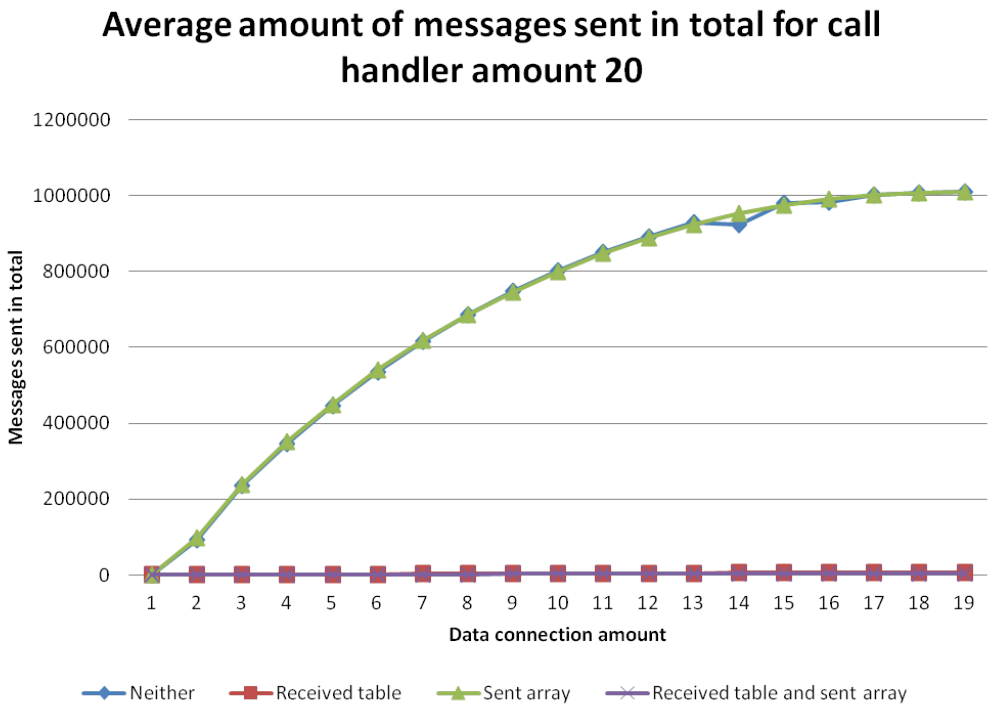

Figure 7.3: Total amount of messages sent from the first message until the stop of message forwarding.

2.

Another point of view for dissemination speed is the amount of sent messages at the first point of time when each call handler has received the message at least once. Larger differences between different versions of the algorithm can be found from this point of view as can be seen in Figure 7.7. Additionally, Figure 7.9 shows the amount of sent messages at the point when all call handlers have received the message once when call handler amount varies from 3 to 10, and b_{data} is constant 2.

The received table does not seem to have any effect on how fast messages have been delivered to all call handlers at least once. On the other hand, the sent array seems to speed up dissemination slightly. Reasons behind the effect of a sent array is in that messages are always forwarded primarily to call handlers which do not appear on received array or in sent array. Thus call handlers will not send a message to the same targets than those which were earlier in the chain of receivers. Hence a message spreads faster to all call handlers.

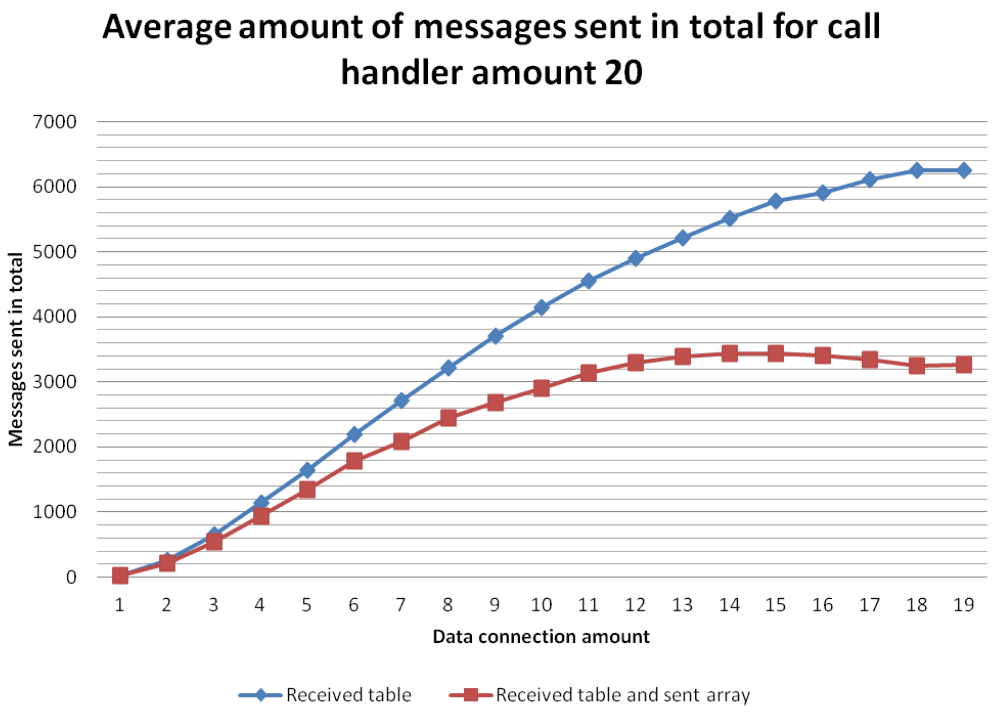

Figure 7.4: Total amount of messages sent from the first message until the stop of message forwarding in cases of using only received table or using both received table and sent array.

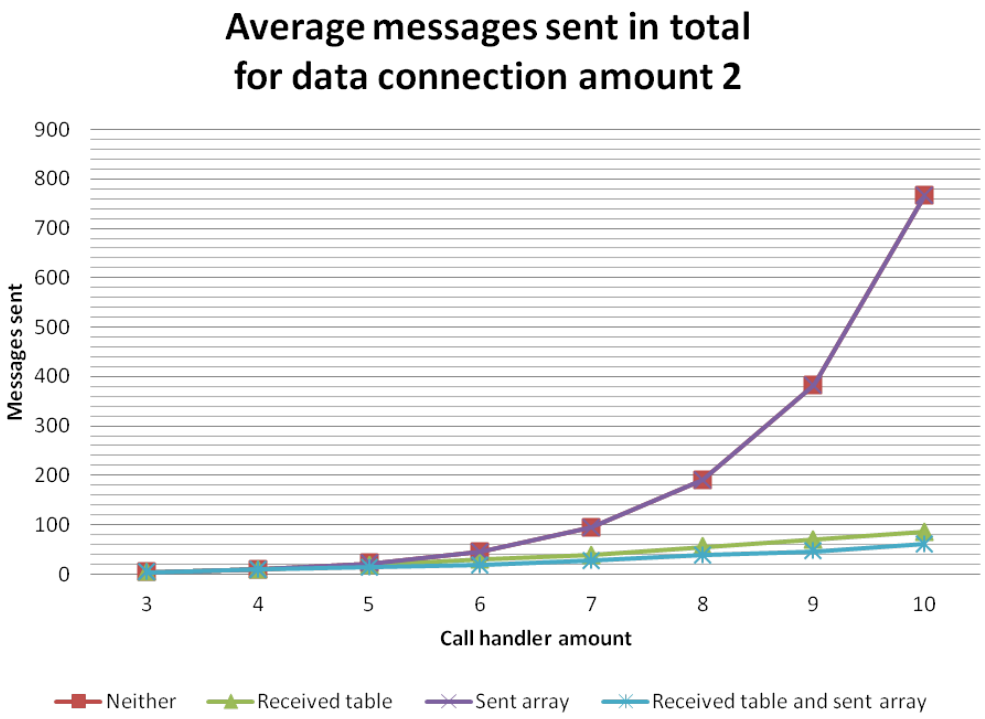

Figure 7.5: Total amount of messages sent from the first message until the stop of message forwarding. The data connection amount is constant and set to 2 and call handler amount varies from 3 to 10.

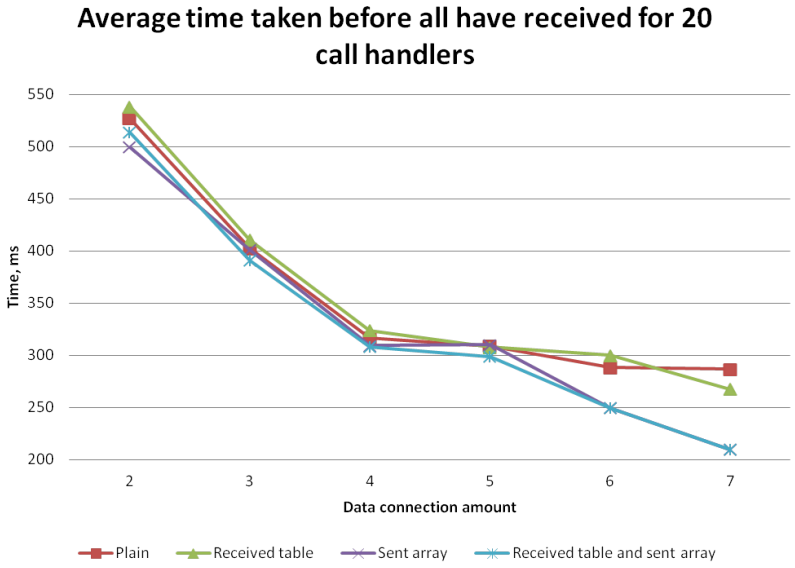

Figure 7.6: Total time in milliseconds from the first message until each call handler has received the data message at least once when for $N = 20$, $2 \leq b \leq 7$.

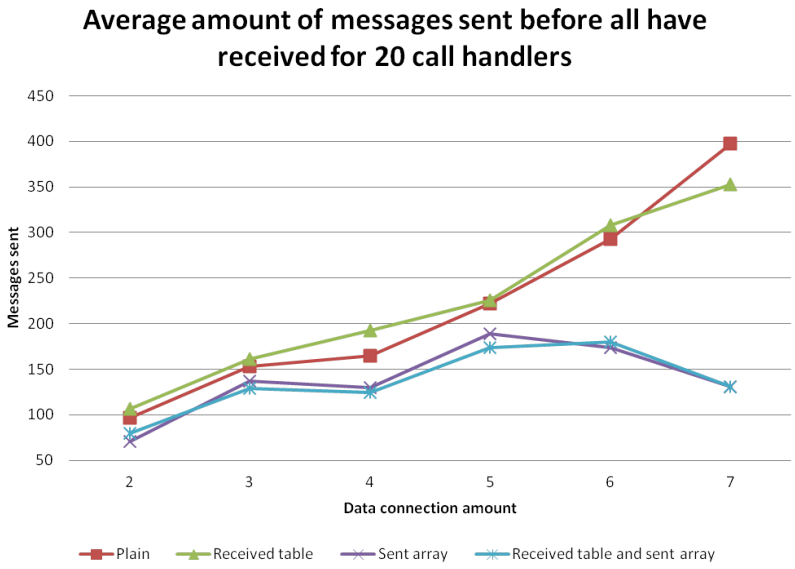

Figure 7.7: Amount of sent messages from the first message until each call handler has received the data message at least once when for $N = 20$, $2 \leq b \leq 7$.

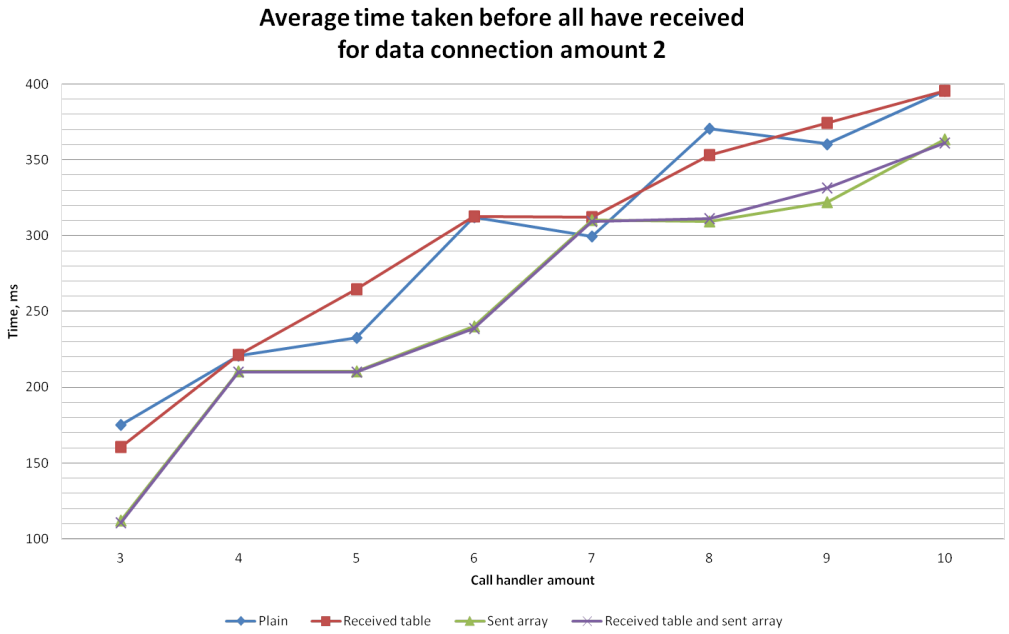

Figure 7.8: Total time in milliseconds from the first message until each call handler has received the data message at least once when $b = 2$, $3 \leq N \leq 10$.

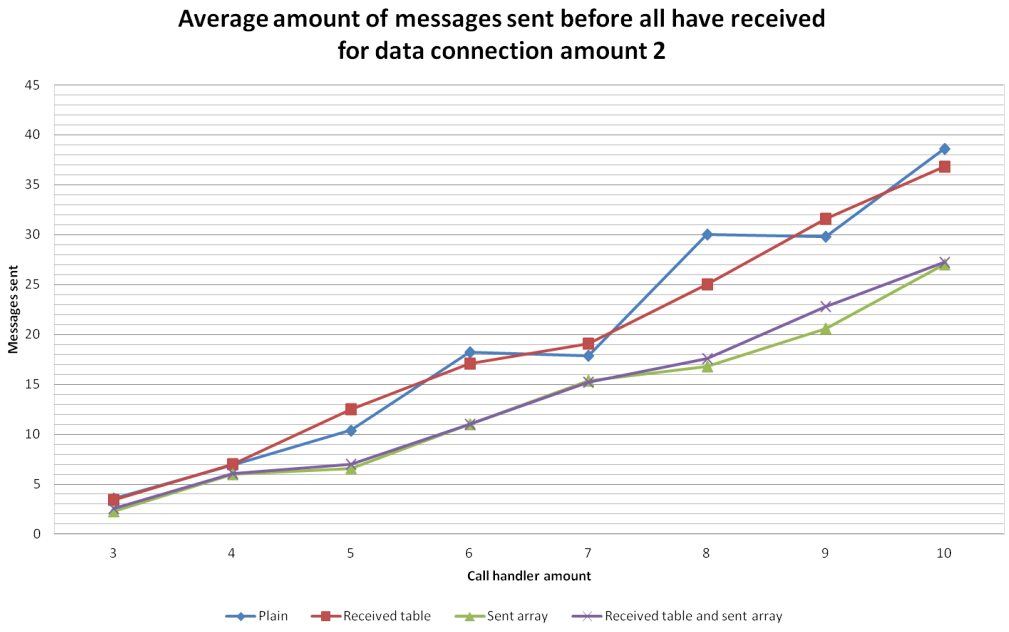

Figure 7.9: Amount of sent messages from the first message until each call handler has received the data message at least once $b = 2$, $3 \leq N \leq 10$.

Chapter 8

Black-box testing

While tests about how different protocols work give directional results, another important subject to test is how they work together. Also, takeover protocol was not tested in the previous chapter, because it is difficult to test separately from the other two protocols. That is why black-box testing is done for getting information about how the whole system functions. The point of these tests is to examine how good reliability can be achieved with different test parameters when the expected reliability of one single call handler is known.

8.1 Test setup

The test environment is the same as in the failure detection tests presented in Section 7.1.1. The only major difference is that calls are made.

Because of the limited resources in the virtualization platform, the maximum call handler amount cannot be large. Initial tests showed that reliable results could only be achieved up to 16 call handlers, after which memory started to run out. In the initial tests, whole the virtualization platform crashed because of running out of both memory and swap space already at call handler amount 25. That is why the tested call handler amounts are between [3, 20]. However, results are reliable only up to call handler amount 16, because after that, memory problems start to occur.

For the failure probability, some smallish values are selected. Otherwise there are not enough crash cases and it cannot be seen how call handlers behave in such cases.

There are two sets of test parameters. The first set is for examining the reliability for as many call handlers as it is possible to test with the test environment. That is why tested call handler amounts are all values between

[3, 10] and all even values between [12, 20]. Selected failure probabilities for the first set are 450, 900, 1350, 1800 and 2250. They mean, respectively, around 0.22 %, 0.11 %, 0.07 %, 0.06 % and 0.04 % probability of crashing every second. While the probabilities are small, for a real world system they would be bad. For example, with 0.22 % probability for a call handler crashing every second, within 24 hours it would mean that the call handler crashes at least once with 96 % probability.

The second set of test parameters is needed for better examining how failure probability affects the reliability. That is why even larger failure probabilities are used: 50, 100, 150, 300, 600 and 750. The tests are run for call handler amounts from 3 to 10. Larger call handler amounts are not tested with the larger probabilities because of the lack of time.

8.2 Criteria

For evaluating test results, criteria needs to be defined. The most important factor is if calls are successful. However, since testing is done by automated tests, it would be difficult to test if the calls are successful from the user's point of view. User experience in calls is difficult to test. For that reason, more system based criteria is needed. One good way for checking success of a single call is to examine the event data record (EDR) which is created from each call. EDR shows whether a call was successful, if it was answered, and how long the call was from the system's point of view.

Calls in the test are simple calls: begin-answer-release. Thus exactly one EDR should be created for each call, and success of a call can be seen from EDRs. When the amount of calls is known, the amount of expected EDRs is known as well. In addition to simple EDR amount, also the content of EDRs is examined. In the example system, EDRs contain at least these: the ID of the call handler which created the EDR, call ID, caller and receiver numbers, start time, end time, duration and release cause. A lot other information is saved in EDRs as well, but they are not relevant for evaluating the results of this test. Now, if any state change of a call goes unnoticed by call handlers, it can be seen in EDRs. Failure to handle the begin of a call would lead to a missing EDR. Not handling answer of a call would create an EDR with unexpected release cause and duration. Finally, in the case of an unprocessed release, either an EDR would be missing or then duplicated EDRs occur, depending on the case.

Test result EDRs are classified into *correct*, *acceptable time*, *unacceptable time*, *duplicate*, *faulty* and *missing* EDRs. Correct and acceptable time EDRs are wanted EDR types, called as *accepted* EDRs, while the others mean that

something has gone wrong in processing the corresponding call. The share of correct and acceptable time EDRs of all EDRs defines how well call handlers perform. In the ideal case, the amount of accepted EDRs is 100 % of all EDRs, and there are no duplicates.

Summarizing the different cases of EDRs in these tests:

- **Correct EDR** All data fields are correct. Accepted EDR.
- **Acceptable time EDR** Duration is too long or too short, however, within accepted variance. This is usually result of call handlers taking over calls which have unanswered messages. Accepted EDR.
- **Unacceptable time EDR** Duration is too long or too short and outside accepted variance. This might be cause for example by a takeover case in which a response is not sent fast enough.
- **Duplicate EDR** If more than one EDR for a call is created, each extra EDR is counted as a duplicate. The term *unique duplicates* presents how many distinct calls there are for which duplicates occur.
- **Faulty EDR** Some data in the EDR is wrong, most usual being wrong release cause. Does not include EDRs in which duration is too long or too short.
- **Missing EDR** There should have been an EDR for a call, but it had not been created by any call handler.

Additional information is received from heavy logging done by all components of the test system. All logs proved to be useful especially when examining reasons for failed cases.

8.3 Test results

Results are examined from two point of views: the effect of call handler amount to reliability, and the effect of failure probability to reliability. Because the total amount of EDRs per a test case varies slightly, the values are presented as percentages instead of absolute values in all graphs.

Figure 8.1 presents the percentage of accepted EDRs, i.e. EDRs which are totally correct or which are otherwise correct but have a too long or too short duration which, however, is within accepted limits. The results are separated by different call handler amount and failure probability pairs. Slight variance in percentages can be seen, but clear trend is not visible up to

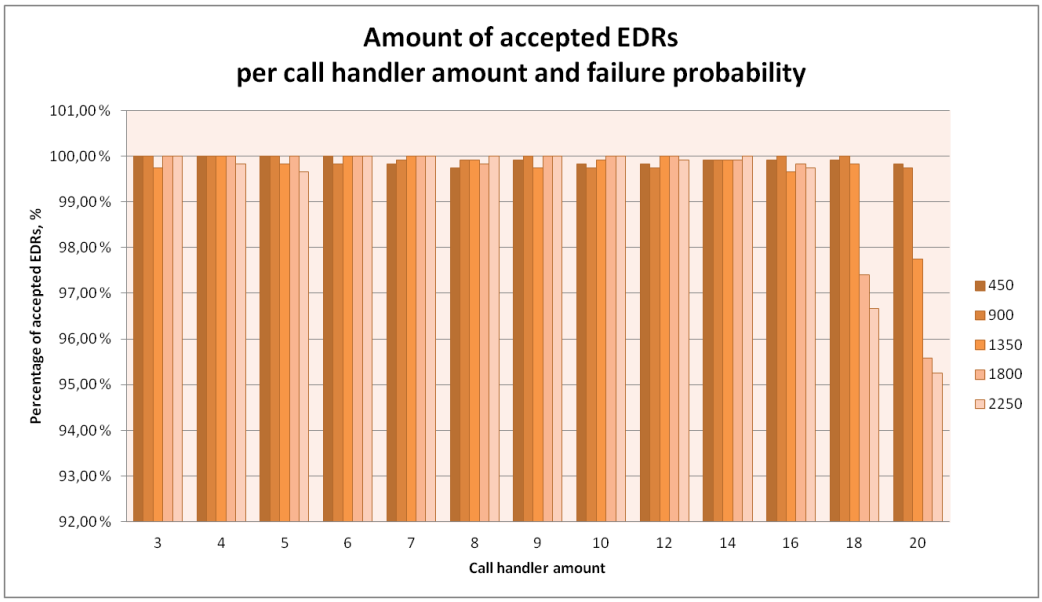

Figure 8.1: Percentages of accepted EDRs sorted by call handler amount between [3, 20] and failure probability.

call handler amount 16. On average, call handler amounts between 3 and 16 seem to work almost equally reliably. Deeper examination of test logs show that the variance is mostly explained by bugs and unexpected problems with the test environment.

Another presentation for the first test set is in Figure 8.2. It shows how the reliability changes depending on the call handler amounts. Results of all failure probabilities are added together. The larger the call handler amount is, the less fully correct EDRs there are, but the total reliability in the amount of accepted EDRs stays mostly unchanges, again up to the call handler amount of 16. The change of fully correct EDRs when the call handler amount grows is explained by the results from failure detection test, which will be further presented in Section 7.1. With larger call handler amount, failure detection time seems to grow, so it also takes slightly more time before takeover takes place. It causes slight changes in call durations in EDRs.

The effect of failure probability to the results is detectable in Figure 8.3. Because reliable results for call handler amounts larger than 16 could not be achieved with the test environment, the effect of failure probability is only examined by combining the results of call handler amounts from 3 to 16. With smaller failure probabilities, more takeover cases occur and cases in which call duration varies are more likely to occur. It can be seen in how the

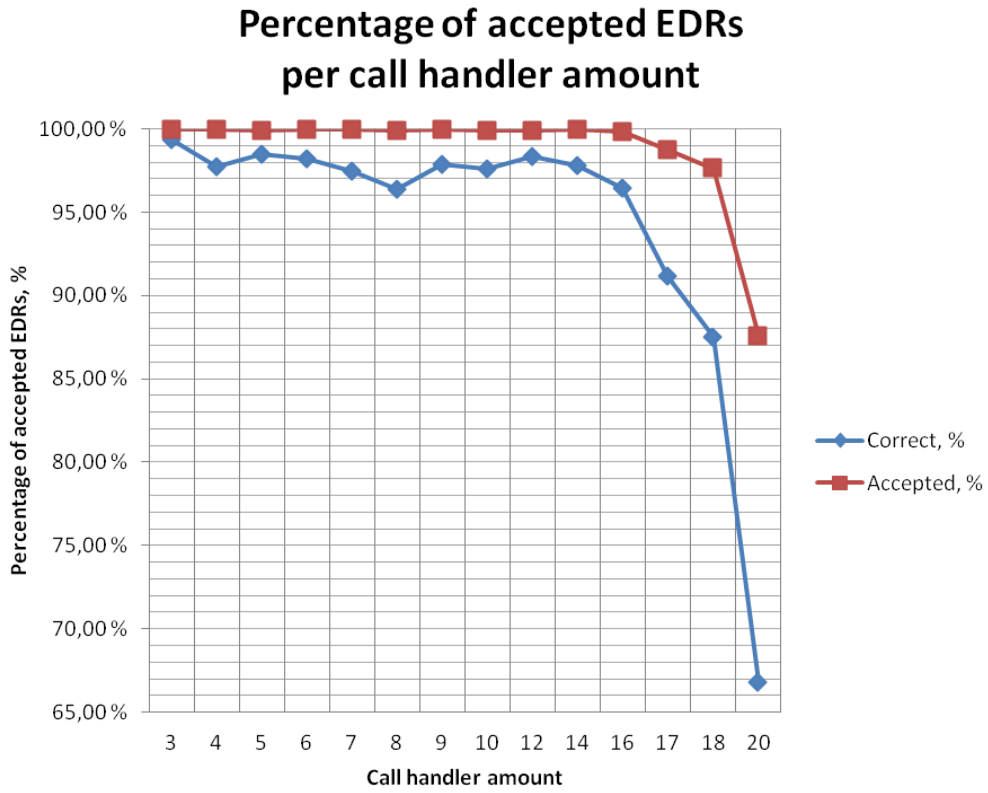

Figure 8.2: Effect of call handler amount to the percentage of correct and accepted EDRs when used failure probabilities are [450, 900, 1350, 1800, 2250].

amount of fully correct EDRs changes. Again, the total amount of accepted EDRs seems to stay the same.

Another interesting information in the test results is the amount of duplicates. Duplicates indicate about problems, because they mean that a call is not correctly removed from memory by at least one call handler. The total amounts of duplicates and the amount of unique duplicates per call handler amounts is shown in Figure 8.4.

Duplicate EDRs are created at least in two cases: when release message of a call did not spread across the system, or if for some reason release of a call is never properly processed. Deeper examination of test logs was done for some duplicate cases. Examining logs showed that some duplicates were results from a bug in call handler source codes. If a timeout for a call occurs and the owner call handler removes the call from memory, it does not erroneously

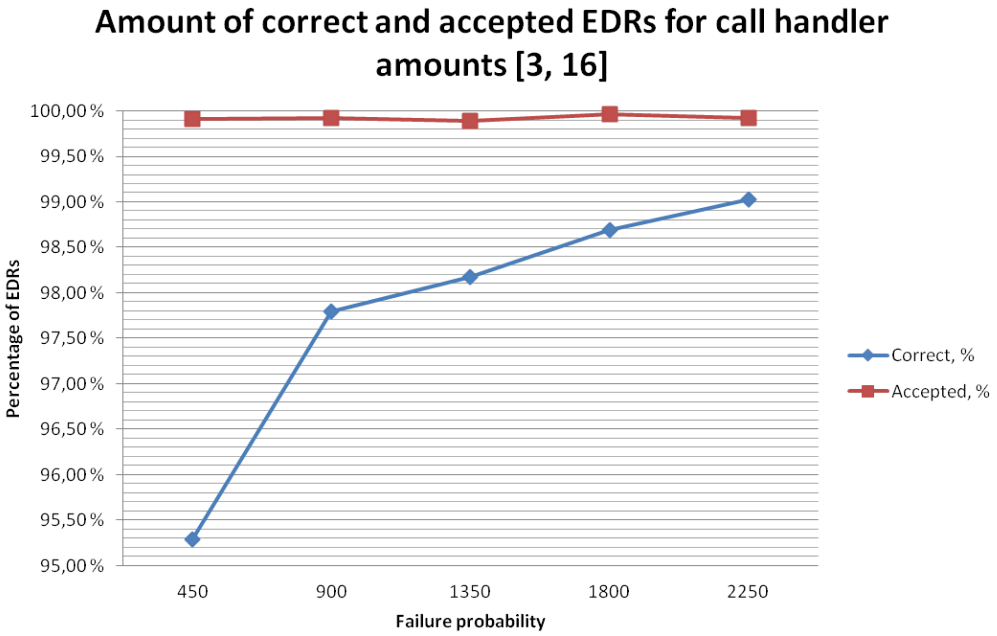

Figure 8.3: Effect of a failure probability to the percentage of correct and accepted EDRs when used call handler amounts are between [3, 16].

inform other call handlers about it. Also, in some cases, a malfunction of the test environment caused duplicates. For unknown reasons, the SIP client handler failed to release a call, which caused together with the timeout bug that the call was never released by call handlers, and when ever takeovers occurred, a duplicate was created after the call timeout.

However, some duplicates are created because of collaborative effect of both a weakness in the data dissemination protocol and a weakness in the failure detection algorithm after restart. The case can be best examined through an example. In the test case with 16 call handlers and the failure probability 450, at one point the call handler 13 crashes while it has ongoing calls. It, however, restarts only after 7 seconds from the crash. Thus others do not noticed it had crashed, and only send the update calls notification after noticing the restart. Thus 13 is able to continue processing the calls that it owned at the time of the crash. It receives release indicators for the calls, and handles the releases correctly. However, having been up only for less than t_{fail} time, it does not know which other call handlers are active and which not. So, when forwarding the information about release, it happens to send the messages only to not active call handlers. Thus the information about release never spread across the system, and duplicate EDRs of the call

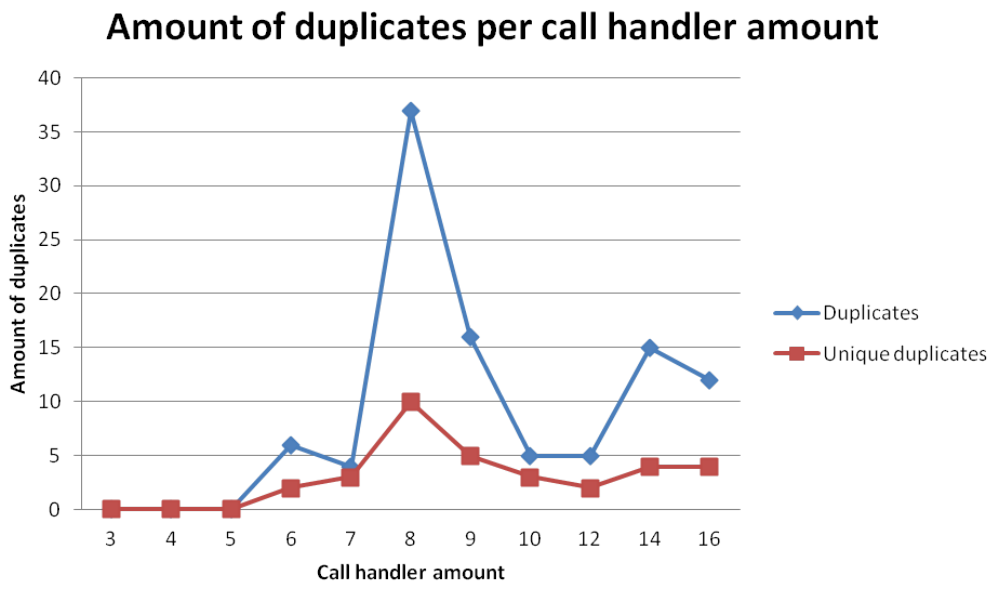

Figure 8.4: The amount of duplicates in total in tests with failure probabilities [450, 900, 1350, 1800, 2250] and call handler amount between [3, 16].

occur when later in the test 13 crashes again and other call handlers take over its calls.

A clear change into worse can be seen both in the Figure 8.1 and in the Figure 8.2 when moving from 16 call handlers to 17 or 18 call handlers. Interesting in results of 17, 18 and 20 is especially that the results are backwards to expected. The best results are achieved with $f = 450$, and results get worse when the failure probability diminishes. A very probable reason for the change was found with the study of test logs: memory issues of the virtualization environment.

The effect of memory running out can be seen starting from call handler amount of 17. Figure 8.5 shows comparison of the usage of memory and Linux swap of virtualization platform during one hour test with otherwise same setup than in normal test cases for call handlers 16 and 18. A clear difference in memory usage can be seen. While already in 16 call handlers the memory usage is all the time close to full, it still is enough, and swap is hardly needed at all. In the case of 18 call handlers, swap usage is constantly growing, though slowly, and the total memory usage is much closer to the full.

Also logs for call handlers endorse the probability of memory issues af-

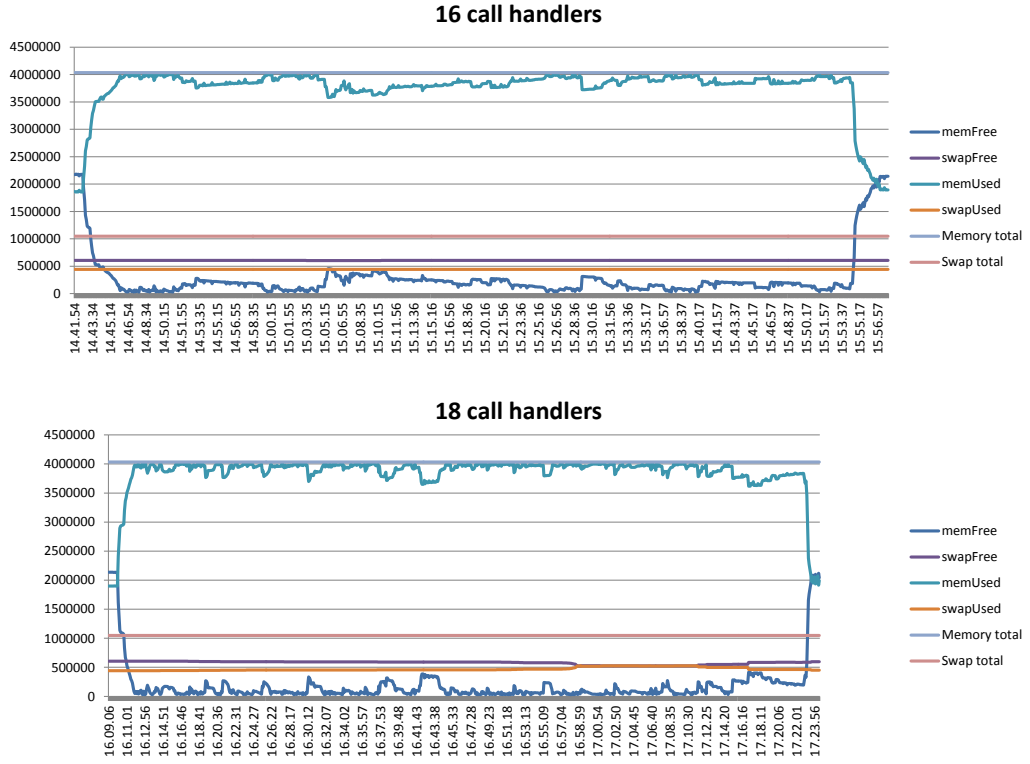

Figure 8.5: Memory usage of the host server when running a test with 16 and 18 call handlers.

fecting the reliability. Log records about out of memory errors for several call handlers could be found for test cases with 17 call handlers and failure probabilities 1800 and 2250; for 18 call handlers with failure probability values 1350, 1800 and 2250; and for 20 call handlers with failure probabilities 900, 1350, 1800 and 2250. Larger failure probabilities causes that, in average, there are less active call handlers every moment. Thus also the total memory usage stays lower than with smaller probabilities.

Because of the memory problems, the only results for larger call handler amounts which can be considered be at least somewhat reliable are the test cases with larger failure probabilities: 450, 900 and 1350 for 17 call handlers, 450 and 900 for 18 call handlers and only 450 for 20 call handlers.

Additional variation to the results is also created by some issues with the test environment. One such a case is that test environment assumes that a restarted call handler is available right away when the start command has been issued. For real, it takes some seconds before a restarted call handler is really up and running. In a few cases, missing or duplicated EDRs were

caused by a situation in which the test environment sent a message to a restarting call handler. The call handler had already loaded the networking modules, but not yet the cluster module which takes care of heartbeat and data message sending. If so happens, the call handler accepts the message causing the test environment erroneously thinks that the message was handled correctly. Call handler, however, cannot inform any other call handler about the message or, in some cases, even handle the message correctly on its own because of some of other important modules not having been loaded yet.

Another problem in the test environment is that the log parser expects all EDRs to present an answered call with the expected information. That is why all EDRs which are somehow distinct from the expected normal case are counted as failed calls. That is the case even if the EDR is actually correct, because a call was unsuccessful due to a problem in test environment. For example, if the SIP client handler never answers a call, an EDR telling that the call was not answered is created. The EDR is then regarded as a failure by the log parser, even if the EDR is correct for the call case.

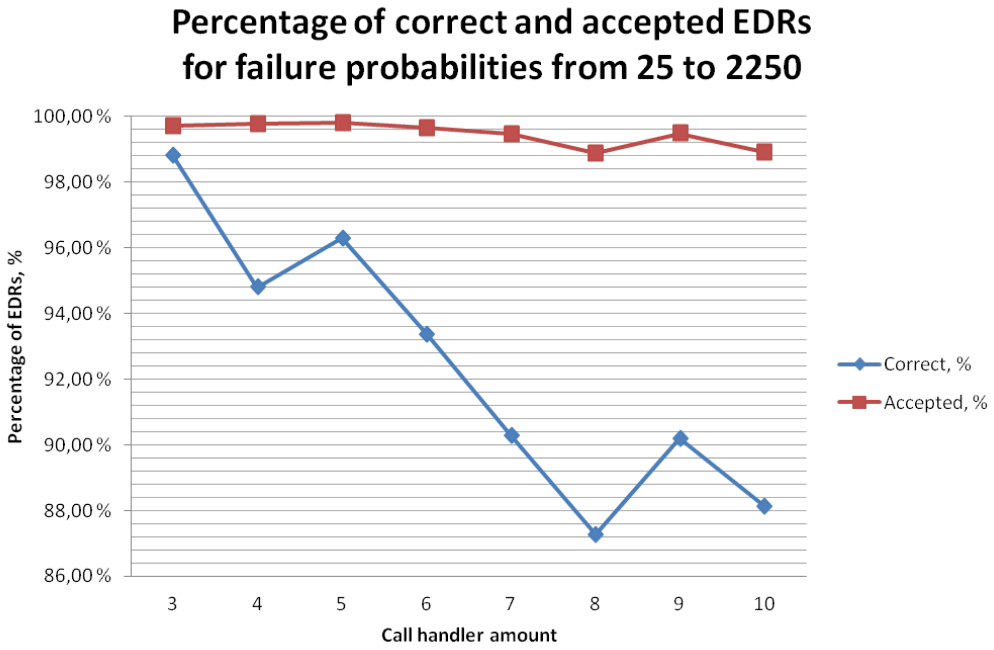

Figure 8.6: Amount of correct and accepted EDRs related to the number of call handlers

The second set of tests are for examining further if there is a trend for

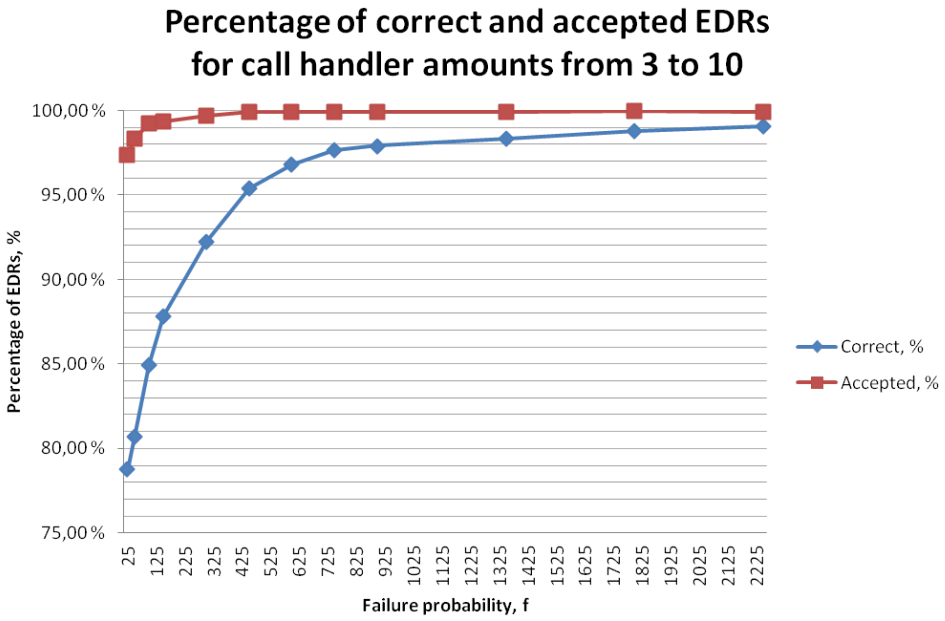

Figure 8.7: Amount of correct and accepted EDRs related to the failure probability for call handler amounts [3, 10]

changes in results if larger failure probabilities are included. Results of the second set tests are plotted in Figure 8.6 and in Figure 8.7. Including larger failure probabilities provides additional information about the functionality of the system. It can be seen clearly that both larger failure probability (i.e. smaller failure probability value f) and increasing the amount of call handlers decrease the amount of both correct and accepted EDRs related to the total amount of EDRs. However, even with the failure probability value 50, which means 2 % probability to crash every second, as much as 98 % of EDRs are still accepted.

Chapter 9

Conclusion

9.1 Conclusion

The starting point for this work was an existing telecommunication service platform. At the very beginning of this paper, three questions were asked:

- How failures in members of a network can be noticed, and which component should be responsible?
- Which structures and protocols provide the expected reliability for data spreading?
- How a system should behave in the case of failures?

Possible solutions for them were searched, and as a result, three protocols were applied to the example system. Separate protocols were needed because of different requirements for failure detection and data dissemination by the system. Takeover protocol was built on top of them both.

Failure detection and data dissemination protocols were tested separately. Additionally, black-box testing was done for seeing how all three protocols work together in a real-world-like environment. Unfortunately, because of limited resources of the used testing environment, reliable tests for larger call handler amounts could not be done.

A response to the first question is the implemented failure detection protocol. It is based on a basic gossip-style heartbeat protocol as presented in [62], however, spreading a heartbeat message resembles more of basic flat gossip presented in [31]. A difference to the original protocol is that heartbeat messages are made slightly lighter by only forwarding part of the status table instead of whole the table. Also, broadcasts do not occur, because each call handler is always at a known location and their addresses do not change.

Results of the failure detection tests show promising results for both accuracy and completeness. For smallest call handler amounts, no faulty detection nor missed detections occur at all. However, the larger the amount of call handlers grows, the larger the amounts of faulty detections and failures to detect a crash or restart grow as well. Test results for the failure detection protocol also suggest that detection time increases slowly as the amount of call handler increases. Tests could be done only up to call handler amount of 20, for which detection times are still adequate. The tests did not provide information about up to which call handler amount the heartbeat protocol has potential to be working well enough.

Worth noting is that two major test parameters, failure detection timeout and heartbeat connection amount, were constant for each call handler amount in the tests. They had been calculated from estimated probabilities of a similar semi-synchronized system. Thus the failure detection timeout and heartbeat connection amount might not have been ideal, and adjusting them could possibly provide even better results.

The second question is solved with a data dissemination protocol. It is initially based on a gossip protocol named Epidemic asynchronous rumour spreading, EARS, presented in [28], which is designed for asynchronous systems. Because of the expectations of the example system, the protocol was altered much. A major change is that rumours are sent to more than only one member. Also, instead of noting to which processes a message is sent, it is recorded that which processes have actually received it. Because of sending a message to b_{data} others, the amount of messages grows largely, and the system becomes quite spamming. For lowering the amount of messages and for speeding up the spread of a rumour, additional arrays and tables are maintained both in a rumour but also in memory for providing more information about receivers. The first array transferred in a rumour is received array, which keeps track on which call handlers have received the message already. The second is sent array, which keeps track on to which call handlers the message has been sent, but which have not confirmed it yet. Received table is maintained by each call handler in its memory. It contains a list of confirmed receivers for each different data message.

Tests were done for examining the necessity of the sent array and the received table. Since the received table is a base component of the protocol, it was present in all test cases. Thus there were four test cases in total: using only the received array, using both received and sent arrays, using both received array and table, and using all arrays and the table.

Test results showed that received table has a great importance on total message amounts. The saved information is useful because most of call handlers will receive a same message more than once. When receiving a message

again, the already known receiver information can be added to the message. Drawback of received tables is that the table requires extra memory from call handlers, and designing the table needs to be done carefully for avoiding leaking memory. A rule has to be created for determining when messages should be removed from the received table. However, memory requirement of received table is not huge when call handler amount is small and the system does not handle huge amounts of messages.

By the test results, the sent array seems to speed up dissemination of a message slightly. It takes slightly less time and less messages in total before all call handlers have received a rumour. Also, only together with the received table, the sent array seems to lower also the total amount of sent messages. This effect might be a result of how a sent array directs the message flow in a system. Drawback of sent arrays is that it makes messages larger in size which might causes more traffic to network.

One great weakness in the implemented data dissemination protocol exists. For keeping the total amount of messages as low as possible, only a small value for data connection amount, b_{data}, is recommended. Thus the most dangerous part of the protocol is the step one, when the original message creator sends a message to b_{data} other call handlers. It always takes some time before failed call handlers are detected, and message omissions might happen. Thus there only needs to be b_{data} message omissions or unexpected crashes at the same time, and there is a probability that no other call handler ever receive a message. Thus the solution can truly only be called $b_{data} - 1$ fault tolerant at any time, which is not a high achievement. However, because of the probabilistic nature of gossip protocols, only single data messages should get lost, maximum.

For the example system, the response to the third question is that other call handlers should continue handling calls if one fails. This has been accomplished with a takeover protocol, which defines who takes over calls of a crashed call handler. It utilizes both failure detection and data dissemination protocol. That is also why there were not test specifically for the takeover protocol: it could not have been easily separated from the two other protocols. Thus takeover was tested as a part of black-box testing.

In black-box testing, a clear change in reliability was not present for call handler amounts from 3 to 16: about 99.9 % reliability was achieved for all test cases with failure probability of 1/450 and smaller. However, with tests for larger probabilities, from 1/50 to 1/450, correlations between failure probability and reliability, and between call handler amount and reliability could be found. The results indicate that when the amount of call handlers rises, the reliability decreases. Also, when increasing the failure probability, the reliability decreases. However, the decrease was only clearly noticeable

with fairly large crash probabilities.

After 16 call handlers a sharp drop in reliability can be seen in test results. One likely reason for the change is the test environment running out of memory. Log entries were found for test cases with 17, 18 and 20 call handlers telling about memory errors. Also tests for 25 call handlers were tried, but they ended up with the virtualization environment running out of both memory and swap space, which lead to a full crash of all call handlers and the virtualization environment itself.

One big problems with the tests was that there were still some bugs in the implementation and some problems with the test environment itself occurred. A subject of further consideration is also the appliance of the test results in the real-world. The test environment tried to take care that there was always at least one active call handler, which is not the case in the real world. Also, the failure probabilities were larger than what they should be for real. These two together cause that it is expected that the reliability of, for example, 10 call handlers is not as good as that of 3 call handlers. For 3 call handlers, the test environment takes care that at least 33.33 % of call handlers is up and running. The same value for 10 call handlers is 10 %.

All things considered, it can be concluded that the solution works well for a small amount of nodes, at least up to the call handler amount of 16. By the results, there seems to be potential for the solution to work adequately also with a slightly larger, undefined call handler amount. However, both the data dissemination algorithm and the failure detection algorithm had slowly growing trends for their properties, which leads to assume that they would not be suitable for large node amounts.

9.2 Subjects for further studies

Detecting and tolerating many different kind of failures were left out of the scope in this paper, even if they most probably happen at some point of time. Especially problematic and worth research would be those arbitrary failures which potentially could change contents in data messages. Additional research would be needed for finding out how to protect the data messages against arbitrary failures.

The test environment only allowed testing with a few call handlers. A test environment with better resources would be needed for finding the upper limit of call handlers until which the solution can be trusted to work well enough.

Another possible subject of further studies is the importance of network. It was assumed in this study that the connections between nodes are equiva-

lent, and the delay caused by the network is always about the same between any two call handlers. When considering the connections between nodes as a graph, they would be presented as an unweighted graph. In the real world, it is very likely that the connections between any two call handler are not equivalent. Especially if nodes are located in different parts of world, it could be reasonable to consider the physical distances and delays in networks. How would weighted connections change protocols and gossip target selections? Could, for example, adaptive timeout instead of constant t_{fail} provide a solution for it?

Bibliography

[1] 3rd Generation Partnership Project (3GPP). TS 23.228 IP Multimedia Subsystem (IMS); Stage 2, June 2015. Version 13.3.0, Available `http://www.3gpp.org/DynaReport/23228.htm`.

[2] Aguilera, M. K., Chen, W., and Toueg, S. Heartbeat: A timeout-free failure detector for quiescent reliable communication. Tech. rep., Ithaca, NY, USA, 1997.

[3] Arora, A., and Kulkarni, S. Detectors and correctors: a theory of fault-tolerance components. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on* (May 1998), pp. 436–443.

[4] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on 1*, 1 (Jan 2004), 11–33.

[5] Bazzi, R. A., and Neiger, G. Simplifying fault-tolerance: Providing the abstraction of crash failures. *J. ACM 48*, 3 (May 2001), 499–554.

[6] Bettner, P., and Terrano, M. 1500 archers on a 28.8: Network programming in age of empires and beyond. *Presented at GDC2001 2* (2001), 30p.

[7] Bhuyan, L., and Agrawal, D. Generalized hypercube and hyperbus structures for a computer network. *Computers, IEEE Transactions on C-33*, 4 (April 1984), 323–333.

[8] Birman, K., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. Bimodal multicast. Tech. rep., Ithaca, NY, USA, 1998.

[9] Burns, M., George, A., and Wallace, B. Simulative performance analysis of gossip failure detection for scalable distributed systems. *Cluster Computing 2*, 3 (1999), 207–217.

[10] Chandra, T. D., and Toueg, S. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (Mar. 1996), 225–267.

[11] Chen, J., Kanj, I., and Wang, G. Hypercube network fault tolerance: a probabilistic approach. In *Parallel Processing, 2002. Proceedings. International Conference on* (2002), pp. 65–72.

[12] Chongnan, W., Zongtao, W., and Hongwei, X. Design of message-oriented middleware with publish/subscribe model on telemetry and command computer. In *Systems and Informatics (ICSAI), 2014 2nd International Conference on* (Nov 2014), pp. 454–458.

[13] Cristian, F. Understanding fault-tolerant distributed systems. *Commun. ACM 34*, 2 (Feb. 1991), 56–78.

[14] D'Angelo, G., Ferretti, S., and Marzolla, M. Adaptive event dissemination for peer-to-peer multiplayer online games. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques* (ICST, Brussels, Belgium, Belgium, 2011), SIMUTools '11, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 312–319.

[15] Datta, A., Girdzijauskas, S., and Aberer, K. On de bruijn routing in distributed hash tables: there and back again. In *Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on* (Aug 2004), pp. 159–166.

[16] de Bruijn, N. G. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen 49* (1946), 758–764.

[17] Defago, X., Schiper, A., and Sergent, N. Semi-passive replication. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on* (Oct 1998), pp. 43–50.

[18] Demers, A., Greene, D., Houser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. Epidemic algorithms for replicated database maintenance. *SIGOPS Oper. Syst. Rev. 22*, 1 (Jan. 1988), 8–32.

[19] Doerr, B., and Fouz, M. Asymptotically optimal randomized rumor spreading. In *Automata, Languages and Programming*, L. Aceto, M. Henzinger, and J. Sgall, Eds., vol. 6756 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 502–513.

[20] Elhadef, M., and Boukerche, A. A gossip-style crash faults detection protocol for wireless ad-hoc and mesh networks. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa* (April 2007), pp. 600–605.

[21] Esfahanian, A.-H., and Hakimi, S. Fault-tolerant routing in debruijn comrnunication networks. *Computers, IEEE Transactions on C-34*, 9 (Sept 1985), 777–788.

[22] Fayyaz, M., Vladimirova, T., and Caujolle, J.-M. Adaptive middleware design for satellite fault-tolerant distributed computing. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on* (June 2012), pp. 23–30.

[23] Fedoruk, A., and Deters, R. Improving fault-tolerance by replicating agents. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2* (New York, NY, USA, 2002), AAMAS '02, ACM, pp. 737–744.

[24] Ferretti, S., and D'Angelo, G. Multiplayer online games over scale-free networks: A viable solution? In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques* (ICST, Brussels, Belgium, Belgium, 2010), SIMUTools '10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 5:1–5:8.

[25] Florio, V. D., and Blondia, C. A survey of linguistic structures for application-level fault tolerance. *ACM Comput. Surv. 40*, 2 (May 2008), 6:1–6:37.

[26] GauthierDickey, C., Lo, V., and Zappala, D. Using n-trees for scalable event ordering in peer-to-peer games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video* (New York, NY, USA, 2005), NOSSDAV '05, ACM, pp. 87–92.

[27] GauthierDickey, C., Zappala, D., Lo, V., and Marr, J. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proceedings of the 14th International Workshop on Network and Operating*

Systems Support for Digital Audio and Video (New York, NY, USA, 2004), NOSSDAV '04, ACM, pp. 134–139.

[28] Georgiou, C., Gilbert, S., Guerraoui, R., and Kowalski, D. R. Asynchronous gossip. *J. ACM 60*, 2 (May 2013), 11:1–11:42.

[29] Guerraoui, R., and Schiper, A. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies — Ada-Europe '96*, A. Strohmeier, Ed., vol. 1088 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996, pp. 38–57.

[30] Gupta, I., Kermarrec, A.-M., and Ganesh, A. J. Efficient epidemic-style protocols for reliable and scalable multicast. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2002), SRDS '02, IEEE Computer Society, pp. 180–.

[31] Gupta, I., Kermarrec, A.-M., and Ganesh, A. J. Efficient and adaptive epidemic-style protocols for reliable and scalable multicast. *IEEE Trans. Parallel Distrib. Syst. 17*, 7 (July 2006), 593–605.

[32] Hosseinabady, M., Kakoee, M., Mathew, J., and Pradhan, D. Reliable network-on-chip based on generalized de bruijn graph. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International* (Nov 2007), pp. 3–10.

[33] Hosseinabady, M., Kakoee, M., Mathew, J., and Pradhan, D. Low latency and energy efficient scalable architecture for massive nocs using generalized de bruijn graph. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 19*, 8 (Aug 2011), 1469–1480.

[34] Imran, A., Ul Gias, A., Rahman, R., Seal, A., Rahman, T., Ishraque, F., and Sakib, K. Cloud-niagara: A high availability and low overhead fault tolerance middleware for the cloud. In *Computer and Information Technology (ICCIT), 2013 16th International Conference on* (March 2014), pp. 271–276.

[35] International Electrotechnical Commission, IEC. Electropedia, 2015. `http://www.electropedia.org/`. Accessed 03.03.2015.

[36] International Electrotechnical Commission, IEC. IEC, TC 56 standards, 2015. `http://tc56.iec.ch/about/faq.htm`. Accessed 03.03.2015.

[37] Jahanian, F., and Mok, A. Safety analysis of timing properties in real-time systems. *Software Engineering, IEEE Transactions on SE-12*, 9 (Sept 1986), 890–904.

[38] JBoss A-MQ Docs Team. Red Hat JBoss A-MQ 6.1 Fault Tolerant Messaging, 2014. `https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_A-MQ/6.1/pdf/Fault_Tolerant_Messaging/Red_Hat_JBoss_A-MQ-6.1-Fault_Tolerant_Messaging-en-US.pdf`.

[39] Karp, R., Schindelhauer, C., Shenker, S., and Vocking, B. Randomized rumor spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 2000), FOCS '00, IEEE Computer Society, pp. 565–.

[40] Ku, H.-K., and Hayes, J. P. Optimally edge fault-tolerant trees. *Networks 27*, 3 (1996), 203–214.

[41] Laprie, J.-C. Dependable computing and fault tolerance : Concepts and terminology. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on* (Jun 1995), pp. 2–.

[42] Lavinia, A., Dobre, C., Pop, F., and Cristea, V. A failure detection system for large scale distributed systems. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on* (Feb 2010), pp. 482–489.

[43] Loguinov, D., Casas, J., and Wang, X. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. *Networking, IEEE/ACM Transactions on 13*, 5 (Oct 2005), 1107–1120.

[44] Marian, T., Birman, K., and van Renesse, R. A scalable services architecture. In *Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on* (Oct 2006), pp. 289–300.

[45] Nayak, A., Jone, W.-B., and Das, S. Designing general-purpose fault-tolerant distributed systems-a layered approach. In *Parallel and Distributed Systems, 1994. International Conference on* (Dec 1994), pp. 360–364.

[46] Nguyen, N. C., Dinh, T. V., and Anh, T. D. Improving shortest path routing in hyper-de bruijn networks. In *Strategic Technology, 2007. IFOST 2007. International Forum on* (Oct 2007), pp. 288–292.

[47] Object Management Group, Inc. Common Object Request Broker Architecture: Core Specification, December 2002. Version 3.0.2 - Editorial update.

[48] Oracle. JavaSE version 7 javadocs, 2015. `http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html`. Accessed 14.09.2015.

[49] Pietzuch, P., and Bacon, J. Hermes: a distributed event-based middleware architecture. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on* (2002), pp. 611–618.

[50] Pradhan, D. Dynamically restructurable fault-tolerant processor network architectures. *Computers, IEEE Transactions on C-34*, 5 (May 1985), 434–447.

[51] Pradhan, D. Fault-tolerant multiprocessor link and bus network architectures. *Computers, IEEE Transactions on C-34*, 1 (Jan 1985), 33–45.

[52] Raghavendra, C., AVIvizienis, A., and Ercegovac, M. Fault tolerance in binary tree architectures. *Computers, IEEE Transactions on C-33*, 6 (June 1984), 568–572.

[53] Ranganathan, S., George, A., Todd, R., and Chidester, M. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing 4*, 3 (2001), 197–209.

[54] Red Hat, Inc. Apache ActiveMQ, 2015. `http://www.jboss.org/products/amq/overview/`. Accessed 03.03.2015.

[55] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. Sip: Session initiation protocol, 2002.

[56] Seeger, C., Kemme, B., Kabus, P., and Buchmann, A. Area-based gossip multicast. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games* (New York, NY, USA, 2008), NetGames '08, ACM, pp. 40–45.

[57] Sengupta, A., Sen, A., and Bandyopadhyay, S. On an optimally fault-tolerant multiprocessor network architecture. *Computers, IEEE Transactions on C-36*, 5 (May 1987), 619–623.

[58] Sengupta, A., Sen, A., and Bandyopadhyay, S. Fault-tolerant distributed system design. *Circuits and Systems, IEEE Transactions on 35*, 2 (Feb 1988), 168–172.

[59] Sistla, K., George, A., Todd, R., and Tilak, R. Performance analysis of flat and layered gossip services for failure detection and consensus in scalable heterogeneous clusters. In *Parallel and Distributed Processing Symposium., Proceedings 15th International* (April 2001), pp. 802–809.

[60] Sridhar, N. Decentralized local failure detection in dynamic distributed systems. In *Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on* (Oct 2006), pp. 143–154.

[61] The Apache Software Foundation. Apache ActiveMQ, 2015. `http://activemq.apache.org/`. Accessed 03.03.2015.

[62] van Renesse, R., Minsky, Y., and Hayden, M. A gossip-style failure detection service. In *Middleware'98*, N. Davies, S. Jochen, and K. Raymond, Eds. Springer London, 1998, pp. 55–70.

[63] Wang, J., Chen, J.-W., Deng, Y., and Zheng, D. Research of the middleware based fault tolerance for the complex distributed simulation applications. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on* (Dec 2009), pp. 1–4.

[64] Wang, J., Wei, Y., and Jia, X. The design and implementation of emp: A message-oriented middleware for mobile cloud computing. In *Global High Tech Congress on Electronics (GHTCE), 2013 IEEE* (Nov 2013), pp. 78–81.

[65] Wang, L., and Zhou, W. An architecture for building reliable distributed object-based systems. In *Technology of Object-Oriented Languages, 1997. TOOLS 24. Proceedings* (Sep 1997), pp. 260–265.

[66] Wickramarachchi, C., Perera, S., Jayasinghe, S., and Weerawarana, S. Andes: A highly scalable persistent messaging system. In *Web Services (ICWS), 2012 IEEE 19th International Conference on* (June 2012), pp. 504–511.

[67] Wuhib, F., Dam, M., and Stadler, R. A gossiping protocol for detecting global threshold crossings. *Network and Service Management, IEEE Transactions on 7*, 1 (March 2010), 42–57.

[68] Yeh, C. The robust middleware approach for transparent and systematic fault tolerance in parallel and distributed systems. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on* (Oct 2003), pp. 61–68.

[69] Zhao, W., Melliar-Smith, P., and Moser, L. Fault tolerance middleware for cloud computing. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* (July 2010), pp. 67–74.

Appendix A

Probability of receiving a heartbeat within r rounds

This probability calculation is used as a directions when estimating the required round time limits for crash detection. When the actual example system is fully asynchronous and nothing can be assumed about the environment where it is running, calculating the actual probabilities is difficult. In these calculations, the system is assumed to be synchronous in such a way that each call handler executes the local round at the very same time. Thus the time when local rounds are executed at the very same time can be called *global rounds*. It is also assumed that every message sent on round $r = i$ are received on round $r = i + 1$, no message losses or delays occur. There are n_{total} call handlers in total in the system, and on every global round, the rumour is sent to b fully randomly selected call handlers. The selection of b addressees is done from $n_{target} = n_{total} - 1$ call handlers, because a call handler never sends a message to itself.

The point in interest is to calculate the worst case probability of call handler n_i to receive any new heartbeat value from n_j within r global rounds. If a call handler n_k receives a heartbeat value on round r_i, the heartbeat is forwarded to b others on round $r_i + 1$. If more than one heartbeat value is received, only the newest of them is forwarded. n_k always send the rumours to b distinct call handlers.

The worst case on every round is when as few call handlers as possible know any heartbeat value to forward. The worst case from the point of n_i is that there are as few call handlers as possible, which know any new heartbeat value. This happens if there are $b + 1$ call handlers which are only sending messages to each other, thus keeping the total amount of call handlers which know any new value constant $b + 1$.

On the first global round, there are not any call handlers which know a

new heartbeat. n_j creates a heartbeat h_1 and forwards it to b distinct call handlers. During the first round, the probability of n_i receiving the heartbeat value h_1 is

$$\begin{aligned} P(R1) &= P(one\ selection) = P(OS) \\ &= \frac{\binom{1}{1}\binom{n_{target}-1}{b-1}}{\binom{n_{target}}{b}} \end{aligned}$$

At the beginning of round 2, there are b call handlers which know the heartbeat count h_1, and n_j creates a new heartbeat rumour with the value h_2. Each of them forwards the heartbeat rumour known by them to b call handlers. Thus the probability of n_i to receive either the heartbeat value h_1 or h_2 during the round 2 is:

$$\begin{aligned} P(R2) &= 1 - P(R2^c) \\ &= 1 - (\prod_{i=1}^{b+1} P(OSi^c)) \\ &= 1 - (P(OS^c)^{b+1}) \\ &= 1 - ((1 - P(OS))^{b+1}) \end{aligned}$$

As the worst case is examined, the start for the round 3 is the same as was the state for the round 2. There are only b call handlers which now know the heartbeat value h_2, and additionally h_j creates again a heartbeat message with the value h_3. Each call handler then sends their most recent known value to b others. Thus the probability of n_i to receive any heartbeat value about n_j during the round 3 is exactly the same as on the round 2. In the worst case, the same probability applies also for any round r. Thus the probability $P(R)$ of n_i receiving any new heartbeat value during one round r for rounds $r > 1$ is the same as $P(R2)$, $P(R) = P(R2)$.

Now, the total worst case probability of n_i to receive any new heartbeat at least once within r rounds can be found.

Probability of receiving during the round 1:

$$P(r_1) = P(OS)$$

Probability of receiving during the round 1 or round 2:

$$\begin{aligned} P(r_2) &= 1 - P(r_2^c) \\ &= 1 - (P(OS^c)P(R^c)) \\ &= 1 - ((1 - P(OS))(1 - P(R))) \end{aligned}$$

Probability of receiving during the round 1, 2 or 3:

$$
\begin{aligned}
P(r_3) &= 1 - P(r_3^c) \\
&= 1 - (P(r_2^c)P(R^c)) \\
&= 1 - (P(OS^c)P(R^c)P(R^c)) \\
&= 1 - ((1 - P(OS))(1 - P(R))(1 - P(R))) \\
&= 1 - ((1 - P(OS))(1 - P(R))^2)
\end{aligned}
$$

$$\vdots$$

Probability of receiving during the round 1, 2, 3 ... r:

$$
\begin{aligned}
P(r) &= 1 - P(r^c) \\
&= 1 - (P(OS^c)P(R^c)^{r-1} \\
&= 1 - ((1 - P(OS))(1 - P(R))^{r-1})
\end{aligned}
$$

From the equation of $P(r)$ some estimated values for t_{fail} can be calculated. At first, some expected probability for $P(r)$ must be given. Then r is solved for different values of b. In Figures A.1 and A.2 some estimated reasonable values for t_{fail} for different call handler amounts and connection amounts are calculated, when the expected $P(r)$ is 0.9999999999999999. Vertical axis shows values for t_{fail}, while horizontal axis shows the used connection amount b, which is always $b < n_{total}$.

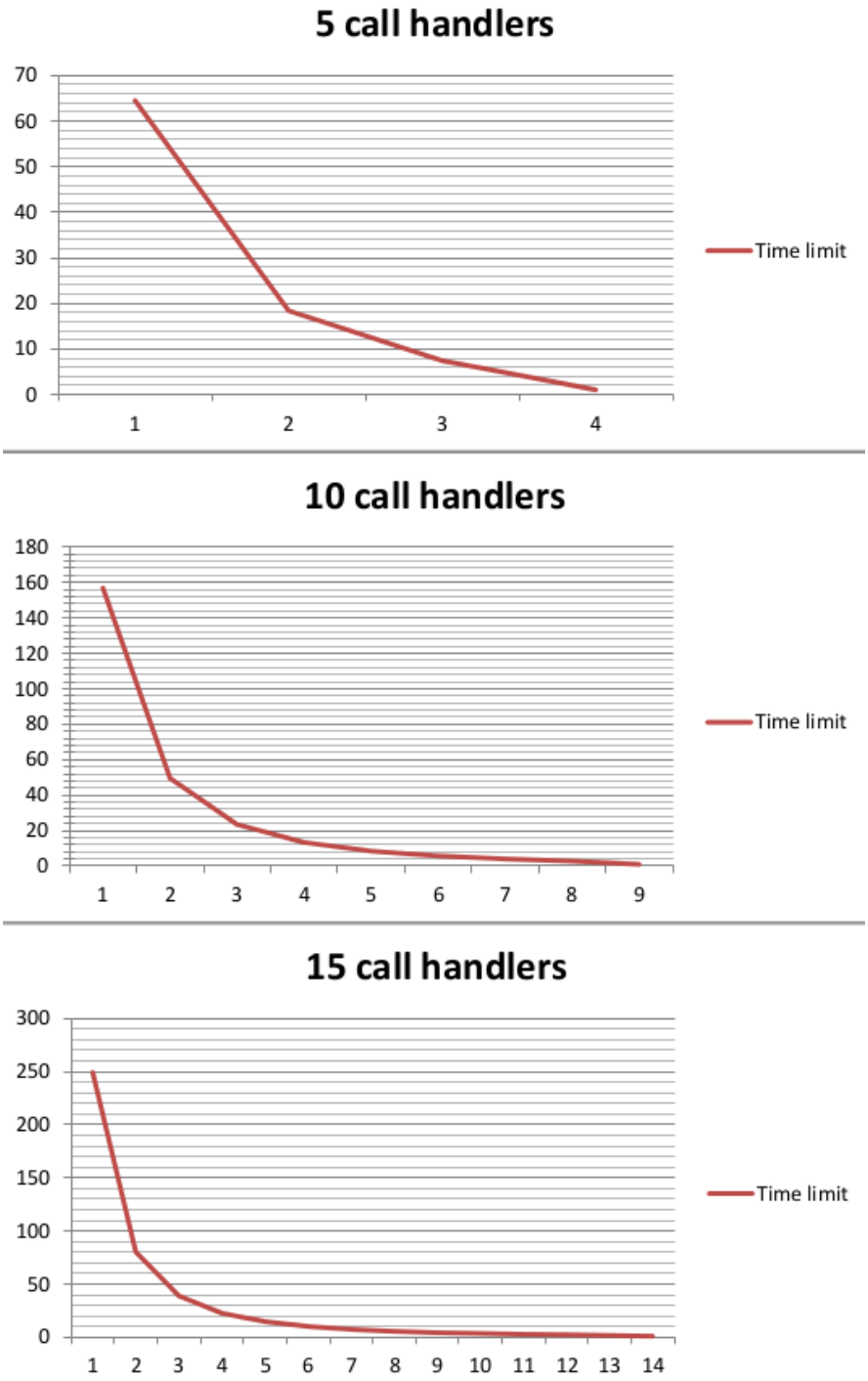

Figure A.1: Estimations for good t_{fail} values for call handler amounts 5, 10 and 15.

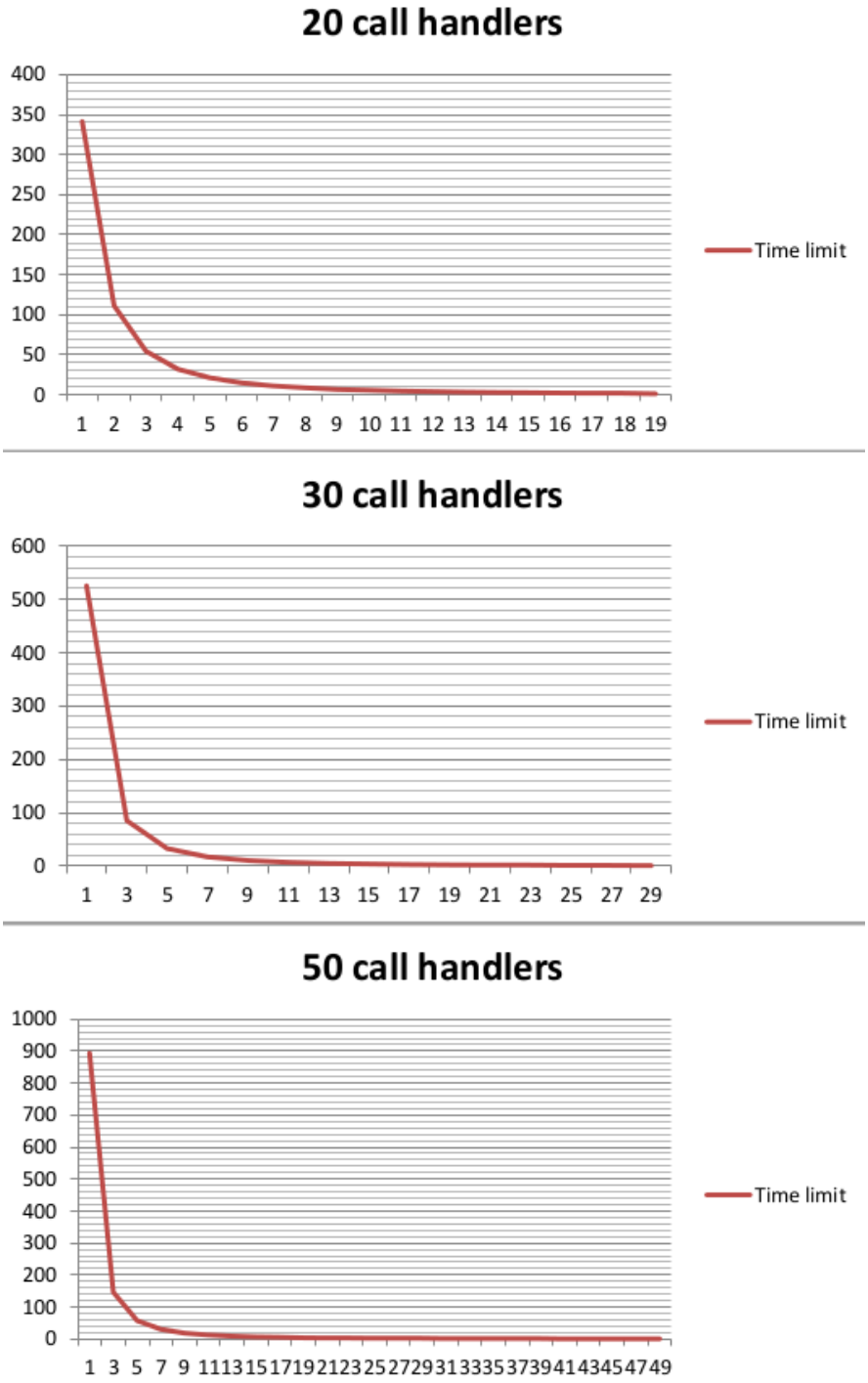

Figure A.2: Estimations for good t_{fail} values for call handler amounts 20, 30 and 50.

Appendix B

Presentations of messages

For transfer purposes, the message presentations are translated into byte presentations. This Appendix shortly describes byte presentations for the three kind of messages which are used for connection between call handlers: heartbeat messages, data messages and one-time messages.

A heartbeat message contain one or more heartbeat rumours, and they are forwarded using the heartbeat protocol presented in Section 5.1. One heartbeat rumour in a message consists of a pair of a call handler ID and HB. Content of a heartbeat message is shown in Figure B.1.

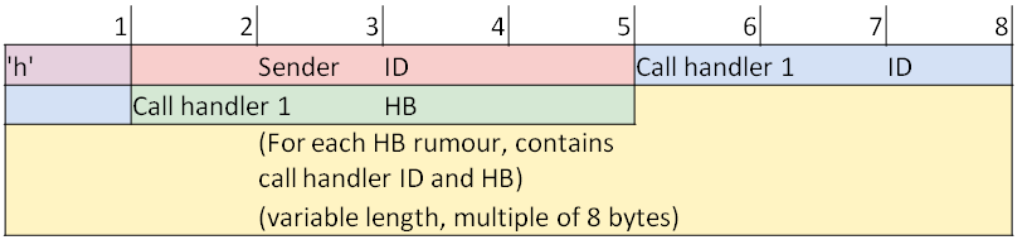

Figure B.1: Bytes in a heartbeat message.

There are three kind of data messages: call state change messages, takeover messages and general message forward requests, which are presented in Section 6.3. Forwarding of these messages is done using the data dissemination algorithm presented in Section 5.2. There is some common information needed by all these data messages: received array and sent array needed in dissemination, but also unique message reference ID, related call ID (except in takeover message) and call handler ID and HB of the original sender. All the information common for all data messages is presented in a *Data message envelope*, which wraps the actual message. The byte presentation of a data

message envelope is shown in Figure B.2.

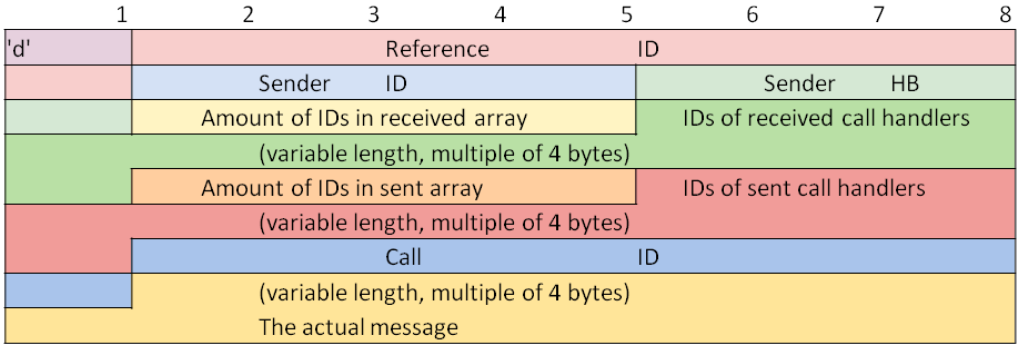

Figure B.2: Bytes in a data envelope message.

Takeover messages are slightly different from the other two data message types, because it is call handler specific while the other two are call specific. Thus it is the only message which does not need call ID information. However, the call ID header is still present in its envelopes. The message contains required information for takeover handling: ID and HB of the overtaker call handler, and ID and HB of the yielder call handler at the time of the detected crash. Content of a takeover message is shown in Figure B.3.

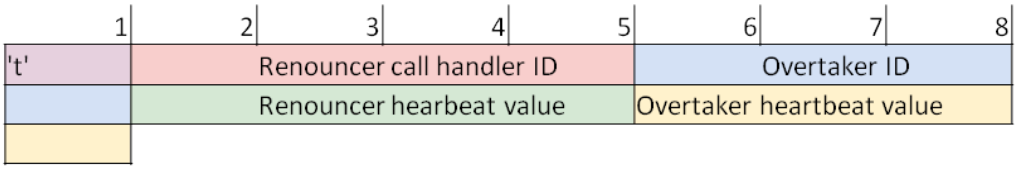

Figure B.3: Bytes in a takeover message.

Both general forward message request and call state change message are meant for updating call states. General messages, which is AGI, AMI or command message, are presented as a serialized version of the object used in the code, because there are many different kind of AGI, AMI and command messages which contain different information. That is why the exact form of general messages is not presented.

Call state change messages, then again, contain a new state for the given call at the given point of a session. A state presentation contains the same information than the state of a global call in memory. Content of a state

change message is shown in Figure B.4, and presentation of a call state which is included in the call state change message is presented in Figure B.5.

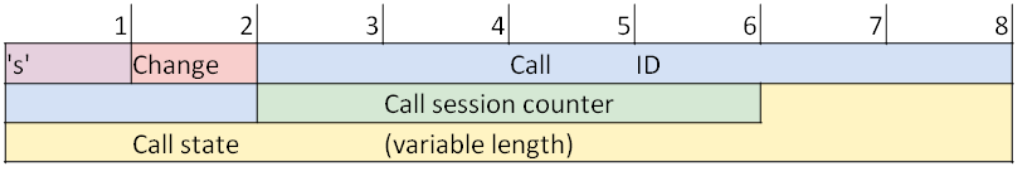

Figure B.4: Bytes in a state change message.

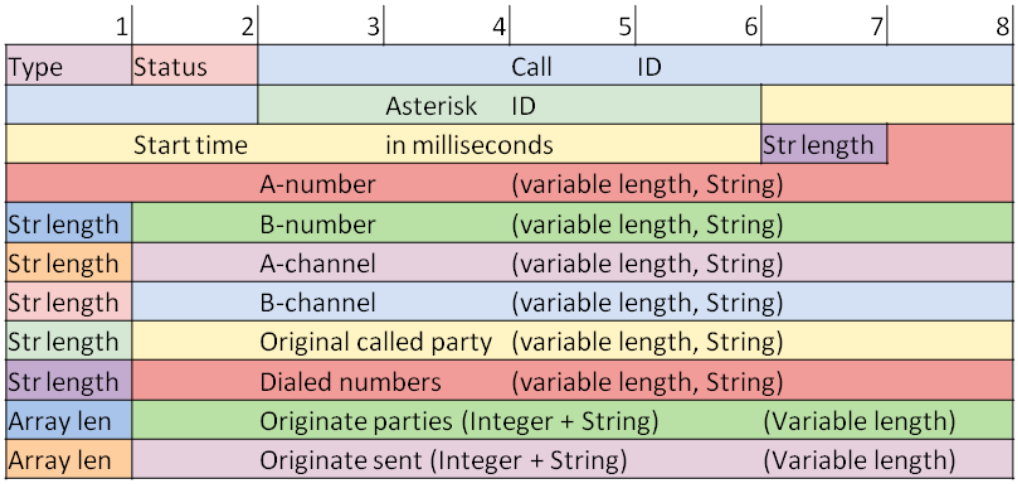

Figure B.5: Bytes in a state representation used in a state change message.

The third message type, notification messages, contains messages which are only sent once from one call handler to another, and they are not forwarded. The two types of notification messages are: unloading notification message and update calls notification. Contents of both of the messages are simple. An unloading notification only contains the ID of the call handler which is being unloaded. An update calls notification contains a list of byte presentations of calls, containing all the same information than what is contained in a call in memory. A major part of a call presentation is the call state presentation. Additionally, the currently known ID of the call handler which currently owns the call, the session counter and the start heartbeat count of the owner call handler at the time of either start or takeover of the call.