Computational Effort Reduction of Fault Tolerance in Scalable Agent Support Systems

By: Ashenafi Kassahun

Submitted to the School of Graduate Studies of Addis Ababa University in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Addis Ababa University, Faulty of Informatics

Department of Computer Science

September, 2008

Addis Ababa University
School of Graduate Studies
Faculty of Informatics
Department of Computer Science

Computational Effort Reduction of Fault Tolerance in Scalable Agent Support Systems

By: Ashenafi Kassahun

Approved by:

Examining Board:

1. Mulugeta Libsie (Ph. D.), Advisor ____________
2. ______________________________ ____________
3. ______________________________ ____________
4. ______________________________ ____________
5. ______________________________ ____________

Acknowledgement

Firstly I praise GOD for what HE has done. Secondly, I wish to thank thoroughly my advisor, Dr. Mulugeta Libsie, for his supervision, clever advice, patience, and invaluable support throughout this thesis work.

I wish to pass eternal thanks and all my gratitude to my family: to my lovely mother and father, Etete and Kassahun, who was there supporting every step, with tender, inspiring strong will and loving care, Abel, whose presence is a driving force, and all the many others for their unconditional love. I thank you all for your kind, witty, and encouraging presence at all times and you made me who I am right now.

Many thanks go to my friends, who were there for me when I need them. It is crucial to name those who gave their precious time to read the thesis document, to share ideas, and gave me moral and material support. Nesredien Suleiman, Solomon Asres, Solomon Gizaw, Workagegnehu Petros, Gizahnegn Kassa and Meron are few of them. I am very grateful to thank them for what they did.

Last but not least, I would like to sincerely thank all of my friends, colleagues, and classmates for their assistance, encouragement, and inspiration during this research.

Table of Contents

LIST OF TABLES VII

LIST OF FIGURES VIII

LIST OF ACRONYMS X

ABSTRACT XI

1 INTRODUCTION 1

1.1 BACKGROUND 1

1.2 MOTIVATION 6

1.3 RESEARCH PROBLEM 8

1.4 OBJECTIVE 9

1.4.1. GENERAL OBJECTIVE 9

1.4.2. SPECIFIC OBJECTIVES 10

1.5 SCOPE AND LIMITATIONS 10

1.6 ACTIVITIES 11

1.7 APPLICATIONS 11

1.8 ORGANIZATION OF THE THESIS 11

2 LITERATURE REVIEW 12

2.1 BYZANTINE GENERALS PROBLEM 12

2.1.1. RELIABLE SYSTEMS 21

2.2 BYZANTINE QUORUM SYSTEMS 21

2.3 AGENTSCAPE 25

2.4 DARX – INSPIRED FAULT TOLERANCE MECHANISMS 31

2.5 INTEGRATION OF AGENTSCAPE/DARX 37

2.6 SUMMARY 38

3 RELATED WORKS 41

3.1 OPTIMIZING BYZANTINE AGREEMENT 41

3.2 EVENTUAL BYZANTINE AGREEMENT 44

3.3 APPROXIMATE AGREEMENT 44

3.4 FAULT SCALABILITY IN BYZANTINE AGREEMENT 46

3.5 SUMMARY 48

4 DESIGN 50

4.1 BYZANTINE AGREEMENT ALGORITHM 50

4.1.1. SYSTEM MODEL 52

4.1.1.1. PROTOCOLS DEVELOPED 55

4.1.1.2. MAINTAINING STATE OF A REPLICA 58

4.2 BYZANTINE AGREEMENT IN MULTI-AGENT SYSTEM 63

4.3 BYZANTINE AGREEMENT IN AGENTSCAPE/DARX INTEGRATED FRAMEWORK 64

4.4 DESIGN ISSUES 66

4.4.1. SAFETY 66

4.4.2. LIVENESS 67

4.5 SUMMARY 68

5 IMPLEMENTATION AND EXPERIMENT 70

5.1 IMPLEMENTATION ISSUES 70

5.2 EXPERIMENT 74

5.2.1. RESULTS 78

5.3 SUMMARY 80

6 CONCLUSION AND FUTURE WORKS 81

6.1 FUTURE WORK 83

REFERENCES 85

List of Tables

TABLE 1. 1: DIFFERENT TYPES OF FAILURES IN A DISTRIBUTED SYSTEM 2

TABLE 4. 1: DEFINITION OF TERMS USED IN MESSAGES. 60

List of Figures

FIGURE 2. 1: MAJORITY VOTING 13

FIGURE 2. 2: A SCENARIO THAT HAS 4 GENERALS WITH A TRAITOR GENERAL IN OM(0) 17

FIGURE 2. 3: A SCENARIO THAT HAS 4 GENERALS WITH A TRAITOR GENERAL IN OM(1) 17

FIGURE 2. 4: A SCENARIO WITH 3 GENERALS, ONE TRAITOR IN SM 19

FIGURE 2. 5: A SCENARIO WITH 4 GENERALS, ONE TRAITOR IN SM STEP 1 20

FIGURE 2. 6: A SCENARIO WITH 4 GENERALS, ONE TRAITOR IN SM STEP 2 20

FIGURE 2. 7: READING FROM A MASKING QUORUM Q2 23

FIGURE 2. 8: A DISTRIBUTED SYSTEM ORGANIZED AS MIDDLEWARE; NOTE THAT THE MIDDLEWARE LAYER EXTENDS OVER MULTIPLE MACHINES. 26

FIGURE 2. 9: THE AGENTSCAPE CONCEPTUAL MODEL 27

FIGURE 2. 10: AGENTSCAPE MIDDLEWARE ARCHITECTURE 29

FIGURE 2. 11: REPLICATION MANAGEMENT SCHEME 34

FIGURE 2. 12: A SIMPLE AGENT APPLICATION EXAMPLE 35

FIGURE 3. 1: ILLUSTRATION OF FAULT-SCALABILITY. 48

FIGURE 4.1: PROTOCOL COMMUNICATION PATTERNS BETWEEN CLIENT AND REPLICAS 54

FIGURE 4.2: A LOCATION WITH 4 REPLICAS AND A PRIMARY, SHARED BETWEEN THE TWO QUORUMS 57

FIGURE 4.3: MODIFIED REPLICATION MANAGEMENT SCHEME 66

FIGURE 5.1: PSEUDO-CODE FOR CLIENT SIDE 72

FIGURE 5.2: PSEUDO-CODE FOR PRIMARY SIDE 73

FIGURE 5.3: PSEUDO-CODE FOR REPLICA SIDE 74

FIGURE 5.4: STARTUPGUI TO LAUNCH A LOCATION. 75

FIGURE 5.5: INJECTORGUI USED TO INJECT AN AGENT TO A LOCATION (LOC1) 76

FIGURE 5.6: ASADMINGUI USED TO SHOW REPLICAS RUNNING IN A GIVEN LOCATION (LOC1) 77

FIGURE 5.7: MEAN RESPONSE TIME VS. RG SIZE 78

FIGURE 5.8: THROUGHPUT (REQUEST/SEC) VS. NUMBER OF FAULTY REPLICAS 79

List of Acronyms

MAS – Multi – Agent System

BFT – Byzantine Fault Tolerance

DARX – Dynamic Agent Replication eXtensible

Q/U –Query/Update

Abstract

Distributed systems are characterized by partial failures. These partial failures might prohibit the application from performing a system-wide computation and reaching consensus among its components (probably agents). Partial failure might happen due to several reasons, but one of the reasons is the Byzantines failure.

Many researches have dealt with Byzantine failure by proposing different algorithms to solve the problem. Most of the algorithms focus on how to reach on agreement and tolerate fault, while Query/Update (Q/U) protocol discusses the problem of fault tolerance with respect to scalability.

However, these algorithms do not take into account the computational effort it requires, thus making it unattractive for practical use. These techniques do not address the fault scalability of Byzantine faults thoroughly. Hence in this research we looked for better technique to address fault-scalability of Byzantine failures.

To address the Byzantine failure two approaches have been used: the agreement protocol and the quorum protocol. The quorum protocol has been proven to be fault scalable and efficient. The protocol has used the combination of the agreement based and quorum based approaches for the agreement sub-protocol. The agreement sub-protocol focuses on how to achieve a common value from quorums and between quorums. The election sub-protocol has to conduct election of a primary among the existing replicas.

The integration of AgentScape/DARX has been used in the experiment to test the fault tolerance in scalable agent support systems. Performance evaluation through experimentation has been done to assess the scalability fault tolerance support system.

The protocol executes requests in three one-way message delays, which is acceptable latency for agreement on a client request. The protocol is robust to increasing the number of tolerated Byzantine faults and continues to provide significantly higher throughput, achieving scalability feature of the fault tolerant multi-agent support system.

Keywords: Multi-agent support system, Scalable fault tolerance, Byzantine faults

1 Introduction

1.1 Background

A number of definitions of a distributed system has been given by various scholars in the past years, which none of them seem to agree. Tanenbaum and Van Steen [17] define a distributed system as a collection of independent computers that appears to its users as a single coherent system. This definition can be seen from two perspectives:

- Hardware, the machines are autonomous and
- Software, the users think they are dealing with a single system.

Lamport [18] puts a distributed system as a system which consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages.

Sommerville [19] defines a distributed system as a system that runs on a loosely integrated group of cooperating processors linked by a network.

From the above definitions one can come up with a more generalized definition that can summarize the definitions given by saying "A distributed system is a non-centralized network consisting of numerous computers that can communicate with one another and that appear to users as parts of a single, large, accessible shared hardware, software, and data." A distributed system is conceptually the opposite of a centralized, or monolithic, system in which clients connect to a single central computer, such as a mainframe. A distributed system is a system with components that must work together where those components are distributed over a private or public network.

A reliable distributed system should provide services even when partial failure is exhibited. A partial failure may happen when some components in a distributed system fail. This failure might affect the proper operation of other components, leaving yet other components totally unaffected [17]. Tanenbaum and Van Steen [17] give classification schemes of several failure types as shown in Table 1.1.

Table 1. 1: **Different types of failures in a distributed system**

Type of failure	**Description**
Crash failure (fail-stop)	A server halts, but was working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

The main focus of this thesis is the arbitrary failure type, where a server produces responses at arbitrary times.

A failed component may exhibit a type of behavior that is often overlooked- namely, sending conflicting information to different parts of the system. The problems of coping with this type of

failure are expressed abstractly as the Byzantine Generals Problem [8]. The concept is also demonstrated as common knowledge in [30]. The notion of common knowledge, where everyone knows, everyone knows that everyone knows, etc., has proven to be fundamental in various disciplines, including Philosophy, Artificial Intelligence, Game Theory, Psychology, and Distributed Systems. Common knowledge also turns out to be a pre-requisite for agreement and coordinated action. Although common knowledge can be shown to be a prerequisite for day-to-day activities of coordination and agreement, it can also be shown to be unattainable in practice. This is precisely what makes it such a crucial notion in the analysis of interacting groups of agents. On the other hand, in practical settings common knowledge is impossible to achieve.

Byzantine failure was demonstrated for the first time in [8]. The authors illustrated the case of a group of Byzantine Generals whose army is camped around an enemy city. Reliable communication should exist between the Generals so as to come to consensus whether to attack or retreat. If all the Generals attack, they will win. If only some of the Generals attack, they will lose; and if none of the Generals attack, the fight will be postponed to another day. Communication between the Generals is done via messages. The problem here is that some of the Generals may be traitors, trying to prevent the loyal Generals from reaching agreement. The authors discussed the problem in the context of oral messages and written signed messages. The definition of an *oral message* is embodied in the following assumptions which are made for the Generals' message system:

A1. Every message that is sent is delivered correctly

A2. The receiver of a message knows who sent it.

A3. The absence of a message can be detected.

Assumptions A1 and A2 prevent a traitor from interfering with the communication between the other Generals.

With these properties, the paper showed that with three Generals two of them can not reach to a consensus, assuming that one of the Generals is a traitor. Three out of four Generals with one of them a traitor, using the above properties of oral messages, can reach to consensus. Thus, to tolerate traitorous Generals, it requires 3m+1 Generals utilizing m+1 message exchange to reach to consensus, where m is the number of Generals that are traitors.

Byzantine Generals can communicate via *written signed messages*. One can visualize the written signed message as a mail that can be transferred through some communication means. Written signed messages, which assume all the properties of oral messages, are further characterized by the following assumptions:

A4. A loyal General's signature can not be forged, and any alteration of the contents of her/his signed message can be detected.

A5. Any one can verify the authenticity of a General's signature.

Using simple majority voting and the above written signed message properties, it is demonstrated that consensus can be achieved using three Generals, where one of the Generals is a traitor. Thus, to tolerate m traitorous Generals, it requires 2m+1 loyal Generals and m+1 message exchange is necessary to reach to consensus.

Distributed systems exhibit a number of failures as illustrated in Table 1.1. These failures have been dealt with and were tried to be solved. Among these different failure types, arbitrary failure (Byzantine failure) is considered to be the worst case. Since its introduction, the Byzantine

General's problem has been a great concern for those who want to develop a reliable fault-tolerant distributed system.

To more easily identify the problem, concise definitions of a Byzantine fault and a Byzantine failure are presented here under:

Byzantine fault: a fault presenting different symptoms to different observers.

Byzantine failure: the loss of a system service due to a Byzantine fault.

Note that for a system to exhibit a Byzantine failure there must be a system-level requirement for consensus. If there is no consensus requirement, a Byzantine fault will not result in a Byzantine failure [3].

The possibility of partial failure is a fundamental characteristic of distributed applications. The fault-tolerance research community has developed solutions (algorithms and architectures), mostly based on the concept of replication, applied, for instance, to databases [11]. New cooperative applications, for example, air traffic control, cooperative work, and e-commerce, are much more dynamic and large scale. It is thus very difficult, or even impossible, to identify in advance the most critical software components of the application. Furthermore, criticality of a component can vary over run time. Hence information, that should be used to best allocate the scarce resources to the replicated components, is crucial.

Guessoum *et al.* [11] argued that such cooperative applications are now increasingly designed as a set of autonomous and interactive entities, named agents, which interact and coordinate (multi-agent system). Nowadays, such applications are being developed as a set of entities named as agents. Agents are active entities that have a running code, execution state and data to manipulate. Agents communicate with each other by message passing. Agents are different from objects, in such a way

that, the latter are passive entities that are only engaged into computations reactively on an agent's initiative.

In cooperative applications, the roles and relative importance of the agents can greatly vary during the course of computation, of interaction and of cooperation, the agents being able to change roles, and strategies. Also, new agents may join or leave the application [11].

Open multi-agent systems need to cope with the characteristics of distributed systems, for example, dynamic availability of computational resources and latency and diversity of services. Large – scale multi - agent systems employed on wide-area distributed systems are susceptible to both hardware and software failures [1]. The replication concept developed is being used to meet the partial failures of agents in multi-agent systems. However, replicating every agent in an application comprising of millions of agents is likely wastage of resources and increases the communication overload (cost). Particularly it affects the performance of the system. Overeinder *et al.* [1] demonstrated that at any given time, some agents can be lost without significant incidence on the rest of the computation whereas some others are crucial to it. Moreover, this relative importance that every agent has within the application—referred to as its "criticity"—is expected to evolve dynamically.

The major concern of this study is to contribute to the research in Byzantine failure and the reduction of the computational effort it requires by looking for improved techniques.

1.2 Motivation

Nowadays, systems are being developed in a distributed environment. Distributed systems are considered from different points of view; recall the definition given for the distributed system at the beginning of this chapter. Concepts of software engineering, networking, various hardware technologies, etc. are emphasized in distributed systems than other systems.

Business applications, like air traffic controllers, stock exchange systems, etc., are now being developed as distributed systems. This is because the need to give service to the customer has been facilitated by the networking technology available. One can see the very fast expansion of the Internet, company intranets and extranets, virtual private networks (VPN), etc. Using these networking technologies, organizations require systems that run on top of these technologies, to be able to give facilitated services and attract customers.

The main thing here is to develop a system that is fault tolerant. But, distributed systems face the problem usually called partial failure. This partial failure can be caused due to several reasons. Table 1.1 lists the type of failures that distributed systems have. The arbitrary failure, also known as Byzantine failure, is considered to be the worst of all. Different algorithms were developed. But these algorithms do not consider the computational effort required along with the scalability of fault tolerance.

Currently these distributed system applications are being developed by agent-oriented software development, which is a shift from object oriented software development. An application might be composed of multiple agents that are autonomous and communicate with each other to perform a task.

Distributed systems require a middleware to hide heterogeneity of platforms across a network. Different middleware systems have been developed. This thesis uses the AgentScape middleware that has been used with many distributed systems. Also focus on Dynamic Agent Replication eXtensible (DARX), which is a Java framework designed to support the development of fault-tolerant applications built upon multi-agent systems. DARX has a failure detection algorithm that does not consider the Byzantine failure.

The motivation behind this thesis work is to alleviate the problem that distributed systems have with respect to Byzantines failure as the future applications are focusing on developing and implementing of multi-agent systems. There is a need that these systems have to be scalable, fault tolerant, and reliable.

1.3 Research Problem

There are n servers, for which up to t may be faulty and exhibit arbitrary faults (Byzantine faults); the remaining servers function properly. The servers are connected point-to-point by reliable and authenticated links. The system is asynchronous (no bounds on message delays, no local clocks). Every server starts with an initial value and the goal is to agree on a common value, which is the Byzantine Generals problem.

To address this problem, methods were suggested and used practically. Basically, the problem has been investigated since many years and has been modeled as Byzantine failure. Many techniques have been proposed for solving such problems. However, such techniques do not take into account the computational effort it requires, thus making it unattractive for practical use. And also, these techniques do not address the fault scalability of Byzantine faults thoroughly. Thus, Byzantine fault-tolerant agreement-based approaches are not fault-scalable. Performance of such techniques drops as many faults are tolerated. This is due to server-to-server broadcast communication and every correct and functioning server should process a request, which is a requirement. Hence in this research, we look for better techniques to address fault-scalability of Byzantine failures.

AgentScape provides a means for large-scale deployment and interoperability between platforms. Instead of trying to interconnect both the DARX and AgentScape middleware systems while running them concurrently, the current design is to integrate DARX components into AgentScape.

The first design of the integration of DARX agent replication to support fault tolerance is presented in the AgentScape framework [2]. The design premises of both DARX and AgentScape are that the software systems must be open, i.e., able to be used in combination with other software systems. With the integration of both systems, the added value of these design requirements greatly simplified the integration effort.

An extension of the current model is to adapt DARX's failure detection mechanism for replication groups. Instead of servers being responsible for checking the liveness of their neighbors as well as their hosted agents, all replicas will work at detecting failure occurrences inside their own groups. As more than one failure detection relationship may exist between two hosts, the cost of modification may be significant.

Hence, in this research we will implement DARX specific agent servers, to make DARX fault tolerant functionality (including replication management and failure detection) which is the responsibility of the AgentScape middleware.

1.4 Objective

1.4.1. General objective

The major objective of this study is to look for better techniques that solve the problem with a certain degree of approximation, but require little computational effort, so as to make it attractive for practical applications.

Performance evaluation through experimentation is planned to assess the scalability and performance of the fault tolerant agent support system: AgentScape/DARX.

1.4.2. Specific objectives

In order to achieve the above general objective, the following specific objectives (activities) will be accomplished:

- Study the different approaches used to address Byzantine failures.
- Analyze the computational effort different approaches require.
- Demonstrate the fault-scalability of different techniques that have already been practiced.
- Thoroughly study different approaches that are able to decrease computational effort and increase fault-scalability of Byzantine failures.
- Develop a better technique of fault-scalable Byzantine fault-tolerant services
- Performance evaluation through experimentation to assess the scalability and performance of AgentScape/DARX.
- Develop a simulated environment to test the proposed techniques that can be used to solve the problem.

1.5 Scope and Limitations

This thesis work focuses to reduce the computation effort to tolerate faults in multi-agent support systems. Basically, our main strategy is to solve the Byzantine failure in fault scalable multi-agent support systems. Among the different kinds of failures only Byzantine failure is considered. This is due to the fact that Byzantine failure is the worst of other failures. This thesis does not cover how to detect an orphan agent and how to handle them.

1.6 Activities

This thesis work has undergone through number of activities: literature review, to study the problem; review of related works, assessment of what had been done before and results achieved; designing of a model of our proposal; and implementation of the model using selected tools. The tools used are AgentScape and Dynamic Agent Replication eXtensible (DARX). But, DARX was not fully developed. This forced us to develop the major components of the DARX system. Evaluation of the developed system is conducted and shown using charts.

1.7 Applications

Software developers can make use of the integrated framework developed in this thesis work to run multi-agent system in a distributed environment. This thesis work ensures reduced computational effort and higher reliability for multi-agent system systems running in a distributed environment.

1.8 Organization of the Thesis

The rest of this thesis is organized as follows: chapter two shows the literature review, where the classic problem is discussed; chapter three illustrates the related works which had been done in the area of the problem with their results; chapter four shows the proposed design of this thesis; chapter five demonstrates the algorithms used along with the experiment and the results obtained; and chapter six contains the conclusion of this thesis work and the recommended future work of this thesis work.

2 Literature Review

This chapter focuses on the Byzantine Generals problem deeply by considering different Generals being loyals and traitors. Introduction to AgentScape and Dynamic Agent Replication eXtensible (DARX), tools used in this thesis work, are also given. This chapter is organized in five sections: Byzantine Generals Problem, Byzantine Quorum Systems, AgentScape, DARX, and integration of AgentScape/DARX.

2.1 Byzantine Generals Problem

The majority of this section is taken from the discussion by Lamport *et al.* [8]. Additional explanations and figures are also given. Papers of different scholars are used and referred as appropriate.

The Classic Byzantine Generals Problem states that

- Each division of Byzantine army are directed by its own general
- Generals, some of which are traitors, communicate each other by messengers

The problem can be restated as:

- All loyal generals receive the same information upon which they will somehow get to the same decision
- The information sent by a loyal general should be used by all the other loyal generals

The above problem can be reduced into a series of one commanding general and multiple lieutenants' problem - **Byzantine Generals Problem*:***

- All **loyal** lieutenants obey the **same** order
- If the commanding general is **loyal**, then every **loyal** lieutenant **obeys** the order s/he sends

A failed component of a distributed system may send conflicting information to different parts of a system. One of the concerns of research in distributed systems is to build reliable systems in the presence of faulty components.

The common approach followed is:

- Have multiple (potentially faulty) components compute same function
- Perform majority vote on outputs to get "right" result. The majority voting, shown in Figure 2.1, shows that C1 and C2 produce the same result and C3 has a different result. Each of the Ci is a component in the distributed computation. The majority voting function will basically choose the majority of the results.

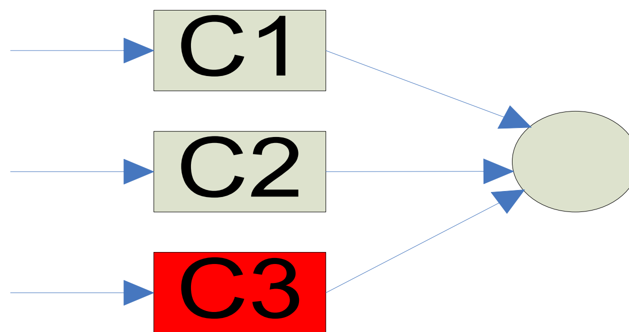


Figure 2. 1: Majority voting

If the number of faulty components is f, and there are f+1 good components, then there are 2f+1 components totally. The assumption is non-faulty components must use same input, otherwise can't trust the output. For majority voting to work:

1) All nonfaulty processes must use same input.

2) If input is nonfaulty, then all nonfaulty processes use the value it provides.

Byzantine Failure. In fault-tolerant distributed computing, a Byzantine failure is an arbitrary fault that occurs during the execution of an algorithm by a distributed system. It encompasses those faults

that are commonly referred to as arbitrary failures. When a Byzantine failure has occurred, the system may respond in any unpredictable way.

The problem of coping with Byzantine failures is known as the Byzantine Generals' problem. An algorithm to achieve agreement among "loyal generals" (i.e., working components) given m "traitors" (i.e., faulty components) should be designed. The Generals must have an algorithm such that:

A. All loyal Generals decide upon the same plan of action

B. Small number of traitors cannot cause the loyal Generals to adopt a "bad plan".

Let v(i) be information communicated by the ith General and all generals use same method for combining information.

Then combine values v(1) … v(n) and decision is majority function of values v(1) ... v(n)

Then the following should hold:

1. Every loyal General must obtain the same information v(1),…, v(n).

   Any two loyal Generals use the same value of v(i). A traitor will try to make loyal Generals to use different v(i)'s.

2. If the ith General is loyal, then the value that s/he sends must be used by every loyal General as the value of v(i).

The question here is that "How can each General send her/his value to n-1 others?". A General must send a message to n-1 Generals such that:

IC1. All loyal Generals obey the same order

IC2. If the commanding General is loyal, then every loyal General obeys the order s/he sends.

Conditions IC1 and IC2 are called the *interactive consistency* conditions.

With the above assumptions, there is a condition where the number of Generals available cannot reach to consensus.

- For $n = 3$ Generals and 1 traitor, there is no solution (protocol). This is because a loyal General cannot distinguish who the traitor is when s/he gets conflicting information from the two Generals. Let's call this the *3-Generals Problem*.
- Byzantine Generals Problem for $n < 3m+1$ Generals and m traitors can be reduced to the 3 - Generals problem, with each of the Byzantine generals simulating at most m Generals and taking the same decision as the loyal Generals they simulate. Thus the Byzantine Generals Problem for $n < 3m+1$ and m traitors is not solvable.
- The Generals have a problem even if they want to agree on time duration to attack, i.e., reaching approximation is as hard as reaching agreement.

The solution to the problem is given in the context of *oral messages* and *written signed messages*.

One can inductively define the oral messages algorithm OM(m), for all non-negative integers m, by which a General sends an order to n-1 Generals. The algorithm assumes a function *majority* with the property that if a majority of values v_i equal v, then majority($v_1,\ldots,v_{n-1}$) equals v. There are two natural choices for the value of majority($v_1,\ldots,v_{n-1}$):

1. The majority value among the v_i if it exists, otherwise the value RETREAT;
2. The median of the v_i, assuming that they come from an ordered set.

The following algorithm requires only the aforementioned property of majority.

Algorithm OM(0)

1. The General sends her/his value to every other General.
2. Each General uses the value s/he receives, or uses the value RETREAT if s/he receives no value.

NB: 0 refers to the initial message sent from a General

Algorithm OM(m), m>0

1. The General sends her/his value to every other General.
2. For each i, let v_i be the value General i receives, or else be RETREAT if s/he receives no value. General i acts as the General in Algorithm OM(m-1) to send the value v_i to each of the n-2 other Generals.
3. For each i, and each $j \neq i$, let v_j be the value General i received from General j in step (2) (using Algorithm OM(m-1)), or else RETREAT if s/he received no such value. General i uses the value majority $(v_1,\ldots,v_{n-1})$.

Consider a scenario that has 4 Generals with a traitor General, i.e., m=1, n=4, and General 4 (G4) is a traitor. General 1 (G1) sends a message attack (A) to all the other Generals.

OM(0):

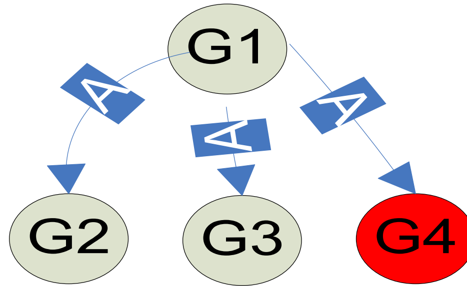


Figure 2. 2: A scenario that has 4 Generals with a traitor General in OM(0)

OM(1):

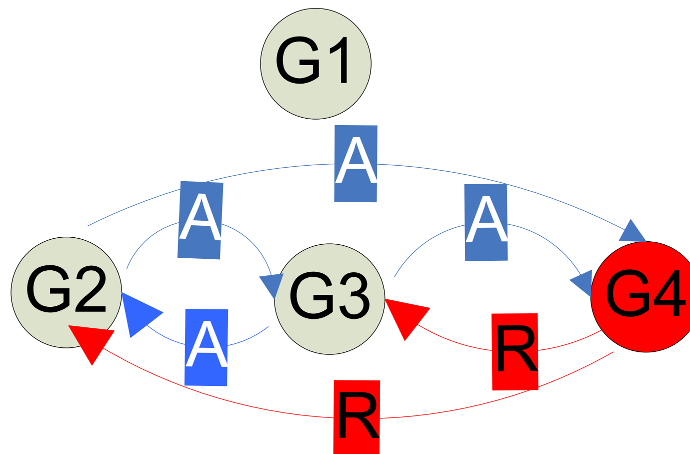


Figure 2. 3: A scenario that has 4 Generals with a traitor General in OM(1)

General 2 (G2) and General 3 (G3) send a message attack (A) to each other and to General 4 (G4). But General 4 (G4) sends a value retreat (R) for both General 1 (G1) and General 4 (G4), falsifying the value s/he receives from General 1 (G1). General 2 (G2) and General 3 (G3) uses the majority value and decide to attack (A).

Byzantine Generals can communicate via *written signed messages*. One can visualize the written signed message as a mail that can be transferred through some communication means. Written signed messages, assume all the properties of oral messages (A1-A3) and are further characterized by the following assumptions:

A4) (a) Loyal General's signature cannot be forged, and contents cannot be altered

(b) Anyone can verify authenticity of a General's signature.

It can be observed that this assumption simplifies the problem:

- When General i passes on signed message from j, j knows that i did not lie about what j said.
- Generals cannot do any harm alone (cannot forge loyal General's message).
- Only have to check the sender Generals for being traitor or loyal.

The signed message (SM) algorithm assumes a function *choice* which is applied to a set of orders to obtain a single one. The only requirements of this function are:

1. If the set V consists of the single element v, then choice(V) = v.
2. Choice(Ø) = RETREAT, where Ø is the empty set.

In this algorithm, each General i maintains a set v_i, containing the set of properly signed orders s/he has received so far. (If a General is loyal, then this set should never contain more than a single element).

Algorithm SM (m)

Initially all the Generals have empty values as message, i.e., V_i= Ø.

1. The General signs and sends her/his value to every other General as (v:0).
2. For each General i:
    a. If General i receives a message of the form v:0 from the message source General and s/he has not yet received any message, then
        i. s/he lets v_i equal {v};

ii. s/he sends the message v:0:i to every other General.

b. If General i receives a message of the form $v:0:j_1:\ldots:j_k$ and v is not in the set V_i, then

i. s/he adds v to V_i;

ii. if k<m, then s/he sends the message $v:0: :j_1:\ldots:j_k:i$ to every General other than $j_1:\ldots:j_k$.

3. For each i: when General i will receive no more messages, s/he obeys the order choice (V_i).

Algorithm SM (1) with 3 Generals, having m=1, n=3 and General 3 (G3) is a traitor:

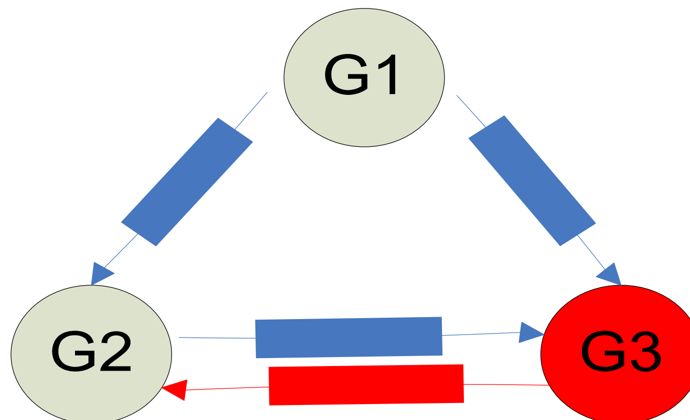


Figure 2. 4: A scenario with 3 Generals, one traitor in SM

General 2 (G2) receive the two messages in step (2), so after step (2) V2 = {“attack”, ”retreat”}. Since the written signed message won’t let any General to forge a loyal General’s signature, s/he is able to know that General 3 (G3) is a traitor. G2 will obey the message “attack”.

Algorithm SM (1) with 4 Generals, having m=1, n=4 and General 4 (G4) is a traitor:

Step 1:

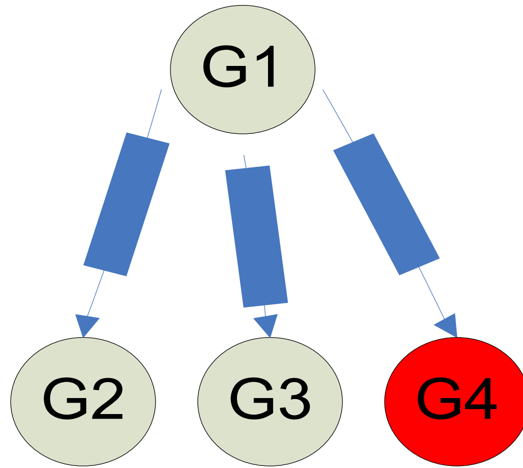


Figure 2. 5: A scenario with 4 Generals, one traitor in SM step 1

Step 2:

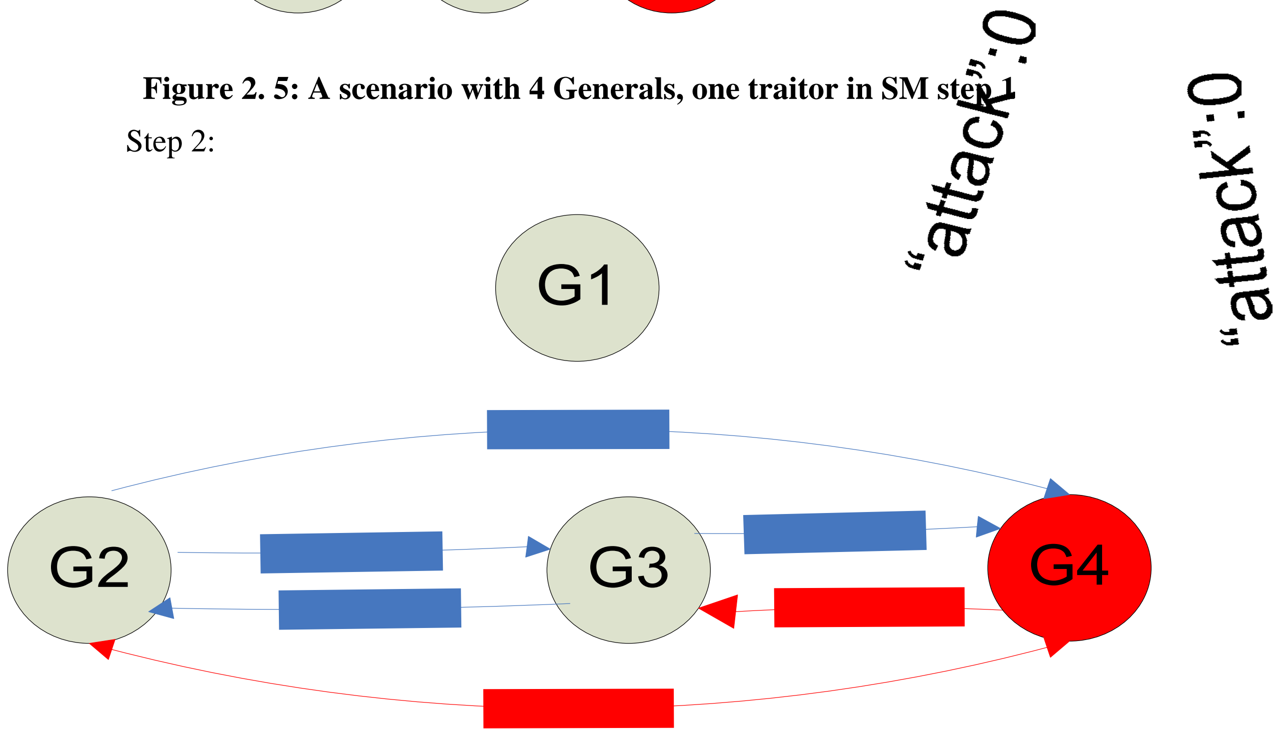


Figure 2. 6: A scenario with 4 Generals, one traitor in SM step 2

Generals (G2, G3, and G4) receive the message in step (2). G4 will send “retreat” at step (3). After step (3):

$$V_2 = V_3 = V_4 = \{\text{“attack”, “attack”, “retreat”}\}$$

And they obey the message choice ({“attack”, “attack”, “retreat”}). According to the signed message algorithm, the Generals can make a decision “attack”.

2.1.1. Reliable Systems

Driscoll *et al.* [7] argued that the Byzantine Generals problem is not a myth and the problem is very real from practitioners' point of view. Byzantine faults can be prevented from causing Byzantine problems by designing systems such that consensus is not needed. However, it is extremely difficult or impossible to design highly reliable systems that do not need some form of consensus. The only practical solution is Byzantine fault tolerance; which is also known as consistency or congruency. Anyone designing a system with high dependability requirements must thoroughly understand these failures and how to combat them.

The best way to provide fault-tolerant decision-making in redundant systems is by majority voting. Faulty input devices may generate meaningless inputs, but majority voting would ensure that the same meaningless values are used.

For majority voting to yield a reliable system, the following two conditions must be satisfied

1. All non-faulty processors must use the same input value
2. If input unit is non-faulty, then all non-faulty processes use the value it provides

But these are just the requirements of the Byzantine Generals Problem!

2.2 Byzantine Quorum Systems

To improve the efficiency and availability of service while still protecting the integrity of replicated service, the use of quorum systems has been proposed. Quorum systems are a family of protocols that allow reads and updates of replicated service to be performed at only a subset (quorum) of the servers. In a t-masking quorum system, the quorums of servers are defined such that any two quorums intersect in at least 2t + 1 servers. In a system with a maximum of t faulty servers, if each

read and write operation is performed at a quorum, then the quorum used in a read operation will intersect the quorum used in the last preceding write operation in at least t + 1 correct servers. With appropriate read and write protocols, this intersection condition ensures that the client is able to identify the correct, upto-date data service.

In this section we introduce masking quorum systems, which can be used to mask the arbitrarily faulty behavior of data repositories. To motivate our definition, suppose that the replicated variable x is written with quorum Q_1, and that subsequently x is read using quorum Q_2. If B is the set of arbitrarily faulty servers, then the following is obtained by reading from Q_2: the correct value for x is obtained from each server in $(Q_1 \cap Q_2) \setminus B$ (see Figure 2.2 in [26]); out-of-date values are obtained from $Q_2 \setminus (Q_1 \cup B)$; and arbitrary values are obtained from $Q_2 \cap B$. In order for the client to obtain the correct value, the client must be able to identify the most up-to-date value/timestamp pair as one returned by a set of servers that could not all be faulty. This yields requirement M-Consistency below. In addition, since communication is asynchronous and thus accurate failure detection is not possible, in order for a client to know it completes an operation with all the correct servers of some quorum, it must be able to obtain responses from a full quorum. Therefore, for availability we require that there be no set of faulty servers that intersects all quorums.

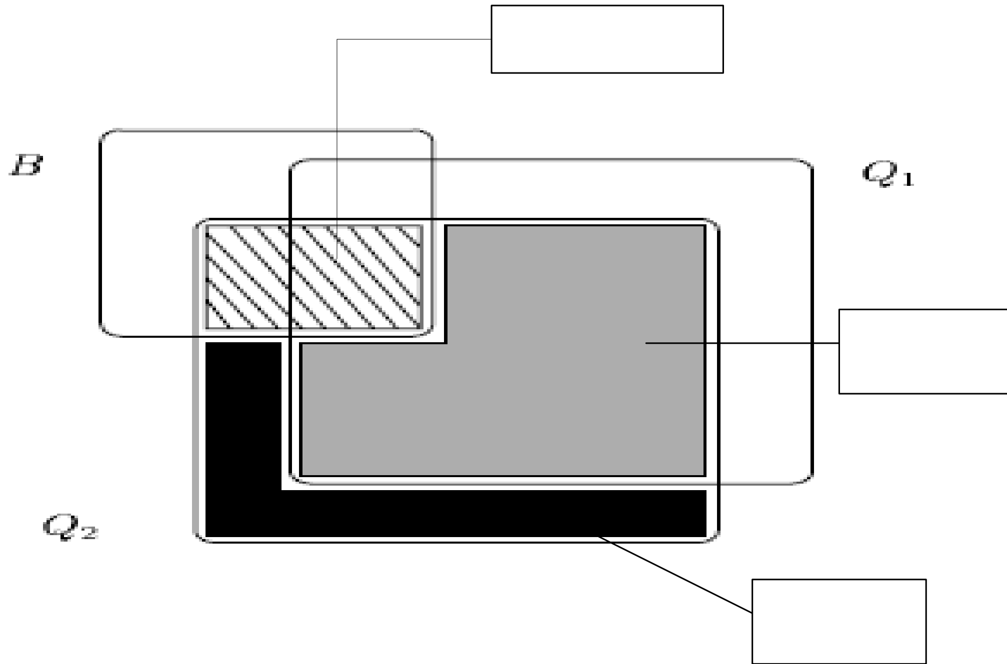


Figure 2. 7: Reading from a masking quorum Q_2

Definition: A quorum system Q is a masking quorum system for a fail-prone system B, if the following properties are satisfied.

M-Consistency: $\forall Q_1, Q_2 \in Q \forall B_1, B_2 \in B : (Q_1 \cap Q_2) \setminus B_1 \not\subseteq B_2$

M-Availability: $\forall B \in B \exists Q \in Q : B \cap Q = \phi$

Let's consider quorum systems which intersect, not just in a single party, making them intolerant to any faults in these parties, but rather in enough parties to complete the protocol in a way tolerant of up to f faults, f>0. One way to do this is called masking which constrains quorum systems to satisfy the following:

M-Consistency: None of the sets of correct servers which contain the latest value is a bad coalition. To achieve this, any quorum intersection minus any bad collation should not be a subset of any other bad coalition. Formally: for all Q1, Q2 element of GC and all B1, B2 element of BC, (Q1 intersect Q2) - B1 is not a subset of B2:

$$\forall Q_1, Q_2 \in Q \forall B_1, B_2 \in B : (Q_1 \cap Q_2) \setminus B_1 \not\subseteq B_2$$

M-Availability: No bad coalition can disable all quorums. Formally: for all B element of BC there exists some Q element of GC such that B and Q are disjoint:

$$\forall B \in B \exists Q \in Q : B \cap Q = \phi$$

These constrains are sufficient to ensure that a replicated system is updated consistently and correctly.

For example, in the case that at most f servers can fail, M-Consistency guarantees that every pair of quorums intersect in at least 2f + 1 elements, and thus in f + 1 correct ones. If a read operation accepts only a value returned by at least f + 1 servers, then any accepted value was returned by at least one correct server.

Malkhi and Reiter [26] describe that a well known way to enhance the availability and efficiency of replicated service is by using quorums. A quorum system for a universe of servers is a collection of subsets of servers, each pair of which intersects. Intuitively, each quorum can operate on behalf of the system, thus increasing its availability and performance, while the intersection property guarantees that operations done on distinct quorums preserve consistency. They consider the arbitrary (Byzantine) failure of clients and servers, and initiate the study of quorum systems in this model. Intuitively, a quorum system tolerant of Byzantine failures is a collection of subsets of servers, each pair of which intersect in a set containing sufficiently many correct servers to guarantee consistency of the replicated data as seen by clients.

They showed:

- Definition of the class of masking quorum systems with which data can be consistently replicated in a way that is resilient to the arbitrary failure of data repositories.

- Two variations of masking quorum systems. The first, called dissemination quorum systems, is suited for services that receive and distribute self-verifying information from correct clients (for example, digitally signed values) that faulty servers can fail to redistribute but cannot undetectably alter. The second variation, called opaque masking quorum systems, is similar to regular masking quorums in that it makes no assumption of self-verifying data, but it differs in that clients do not need to know the failure scenarios for which the service was designed. This somewhat simplifies the protocol by which clients access the replicated service and, in the case that failures are maliciously induced, reveals less information to clients that could guide an attack attempting to compromise the system.
- The load of each type of quorum system, where the load of a quorum system is the minimal access probability of the busiest server, minimizing over all strategies for picking quorums.

Alvisi et al. [27] states the difficulty of using quorum systems for Byzantine fault tolerance is that detecting responsive but faulty servers is hard. In state machine replication, any server response that disagrees with the response of the majority immediately exposes the failure of the disagreeing server to the client. This property is lost, however, with quorum systems: because some servers remain out of date after any given write, a contrary response from a server in a read operation does not necessarily suggest the server's failure. Therefore, we must design specific mechanisms to monitor the existence of faults in a quorum-replicated system, for example, to detect whether the number of failures is approaching to threshold where agreement can not be reached.

2.3 AgentScape

One of the characteristics of distributed systems is platform heterogeneity. For a distributed system to be able to be implemented and become functional, it has to handle the heterogeneity of platforms.

Since different platforms operate under different principles and concepts, distributed systems require a way that eliminates the heterogeneity of platforms and make them interoperable.

To support heterogeneous computers and networks and to provide a single-system view, a distributed system is often organized by means of a layer of software called middleware that extends over multiple machines [17]. Tanenbaum and Van Steen [17] illustrate middleware by using Figure 2.3.

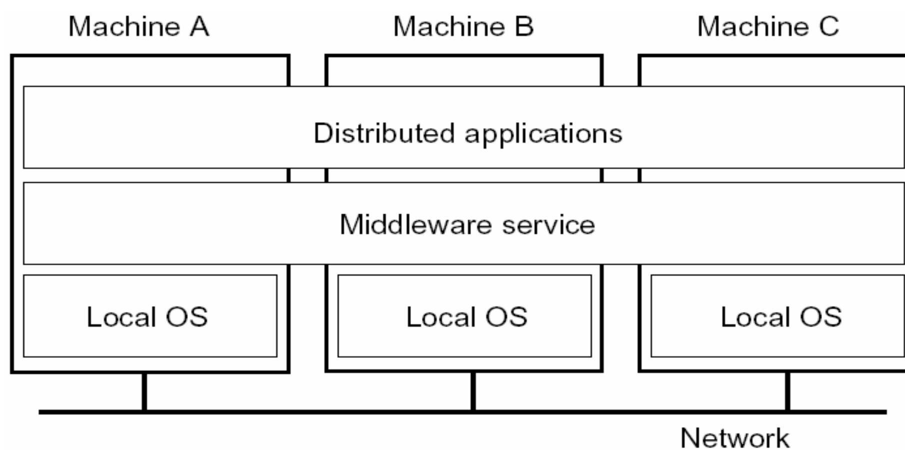


Figure 2. 8: A distributed system organized as middleware; note that the middleware layer extends over multiple machines.

AgentScape is a middleware layer that provides multi-agent system support environment. The rationale behind the design decisions of AgentScape are:

(i) To provide a platform for large-scale agent systems,

(ii) Support multiple code bases and operating systems, and

(iii) Interoperability with other agent platforms [12].

AgentScape middleware is designed to provide a minimal but sufficient support for agent applications, and to be adaptive or reconfigurable such that it can be tailored to a specific application (class) or operating system/hardware platform.

Agents and *objects*, the basic entities in AgentScape, reside in a *location* (see Figure 2.4 [1]).

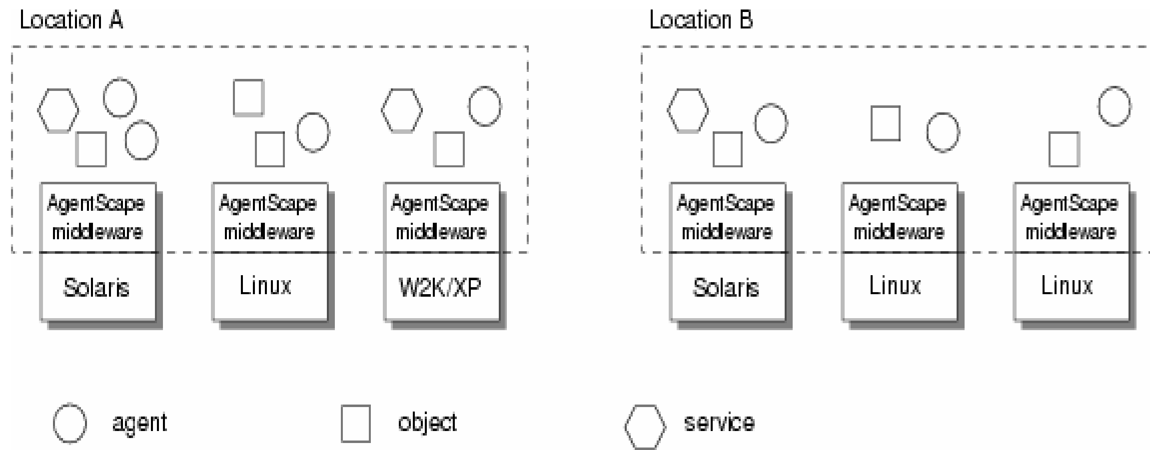


Figure 2. 9: The AgentScape Conceptual Model

Agents are entities in AgentScape that interact with each other by message-passing. Furthermore, agent migration in the form of weak mobility is supported. Weak mobility refers to agent migration with its piece of running code and data, not with its execution state. Objects are passive entities that are only engaged in computations reactively on an agent's initiative. Besides, AgentScape defines services. Services provide information or activities on behalf of agents or the AgentScape middleware. Agent-agent interaction is exclusively via message-passing.

In a message-passing (oriented) communication, communication can be asynchronous or synchronous [17, 31, 32, 33]. The characteristic feature of asynchronous communication is that a sender continues immediately after it has submitted its message for transmission. This means that the message is stored in a local buffer at the sending host, or otherwise at the first communication server. With synchronous communication, the sender is blocked until its message is stored in a local buffer at the receiving host, or actually delivered to the receiver. The strongest form of synchronous communication is when the sender is blocked until the receiver has processed the message.

Scalability is one of the most important design goals for developers of distributed systems [17]. Scalability of a system can be measured along at least three different dimensions. First a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system. Second, a geographically scalable system is one in which the users and resources may lie far apart. Third, a system can be administratively scalable, meaning that it can still be easy to manage even if it spans many independent administrative organizations. It is clear that geographical scalability will be limited due to the performance and reliability problems resulting from wide-area communication. One of the main reasons why it is currently hard to scale existing distributed systems that were designed for Local Area Networks is that they are based on synchronous communication. Hiding communication latencies is applicable in the case of geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote service requests as much as possible, i.e., constructing the requesting application in such a way that it uses only asynchronous communication.

Asynchronous communication for message-passing has good scalability characteristics with a minimum of synchronization between the agents.

AgentScape Architecture

The AgentScape Operating System (AOS), which is modularly designed, forms the AgentScape middleware. An overview of the AgentScape architecture is shown in Figure 2.5 [28].

The AOS offers a uniform and transparent interface to the underlying resources and hides various aspects of network environments, communication, operating system, access to resources, etc. The AgentScape API is the interface to the middleware. Both agents and services (for example, resource management and directory services) use this interface to the AOS middleware. Agent Servers provide the interface and access to AgentScape to the agents that are hosted by the agent server.

Similarly, Object Servers provide access to the objects that are hosted by the object server. Services are made accessible in AgentScape by service access providers.

A location is a closely related collection of agent and object servers, possibly on the same (high speed) network, on hosts which are managed by the same administrative domain.

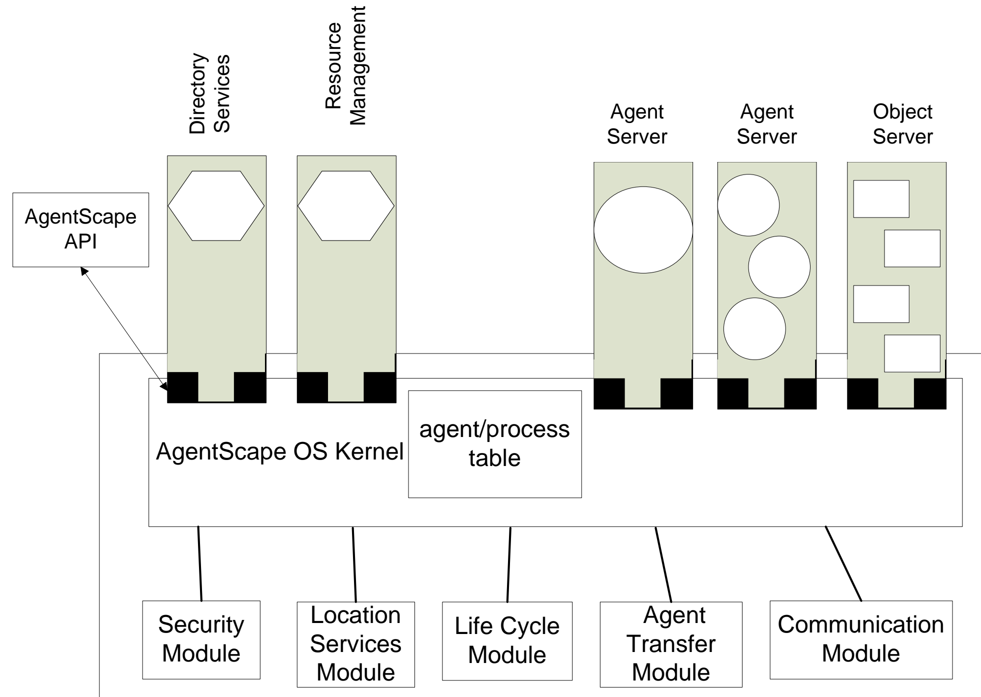


Figure 2. 10: AgentScape middleware architecture

Depending on the policy or resource requirements, one agent can be exclusively assigned to one agent server, or a pool of agents can be hosted by one agent server. An active agent is assigned to, and runs on a server; a suspended agent is not assigned to an agent server [13].

The modules in the AOS middleware provide the basic functionality. Below a brief overview of the most important modules is given.

The ***life-cycle module*** is responsible for the creation and deletion of agents.

The ***communication module*** implements a number of communication services, similar to TCP, HTTP, and streaming, with different qualities of service.

Support for agent mobility is implemented in the ***agent transfer protocol module,*** or also known as ***migration module***.

The ***location service*** associates an agent identifier with its contact address. There are also location services for objects, services, and locations.

The ***security module*** is essential in the design of AgentScape, as it is an integral part of the middleware. Many modules in the middleware have to request authentication or authorization in order to execute their tasks.

AgentScape Prototype

A prototype implementation of the AgentScape architecture is currently available and provides the following basic functionality: creation and deletion of agents, communication between agents and middleware, and weak migration of agents.

Distributed shared (replicated) objects in AgentScape will be supported by the Globe system. Globe is a large-scale wide area distributed system that provides an object-based framework for developing applications.

Multiple programming languages are available at the application level (i.e., building agents and objects in a programming language of choice).

2.4 DARX – inspired fault tolerance mechanisms

A widely used technique to address software failures is replication. Replication of different components of an open multi-agent system can alleviate failure problems. Hence an open multi-agent system can cope the different characteristics of a distributed system.

However, replicating every agent in an application comprising millions of agents is likely to undermine the overall determination of the result, particularly in terms of performance [1]. Guessoum et al. [11] reckon that at any given time, some agents can be lost without significant incidence on the rest of the computation whereas some others are crucial to it. Moreover, this relative importance that every agent has within the application—referred to as its "criticity"—is expected to evolve dynamically.

DARX [14] is a framework designed to support the development of fault-tolerant applications built upon multi-agent systems. The authors of [4] assume a distributed system, in which agents are independent processes; they communicate by message-passing only. A process is assumed to be fail-silent, which results in the prevention of participation to the global computation anymore.

Byzantine behaviors might be resolved, but are not integrated in the failure model. But, as illustrated earlier, Byzantine failures are worst cases than any other types of failures. Even these failures are not considered while failures are detected.

Replication Management

DARX provides fault tolerance through software replication. It is designed in order to be agent-dependent, as opposed to location-dependent[1]. That is, if a machine/location crashes, it can be

[1]A location is an abstraction of a physical location. It hosts resources and processes, and possesses its own unique identifier. DARX uses a URL and a port number to identify each location.

restarted and the state of the agent replicas restored but no server information needs to be recovered. The reason for this choice comes from the fundamental assumption that criticity[1] of an agent evolves over time; therefore, at any given time of the computation, all agents do not require to benefit from the same fault tolerant mechanism.

In order to minimize interference with the application development process, replication management is kept transparent to the supported application. DARX handles replication groups (RGs). Each of these groups consists of software entities (replicas) which represent the same agent.

To maintain consistency in RGs several strategies are made available in the DARX framework. These strategies can be classified into two main types:

(i) Active, where all replicas process the input messages concurrently.

(ii) Passive, where only one is in charge of computation while periodically transmitting its state to the other replicas.

But, in the same RG several strategies can coexist. Even, it is possible to switch between strategies. An RG is described by its replication policy which constitutes:

- the *criticity* of its associated agent,
- its replication degree—the number of replicas it contains,
- the list of these replicas, ordered by potential of leadership[2],

---

[1]The criticity of a process defines its importance with respect to the rest to the application. Its value is subjective and evolves over time. For example, towards the end of a distributed computation, a single agent in charge of federating the results should have a very high criticity; whereas at the application launch, the criticity of that same agent may have a much lower value.

[2] The potential of leadership is the capacity of a replica to represent its RG. The strategy used to keep a replica consistent is the main parameter in the calculation of this potential. The other parameters emanate

- the list of the replication strategies applied inside the group,
- The mapping between replicas and strategies.

A replication policy must be reevaluated in three cases:

(i) When a failure inside the RG occurs,

(ii) When the *criticity* value of the associated agent changes, and

(iii) When the policy cannot be enforced due to environment variations such as CPU or network overloads.

A leader is elected within the RG for consistency maintenance and updating the replication policy. Its objective is to adapt the replication policy to the criticity of the associated agent as a function of the characteristics of its context – the information obtained through observation. Replicas and Strategies can be added to or removed from a group during the course of the computation, and it is possible to switch from one strategy to another.

---

from the DARX integrated observation service; they include the load of the host, the date of creation of the replica, the latency in the communications with the other replicas of the group.

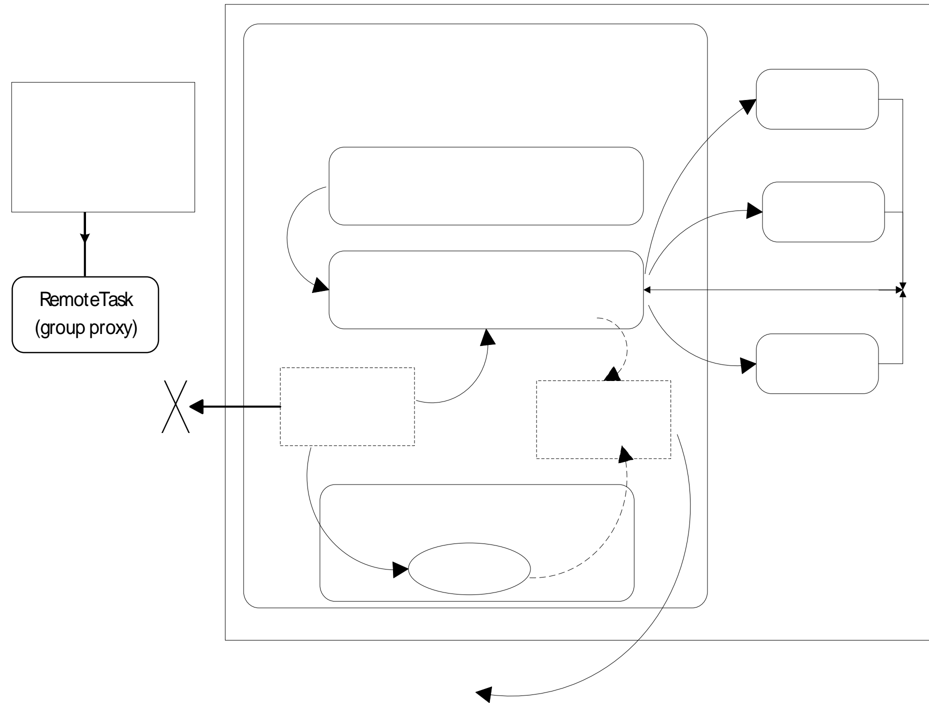


Figure 2. 11: Replication management scheme

Figure 2.6 depicts the composition of a replica [1]. In order to benefit from fault tolerance abilities, each agent inherits the functionalities of a DarxTask object, enabling DARX to control the agent execution. Each task is itself wrapped into a TaskShell, which handles the agent input/output. Leaders are wrapped in enhanced shells, containing an additional ReplicationManager. This manager collects information about the environment – network load, memory available, etc. and performs the periodical reassessment of the replication policy.

Proxies are used to enable communication between agents. They reference RGs. Proxies are the naming service which keep track of every replica to be referenced by providing the corresponding RemoteTask.

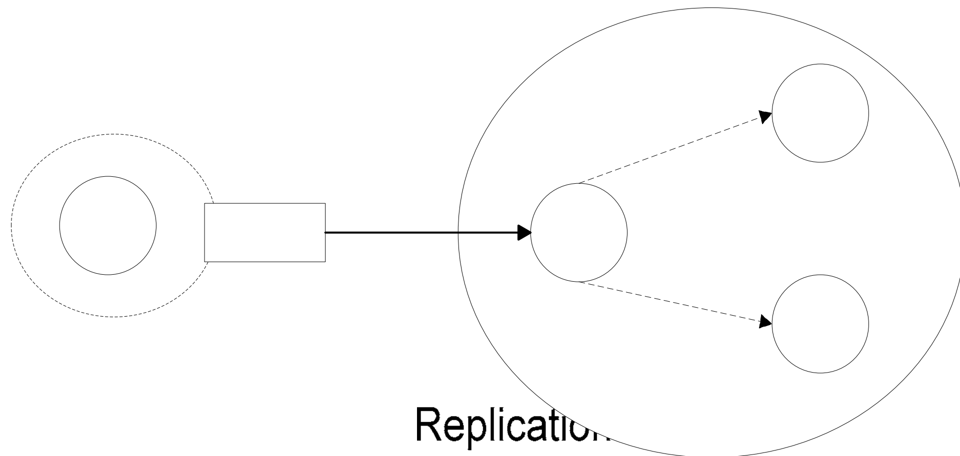


Figure 2. 12: A simple agent application example

Overeinder *et al.* [1] illustrate a simple agent application example as depicted in Figure 2.7 and as seen in the DARX context. An emitter, agent B, sends messages to be processed by a receiver agent A. The RG, that represents agent A, consists of three replicas:

1) An active replica A elected as the leader,

2) A follower A' to which incoming messages are forwarded, and

3) A backup A" which receives periodical state updates from A.

Figure 2.7 shows that the criticity value of agent B is minimal, and that of agent A is higher.

In order to transmit messages to A, B requestes the relevant RemoteTask RTA from the naming service. Since replication group A contains only one active replica, RTA references replica A and no other.

Failure Detection

The major goal of failure detection service in DARX is to maintain dynamic lists of the valid replicas participating in the application. Failure detectors are organized in groups; they exchange

heartbeats[1] and maintain a list of the processes which are suspected of having crashed. These heartbeats are subsequently released by the member as proof of its continued presence in the group. Each heartbeat contains enough information to assess the freshness and authenticity of the message. Thus, we need not ensure reliable delivery. Therefore, in an asynchronous context, failures can be recovered more efficiently [15].

Every DARX entity integrates an independent thread which acts as a failure detector. It sends heartbeats to the other group members and collects the incoming heartbeats. One can see that there is a question of performance and scalability. Once an entity has been suspected several times in a row, it is considered as having crashed. A detector which suspects a crash submits the information to the rest of the group. If all the other group members confirm the crash, then the suspected entity is excluded from the group.

Replication Policy Adjustment

DARX aims at providing decision-making support so as to fine-tune the fault tolerance for each agent with respect to its evaluation and that of its context. Decisions of this type may only be reached through a fairly good knowledge of the dynamic characteristics of the application and of the environment [16].

DARX is a Java framework. It includes several generic classes which assist the application developer through the process of implementing a reliable multi-agent application.

Every agent class must extend a DarxTask for several reasons:

- Because the DarxTask is the point where DARX handles the execution of an agent, application specific control methods to start, stop, suspend and resume agents have to be defined for this purpose.

---

[1] Messages containing information that is generated by an agent wishing to state its continued presence in the group, and validated by interested members.

- The state of an agent is essential in determining its criticity. Any number of states can be defined for the agent; each of these states is to be mapped to a corresponding criticity in the code of the ReplicationManager, and every criticity corresponds a user-defined[1] replication policy which will be applied at runtime.

Although generic replication strategies are implemented, fault tolerance protocols that are specific to the application can be developed. DARX provides a generic ReplicationStrategy class which may be extended to fulfill the needs of the programmer. Basic methods allow defining the consistency information within the group, as well as the way this information ought to be propagated in different cases, such as synchronous and asynchronous messages, for example.

2.5 Integration of AgentScape/DARX

AgentScape provides a means for large scale deployment and interoperability between platforms. Instead of trying to interconnect both the DARX and AgentScape middleware system while running them concurrently, the current design is to integrate DARX components into AgentScape. The modular architecture of both DARX and AgentScape makes this possible.

Agents developed along the DARX guidelines (refer to the previous section), make use of a DARX specific runtime system (RTS) and are hosted by regular AgentScape agent servers.

Agents are extended with the DARX RTS to provide the functionality for fault tolerance. The DARX fault tolerant RTS associated with an agent can make calls to the AgentScape middleware to communicate with the pears in the RG or to obtain monitoring information for the observation service, etc. Monitoring information (for example load of host, date of creation of replicas,

[1] User can define a replication policy that can be used for that specific application. This policy can be adopted and used in the ReplicationManager.

communication latencies to other replicas in the groups), is provided by an observation service that acquires its information from the AgentScape management system.

An extension of the current model is to adapt DARX's failure detection mechanism for RGs. Instead of services being responsible for checking the liveliness of their neighbors as well as their hosted agents, all replicas will work in detecting failure occurrences inside their own groups. Hence, this thesis focuses on using the DARX framework as a means of developing a reliable application built upon multi-agent system and filling the gap that DARX has.

The AgentScape middleware can be used in different families of operating systems. Its major service is to hide heterogeneity of the underlying platform. It gives support to distributed systems to run on different platforms. But AgentScape does not have a way to handle errors while the distributed application is running. If some processes have faults associated with them, AgentScape does not give a service that works to address such kinds of faults.

The DARX framework alleviates the problem of distributed systems by providing agent replication. Services, like replication manegement, observation, naming and localization services, are used to perform failure detection. But DARX does not have a protocol that detects Byzantine failure.

We use the integration of AgentScape and DARX to conduct the experiment of our protocol. AgentScape/DARX is used in our work due to the feature it holds and it is an open source system.

2.6 Summary

In this chapter, the Byzantine Generals' problem was presented. The solutions provided were covered in accordance with oral message and written signed messages. Lamport *et al.* [8] showed for m traitorous Generals there should be 3m+1 Generals to reach to consensus using oral message communication. It only requires 2m+1 Generals to tolerate m traitorous Generals using written

signed messages communication. In both cases, m+1 number of message exchange between the Generals is required.

Another approach is the quorum system. A quorum system for a universe of servers is a collection of subsets of servers, each pair of which intersects. Intuitively, each quorum can operate on behalf of the system, thus increasing its availability and performance, while the intersection property guarantees that operations done on distinct quorums preserve consistency.

In this thesis work we consider the arbitrary (Byzantine) failure of clients and servers, and initiate the study of quorum systems in this model. Intuitively, a quorum system tolerant of Byzantine failures is a collection of subsets of servers, each pair of which intersect in a set containing sufficiently many correct servers to guarantee consistency of the replicated data as seen by clients.

One of the characteristics of distributed systems is platform heterogeneity. For a distributed system to be able to be implemented and become functional, it has to handle the heterogeneity of platforms. To hide heterogeneity, a software layer called middleware is used. AgentScape is one of the middleware systems developed. It supports large scale agent systems. The model, architecture, and prototype of AgentScape have been discussed briefly.

This chapter also covered DARX. It is a framework designed to support the development of fault-tolerant applications built upon multi-agent systems. Basically DARX provides fault tolerance through software replication. The replication management, failure detection, and replication policy adjustment in DARX are thoroughly presented. How agent replicated groups are managed and their criticity handled are discussed. The failure detection service dealt with how information is transferred between groups. The replication policy adjustment section describes how DARX empower the developers of multi-agent systems to implement user defined replication policy.

AgentScape provides a means for large-scale deployment and interoperability between platforms. Instead of trying to interconnect both the DARX and AgentScape middleware systems while running them concurrently, the current design is to integrate DARX components into AgentScape.

A first design is presented of the integration of DARX agent replication to support fault tolerance in the AgentScape framework. The design premises of both DARX and AgentScape are that the software systems must be open, i.e., able to be used in combination with other software systems. With the integration of both systems, the added value of these design requirements greatly simplifies the integration effort.

An extension of the current model is to adapt DARX's failure detection mechanism for replication groups. Instead of servers being responsible for checking the liveness of their neighbors as well as their hosted agents, all replicas will work at detecting failure occurrences inside their own groups. As more than one failure detection relationship may exist between two hosts, the cost of modification may be significant.

The failure detectors in DARX should reach to consensus. Failure detection and replication policy adjustment used do not consider how consensus can be reached. It obviously leads to Byzantine Generals' Problem situation where this failure detection service has to deal with. Our focus will be the results obtained in the Byzantine Generals Problem to be incorporated in the failure detection service.

Hence, in this research we also include the implementation of DARX specific agent servers, making DARX fault tolerant functionality (including replication management and failure detection) the responsibility of the AgentScape middleware.

3 Related Works

This chapter considers different works of scholars who tried to show the importance of the Byzantine Generals' problem and the solutions they have proposed. This chapter is organized in four sections: optimizing Byzantine agreement, eventual Byzantine agreement, approximation agreement and fault scalability in Byzantine agreement.

3.1 Optimizing Byzantine Agreement

The Byzantine Generals Problem is a paradigm for distributed processes, some of which may be faulty. It might be reasonable to assume that most of the time, all processes will be correct (i.e., not faulty). An easy way to minimize the number of messages sent in a Byzantine Agreement algorithm would be not to run the algorithm at all if all of the processes are correct. However, the processes show a faulty behavior. Some are faulty and others are not. The distributed application has to be able to identify the correct processes in the set of processes. For this reason we need to come up with an algorithm that identifies the faulty processes and make a decision.

Hadzilacos and Halpern [5] showed that with oral messages, an average of $\left[\frac{n(t+1)}{4}\right]$ messages are necessary for such Byzantine Agreement algorithm, where **n** is the number of processes and **t** is the number of expected faulty processes. With written messages, an average of $\left[\frac{(n+t-1)}{2}\right]$ messages are necessary, which is considerably better for $n \geq 4$.

Instead of considering the number of messages, we could consider the number of message exchanges. An algorithm which tolerates **t** faulty processes must run at least **t+1** message exchanges in worse-case execution; however, if the actual number of faults **f** is less than **t**, then there may be a

way for the algorithm to terminate earlier. This is referred to as *early stopping*, and was described by Dolev *et al.* in [4]. They showed that if only fail-stop failures are allowed, then the minimum number of rounds is min(t+1, f+2) and they presented an algorithm that achieves this bound. Since Byzantine failures were not considered in [4], this result is somewhat weaker, but possibly useful in some systems where crashes are considered to be the only faults.

Another approach is to use non-deterministic (i.e., probabilistic) solution. In [2], Bracha presented a non-deterministic algorithm with an expected number of rounds of $O(\log n)$, as long as $t = \frac{n}{(3+\varepsilon)}$ for oral messages or $t = \frac{n}{(2+\varepsilon)}$ for written messages, where Є is a real value (Є > 0). For any preassigned Є, an approximation agreement algorithm must satisfy the following two conditions:

Agreement: All nonfaulty processes eventually halt with output values that are within Є of each other.

Validity: The value output by each nonfaulty process must be in the range of initial values of the nonfaulty processes.

The key is to use other Byzantine Agreement algorithms which have the following properties: If **t** (number of faulty processes) requires the number of message exchange $O(n)$ then the algorithm is linear in **n**, but if **t** is $O(\sqrt{n})$ then the expected number of rounds is exponential. There are several such algorithms, and the algorithm used in [5] will work with any “plug-in” algorithm with the above properties. If we can convert a system of **n** processes and $O(n)$ faulty processes into one with **m** processes and $O(\sqrt{m})$ faulty processes, then this can solve the new problem in constant time, leaving us only with the time to translate one system into the other.

Bracha [2] showed that if we choose $m = O(n^2)$ and have each of these **m** processes be *compound processes*[1] of $s = O(\log n)$ actual processes (each of which may be in several compound processes), then we can choose the compound processes in such a way that at most $\sqrt{m}$ of them are faulty. A standard t+1 round Byzantine Agreement protocol can be run in each of the **m** compound processes, and the result can be run on the **m** compound processes, with an expected constant number of rounds. This yields a total expected number of rounds of $O(\log n)$.

Castro and Liskov [10] described a new replication algorithm that is able to tolerate Byzantine faults. The paper illustrated how Byzantine fault-tolerant algorithms will be increasingly important in the future because malicious attacks and software errors are increasingly becoming common and can cause faulty nodes to exhibit arbitrary behavior. The paper described the implemented Byzantine-fault-tolerant NFS service using a new algorithm and measured its performance. The algorithm is a form of state machine replication: the service is modeled as a state machine that is replicated across different nodes in a distributed system. Each state machine replica maintains the service state and implements the service operations. The authors of [10] assume |R| = 3f +1, where R is the set of replicas and f is the maximum number of replicas that may be faulty. The algorithm offers both liveness[2] and safety provided at most $\left\lfloor \frac{(n-1)}{3} \right\rfloor$ out of a total of n replicas are simultaneously faulty. It uses one message round trip to execute read-only operations and two to execute read-write operations. The result showed that service is 3% slower than a standard un-replicated NFS.

[1]A process having two or more atomic processes, i.e., process m has s atomic processes. An atomic process is a process having a single thread to execute.
[2] Liveness is the ability of the system to survive a Byzantine failure and gets operational afterwards.

3.2 Eventual Byzantine Agreement

The original Byzantine Generals problem as stated by Lamport *et al.* [8] did not specify that all processes had to decide on a value simultaneously, although their algorithm had that property. In [6], Halpern *et al.* discussed the possibility of an algorithm in which the processes can decide independently and possibly earlier. This is called Eventual Byzantine Agreement, to differentiate it from Simultaneous Byzantine Agreement.

Their results are based on a technique called *continual common knowledge*, which is a variant of common knowledge[1], i.e., the final result of Simultaneous Byzantine Agreement. The key is that at some point in the algorithm, some processes may have enough information to realize that the algorithm will eventually decide on a particular value. It turns out that eventual common knowledge is too weak to solve the problem (because of the way Byzantine Agreement is done), so continual common knowledge (what everyone knows at any point in time) is used instead.

3.3 Approximate Agreement

Byzantine Generals problem can also be solved by considering time duration. The Generals can agree not on the time of attack or retreat but the duration of the time to attack or retreat. Let the Generals agree on the time t to attack with some duration $\pm\varepsilon$ of time.

We can modify the Byzantine Generals problem so that each process inputs a real value rather than a binary value[2], and all processes must eventually decide on real values which are within ε of each other. Notice that if $\varepsilon = 0$ then it is equivalent to the consensus problem with binary values.

[1] Refers to the understanding (knowledge) that every Byzantine General has under a condition where consensus is achieved.
[2] Here values are considered to be real numbers instead of binary values, i.e., 0 or 1.

An algorithm described by Dolev *et al.* [4] works by successive approximation. That is, every round gets closer to the goal with a guaranteed convergence rate[1]. The most interesting part of this algorithm is that it works both in synchronous and asynchronous systems (with $\varepsilon > 0$). The conditions are similar to that of the consensus problem: all correct processes eventually halt with output values within ϵ of each other, and all the values are within the range of input values of all correct processes. For synchronous systems, the convergence rate depends on the ratio between the number of faulty processes and the total number of processes. Convergence is guaranteed when **n > 3t**. For asynchronous systems, convergence is guaranteed when **n > 5t**, where **n** is the number of processes and **t** is the number of expected faulty processes.

The synchronous algorithm basically works as follows. Each process sends its value to every other process. If a faulty process does not send a value, then some default value, say 0, is chosen. Each process then runs the function $\mathbf{f_{t,t}}$ on the **n** real values. Roughly speaking, the function **f** is chosen so as to eliminate the lowest and highest **t** values from the list and take the average of the rest; the faulty processes are thus unable to affect the convergence of the values. To show that it is necessary to have **n > 3t**, assume **n=3t**. If all **t** faulty processes send a very high value to a correct process **p** and a very low value to a correct process **q**, then **p** and **q** will be taking their averages on two completely different sets of numbers, and they will not converge. However, if **n=3t+1**, then **p** and **q** will have one overlapping value at their median, and will slowly converge towards each other.

The asynchronous algorithm is the same as above, except that each process only waits for **n-t** messages, does not choose a default value, and runs the function $\mathbf{f_{2t,t}}$ on the list. It is interesting to

[1] The rate at which the approximation algorithm works and brings the agreement in a finite sequence of rounds.

note that although exact consensus in a totally asynchronous system is impossible, ϵ can be chosen to be arbitrarily small to get as close to consensus as desired.

In the context of tolerating Byzantine faults, the focus has been on building services via replicated state machines using an agreement – based approach as illustrated in the above discussion.

3.4 Fault Scalability in Byzantine Agreement

Abd-El-Malek *et al*. [9] defined a fault – scalable service to be one in which performance degrades gradually, if at all, as more server faults are tolerated. Experience shows that Byzantine fault-tolerant agreement – based approaches are not fault – scalable: their performance drops rapidly as more faults are tolerated because of server – to – server communication and the requirement that all correct servers process every request.

The authors showed the Query/Update (Q/U) protocol as a fault – scalable alternative to different Byzantine agreement approaches. They argued that the Q/U protocol is an efficient, optimistic, quorum-based protocol for building Byzantine fault-tolerant services. Services built with the Q/U protocol are fault-scalable. The Q/U protocol provides an operations-based interface that allows services to be built in a similar manner to replicated state machines. Q/U objects export interfaces comprised of deterministic methods: *queries* that do not modify objects and *updates* that do. Like a replicated state machine, this allows objects to export narrow interfaces that limit how faulty clients may modify objects.

A client's update of an object is conditioned on the object version last queried by the client. An update succeeds only if the object has not been modified since the client’s last query in which case it is retried. Moreover, the Q/U protocol supports *multi-object updates* that atomically updates a set of objects conditioned on their respective object version.

The Q/U protocol operates correctly in an asynchronous model (i.e., no timing assumptions are necessary for safety), tolerates Byzantine faulty clients, and tolerates Byzantine faulty servers. Queries and updates in the Q/U protocol are strictly serializable[1]. The "cost" of achieving these properties with the Q/U protocol, relative to other approaches, is an increase in the number of required servers: the Q/U protocol requires 5t+1 servers to tolerate t Byzantine faulty servers, whereas most agreement-based approaches require 3t+1 servers.

In a Byzantine fault-tolerant quorum-based protocol, only quorums (subsets) of servers process each request and server-to-server communication is generally avoided. In a Byzantine fault-tolerant agreement-based protocol, on the other hand, every server processes each request and performs server-to-server broadcast. As such, these approaches exhibit fundamentally different fault-scalability characteristics.

The expected fault-scalability of quorum and agreement-based is illustrated in Figure 3.1 [9]. Because each server must process every request in an agreement –based protocol, increasing the number of servers cannot increase throughput. Agreement-based protocols require fewer servers than quorum-based protocols for a given degree of fault-tolerance. With quorum-based protocols, as the number of server faults tolerated increases, so does the quorum size. Servers do a similar amount of work per operation regardless of quorum size. However, the Q/U protocol depends on some cryptographic technique (i.e., *authenticators*) whose costs grow with quorum size.

---

[1] If two or more queries and updates are running at the same time the final result looks as though all queries and updates run sequentially in some order.

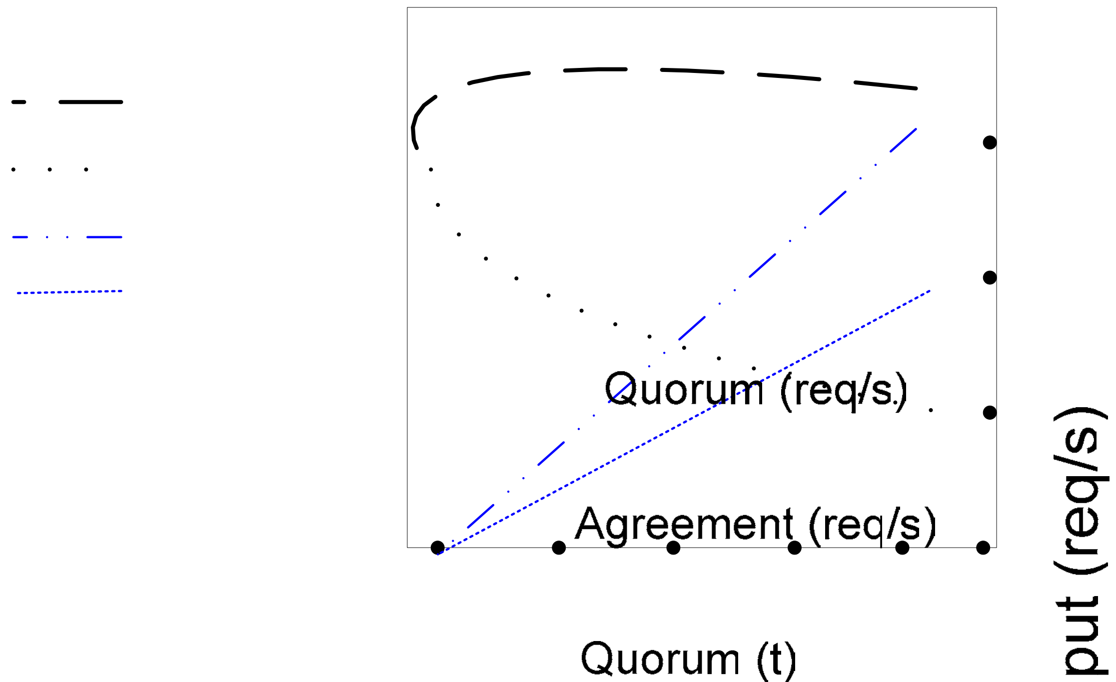


Figure 3. 1: Illustration of fault-scalability.

Figure 3.1 shows as the number of server faults tolerated increases (right axis), the number of servers required (x-axis) by quorum- and agreement-based protocols increases. In theory, throughput (left axis) of quorum-based approaches (for example, based on a threshold quorum) is sustained as more faults are tolerated, whereas agreement-based approaches do not have this property. In practice, throughput of prototype Q/U-based services degrades somewhat as more faults are tolerated.

3.5 Summary

Distributed systems are characterized by partial failures. These partial failures might prohibit the application from performing a system-wide computation and reaching consensus among its components (probably agents). Partial failure might happen due to several reasons, but one of the reasons is the Byzantines failure.

A number of scholars brought the problem of Byzantine Generals for discussion [2, 4, 5, 6, 8, 9]. They have dealt with different algorithms to solve the problem. Most of the algorithms [2, 4, 5, 6, 8] focus on how to reach on agreement and fault tolerance, while [9] discusses the problem of fault tolerance with respect to scalability.

To address the Byzantine Generals problem methods were suggested and used practically. Basically, the problem has been investigated since many years and has been modeled as Byzantine failure. Many techniques have been proposed for solving such problems. However, such techniques do not take into account the computational effort it requires, thus making it unattractive for practical use. These techniques do not address the fault scalability of Byzantine faults thoroughly. Hence in this research we look for better technique to address fault-scalability of Byzantine failures.

These techniques have to be integrated to the new emerging trend in developing systems, i.e., agent-oriented software development. What makes agent-oriented software development different from other ways of development, like object-oriented software development, is that it develops entities known as agents to construct a system. This development way has been adopted to construct cooperative applications, usually known as Multi-Agent Systems (MAS), which can run on a distributed system environment.

4 Design

This chapter contains the design to solve the Byzantine Generals problem that has been proposed. We describe the algorithms to solve the problem in asynchronous environment. The algorithms considered have to be seen from scalability point of view also. Scalability of fault tolerance in Multi-Agent Systems is the issue that is addressed in this work.

The first section in this chapter describes the algorithms used to reach to agreement between the Byzantine Generals with minimum round of information exchange and scalability. The next section describes the relationship between the Byzantine agreement importance in Multi-Agent Systems (MASs) and how it is used. The third section describes the different components that have to be modified and included in the AgentScape/DARX framework to tolerate Byzantine faults in scalable manner. The fourth section mentions the design issues considered regarding the development of this chapter namely safety and liveness of the system. Conclusion of this chapter at the end gives summary of the protocols involved and the design issues considered.

4.1 Byzantine Agreement Algorithm

Previous chapters dealt with the Byzantine Generals problem thoroughly. In this section we deal with the algorithm that can address the Byzantine Generals problem by considering scalability.

Communication between the Generals is considered both in synchronous and asynchronous ways [31, 32]. A system of parallel processes is said to be synchronous if there is a common clock for all processes and asynchronous if each process has its own independent clock. For both cases different approaches have to be followed. One can easily see that byzantine faults in the synchronous environment are easier to be detected and tolerated than asynchronous environment. This is due to the nature of synchronous communication. Synchronous systems negotiate on common clock before

they start to communicate. They synchronize their clocks before transmission, and reset numeric counters for errors, etc. The algorithm developed focuses on asynchronous communication between the Generals.

As shown in the previous chapters, scholars tried to address the Byzantine failure using two ways: agreement based [2, 3, 4, 5, 6, 8, and 10] and quorum based [9, 20, and 21] approaches.

Quorums based systems are tools for increasing the availability and efficiency of replicated data [20]. Guerraoui et al. [21] also argue that quorum based systems are mathematical tools to reason about distributed implementation of shared objects including read/write storage and reaching consensus. In particular, quorum based systems have been used to reason about implementations that tolerate asynchrony and are optimally resilient to process failures. Intuitively, the intersection property guarantees that if a "write" operation is performed at one quorum, and later a "read" operation is performed at another quorum, then there is some server that observes both operations and therefore is able to provide the up-to-date value to the reader. Thus, system-wide consistency can be maintained while allowing any quorum to act on behalf of the entire system. Compared with performing every operation at every server, using quorums reduces the load on servers and increases service availability despite server crashes.

Malkhi *et al.* [20] argued that quorum based systems are traditionally assessed by three measures of quality: load, fault tolerance, and failure probability. The *load* of a quorum based system is a measure of its efficiency: it is the rate at which the busiest server is accessed. The *fault tolerance* of a system is the maximum number of server failures for which there is still guaranteed to be a quorum containing no faulty servers. The *failure probability* is the probability that every quorum contains a faulty server, assuming that servers fail independently with a fixed probability. The fault tolerance of any quorum based system is bounded by half of the number of servers. Moreover, the

failure probability typically increases to 1 when the individual failure probability of servers exceeds $\frac{1}{2}$. Also, there is a trade-off between low load and good fault tolerance, and in fact it is impossible to achieve optimality in both of them simultaneously.

Agreement based protocols require to first agree on a unique, serial ordering of requests; the requests are then executed in that order and replies send to the clients. Quorum-based protocols, on the other hand, are optimistic since they do not require replicas to explicitly run an agreement phase. Typically, quorum-based protocols require higher replication (5f + 1 compared to 3f + 1 for agreement based) but are more efficient since they can complete requests in one to two communication rounds compared to three for agreement-based protocols. However, conflicting write operations – write operations that cause replicas to become inconsistent with each other since they are received by different replicas in different order – causes severe performance degradation of these quorum based protocols.

4.1.1. System Model

A number of realistic assumptions are made in proposing the model. The first assumption is the Byzantine failure model where faulty nodes (replicas or clients) may behave arbitrarily. The second assumption is there is a strong adversary that can coordinate faulty nodes to compromise the replicated service. The third assumption is that the adversary cannot break cryptographic techniques like encryption, and signatures. Our system ensures its safety and liveness properties if at most f replicas are faulty. The final assumption is there is a finite client population. These clients can be faulty.

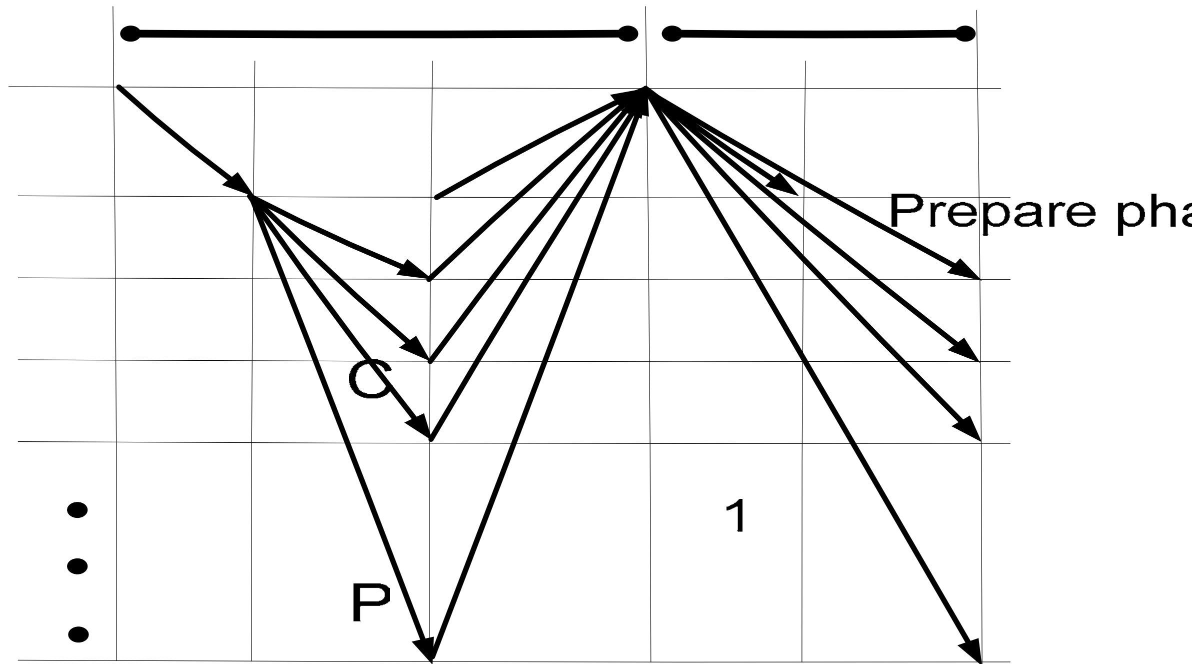
Our system's safety properties hold in any asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, corrupt them, delay them, or deliver them

out of order- thus exhibiting Byzantine behavior. Liveness is ensured only during intervals in which messages sent to correct nodes are processed within some fixed (but potentially unknown) worst case delay from when they are sent.

Our model is depicted in Figure 4.1 and incorporates a client (C), n-1 replicas (numbered as 1, 2, …, n-1) and one primary (P). Two cases were considered based on the behavior of the primary. We approach the communication pattern in two ways:

1. All the replicas and the primary send replies to the client, and then the client is responsible to coordinate the responses as shown in Figure 4.1 (a).
2. All the replicas send their replies to the primary; the primary is responsible to coordinate the responses, and then the primary evaluates the replies and sends the response to the client as shown in Figure 4.1 (b).

Obviously, the former approach is the better approach due to the fact that the load on the primary will be reduced. If the primary is faulty, then we avoid system failure. In the latter case, if the primary is detected to be faulty, we have to give the responsibility of handling consistence between replicas to the client by ordering requests to the replicas; hence, the election sub-protocol has to take effect in this scenario.



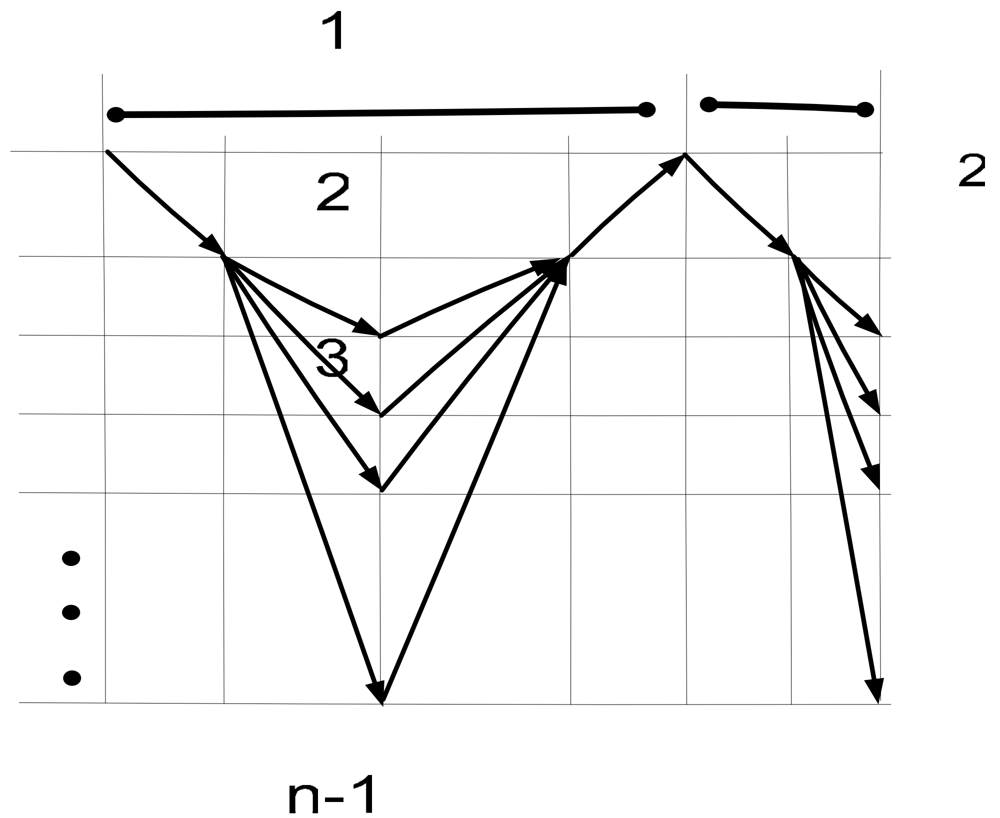


Figure 4. 1: Protocol communication patterns between client and replicas

The model selected here is the first one where the client is responsible to collect responses from the

(a) Client collects respons

primary and the replicas. This choice is obvious for the reasons mentioned when presented

Prepare pha

describing the two approaches. Hence, discussion presented here primarily focus on the model described in Figure 4.1, unless it is mentioned.

Our model (see Figure 4.1(a)), requires 3n one-way message.

The model has two phases: prepare phase and commit phase. A distributed database can be considered an application which makes use of both phases.

Prepare Phase

(1) a client (C) sending a request to the primary (P),

(2) the primary (P) is responsible to send the request to the replicas,

(3a) the replicas and the primary send their responses to the client (Figure 4.1(a)),

or (3b) the replies from the replicas are accepted by the primary at the prepare phase (Figure 4.1 (b)).

(4) the primary sends the replies to the client (Figure 4.1 (b)).

Commit Phase

(5a) the client sends a commit message to all the replicas (Figure 4.1 (a))

(5b) the client initiates the commit phase by sending a message to the primary (P) (Figure 4.1 (b)),

(6) the primary sends a commit message to all the replicas (Figure 4.1 (b))

4.1.1.1. Protocols Developed

The protocol combines state machine replication protocol (like in [10]) and quorum based protocols (like in [9, 20, 21]). It comprises of two sub-protocols:

i. Agreement protocol

ii. Election protocol

The *agreement* protocol focuses on how to achieve a common value from quorums and between quorums. The *election* protocol has to conduct election among the existing replicas in case the current primary fails. According to [34], primary election is a fundamental problem in distributed computing.

In the protocol, clients are responsible to detect replica's inconsistence. A replica might behave arbitrarily. Clients drive replicas to have consistent value by sending single total ordering of requests.

A request completes when a correct client may safely act on the reply to that request. To help a client determine when it is appropriate to act on a reply, the protocol appends log information to the replies received by a client so that the client can judge whether the replies are based on the same ordering of requests. The protocol ensures the following safety condition:

If a request with sequence number n and log l_n completes, then any request that completes with a higher sequence number n' >= n has a log $l_{n'}$ that includes l_n as a prefix.

The election sub-protocol must ensure liveness despite an agreement sub-protocol. We shift work from the agreement sub-protocol to the election sub- protocol by introducing the need to change the primary in case the primary is suspected to be faulty. The protocol ensures the following liveness:

Any request issued by a correct client eventually completes.

The protocol is executed by n replicas, having $[(n-1)/2]$ quorums, and execution is organized into a sequence of locations. Within a location, a single replica is designated as the primary responsible for leading the agreement sub-protocol. Every replica is a member of a single quorum, except the primary which is member of all the $[(n-1)/2]$ quorums. Quorums are created by having at most 2 replicas and a primary as an element. A client receives n responses from the replicas of different quorums. If the client receives 2f responses, then additional response from the primary is the

decision that the client has to consider. The reason to take the response from the primary is due to the fact that the algorithm uses the DARX replication service that calculates the degree of consistency (DOC) for all replicas. The DOC value for the primary is the highest value in the RG. The DOC value histories of the replicas are recorded in the log.

Based on the number of replicas, the number of quorums varies. In Figure 4.2, we show 2 quorums for 5 replicas. For 9 replicas there are 4 quorums, for 11 replicas there are 5 quorums, etc.

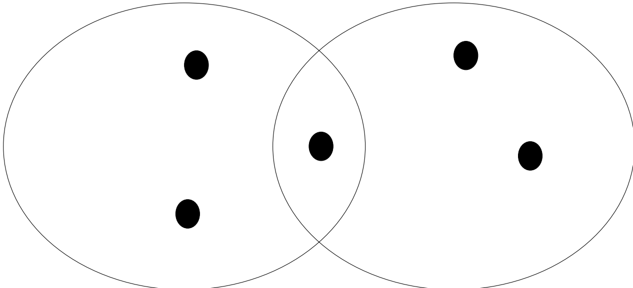


Figure 4. 2: A location with 4 replicas and a primary, shared between the two quorums

Figure 4.1 shows the communication pattern for a single instance of our client-centric agreement sub-protocol. A client sends a request to the primary, the primary forwards the request to the replicas, and the replicas execute the request and send their responses to the client. A request completes at a client. If the client receives 2f + 1, where f refers to faulty replica, mutually-consistent responses from every quorum and $2f_q$ quorums, where f_q refers to faulty quorum, then the client considers the request complete and acts on it. If a sufficient number of replicas suspect that the current primary is faulty, then an election is conducted and a new primary is elected.

With respect to latency, the protocol executes requests in three one-way message delays, which matches the accepted lower bound in the distributed systems literature for agreeing on a client request [29].

4.1.1.2. Maintaining State of a Replica

In this section, we discuss how maintenance of replica state is done. Each replica i maintains a log of the requests it has executed. The log up to and including the request with the highest sequence number covered by requests made is the log information. We say that a request made has sequence number n if n is the highest sequence number of any request in the log information.

A replica constructs a checkpoint every requests interval in order to achieve consistency. To avoid inconsistency among replicas, a replica maintains one *stable checkpoint* and a corresponding *stable application state snapshot*, and it may store up to one *tentative checkpoint* and corresponding *tentative application state snapshot*. The process by which a tentative checkpoint and application state become committed is similar to the one used by earlier Byzantine Fault Tolerance (BFT) protocols [10, 22, 23]. However, to summarize briefly: when a correct replica generates a tentative checkpoint, it sends a signed checkpoint message to all replicas. The message includes the highest sequence number of any request included in the checkpoint and the application snapshot. A correct replica considers the checkpoint and corresponding application snapshot stable when it collects f matching checkpoint messages signed by different replicas.

To bound the size of the log, a replica:

(1) truncates the log before the marked checkpoint and

(2) blocks processing of new requests after processing `2* requests_intervals`
since the last replied request checkpoint.

Finally, each replica maintains a *response cache* containing a copy of the latest request from, and corresponding response to, each client.

i. The Agreement Protocol

Figure 4.1 illustrates the basic flow of the agreement sub-protocol in a location. Because replicas execute requests proposed by the primary without communicating with other replicas, the key challenge is ensuring that clients only act upon replies. The protocol is constructed so that a request completes at a client when the client receives $2f_q$ (faulty quorums), each of the quorums having 2f (faulty replicas) + 1 matching responses.

To describe how the system deals with this and other challenging, but standard, issues—lost messages, faulty primary, faulty clients, etc.—we follow a request through the system, defining the rules a server uses to process each message. The labeled numbers in Figure 4.1 correspond to numbers in the text identifying major steps in the protocol and Table 4.1 summarizes the labels given to fields in messages.

Table 4. 1: **Definition of terms used in messages**.

Label	**Definition**
c	Client Code
l_n	Log through sequence number n
msg	Message containing client request
max_n	Maximum sequence number accepted by replica
n	Sequence number
req	Requested task by the client
res	Response to a client
t	Timestamp to a request by the client
loc	Location number

1. Client sends request to the primary.

A client, c, requests a request, req, to be performed by the replicated service by sending a message $<request, req, t, c>_c$ signed by client, c, to the replica it believes to be the primary (i.e., the primary for the last response the client received).

If the client guesses the wrong primary, the retransmission mechanisms are discussed later to forward the request to the current primary. The client's timestamp t is included to ensure exactly-once semantics of execution of requests.

2. Primary receives request, assigns sequence number, and forwards ordered request to replicas.

When the primary, p, receives message msg <request, req, t, c> $_c$ signed from client, c, the primary assigns a sequence number, n, in location, loc, to the request and relays a message <<redirect_request, loc, n>$_p$, msg> to the backup replicas. The primary only takes the above actions if $t > t_c$ where t_c is the highest timestamp previously received from c.

3a. Replica receives request, executes it, and responds to the client.

Upon receipt of a message <<redirect_request, loc, n>$_p$, msg> from the primary, p, replica, i, accepts the redirected request. If msg is a request message, then $n = max_n + 1$ where max_n is the largest sequence number in i's log. Upon accepting the message, replica, i, appends the redirected request to its log, executes the request using the current application state to produce a reply res, and sends a message <<response, loc, n, q_k, l_n, c, t>$_i$, i, res, <redirect_request, loc, n>$_p$ >, to client c where q_k is the quorum number.

A replica may only accept and execute requests in sequence-number order, but message loss or a faulty primary can introduce missing sequence numbers. Replica, i, discards the redirect_request message if $n <= max_n$. If $n > max_n +1$, then i discards the message, sends a message <missing_sequence_number, loc, max_n +1, n, i> signed message to the primary, and starts a timer. Upon receipt of a message <missing_sequence_number, loc, o, n, i> $_i$ from replica i, the primary p sends a <<redirect_request, loc, n', $l_{n'}$,>$_p$, msg'> $_p$ to i for each request msg' that p ordered in o <= n' <= n in the current location; the primary ignores missing sequence number requests from other locations. If i receives the valid redirected request messages needed to correct missing sequence numbers, it cancels the timer. Otherwise, the replica broadcasts the missing_sequence_number message to all other replicas and initiates election of new primary when the timer expires. Any

replica j that receives a missing_sequence_number message from i sends the corresponding redirected request message, if it has received one. If, in the process of correcting missing sequence numbers in the replica sequence, replica i receives conflicting redirected request messages then the conflicting messages form a proof of inconsistency.

5a. Client sends commit message to the replicas.

Client, c, sends commit message msg $\langle commit, req, t, c, loc, n \rangle_c$ by assigning a sequence number, n, in location, loc, to the backup replicas. Replicas have to commit the request and record the information in the log.

ii. The Election Protocol

The election sub-protocol elects a new primary and guarantee that it will not introduce any changes in a log that has already completed at a correct client. To maintain this safety property, traditional election sub - protocols [10, 22, 23] require a correct replica that commits to an election to stop accepting messages other than checkpoint, and election messages. Also, to prevent faulty replicas from disrupting the system, an election sub-protocol should never remove a primary unless at least one correct replica performs the election. Hence, a correct replica traditionally commits to an election if either:

(a) It observes the primary to be faulty or

(b) It has a proof that f_q quorums, each having f+1 replicas, have performed an election request.

On performing an election a correct replica sends a signed election message that includes the new primary, and the sequence number of the replica's latest stable checkpoint—collected by the replica. The traditional election of a new primary completes when the new primary, using 2f + 1 election messages from distinct replicas, computes the history of requests that all correct replicas must adopt

to have new primary. The new primary includes this log in a signed new primary message that it broadcasts to all replicas.

4.2 Byzantine Agreement in Multi-Agent System

Overeinder *et al.* [1] demonstrated that a large scale multi-agent system is characterized to be physically distributed, found in a dynamic environment, and considered to be open. The complex nature of multi-agent systems makes them vulnerable to faults and system failures. Failure can happen on several parts of the system. It can be software (like bug, deadlocks, etc…) or hardware (like network links, machine, etc).

There are four essential characteristics of multi-agent systems as described in [24]: a MAS is composed of autonomous software agents, a MAS has no single point of control, a MAS interacts with a dynamic environment, and the agents within a MAS are social (agents interact with each other).

The main sources of complexity in MASs are the agent's environment, the nature of the interaction between the agents, the environment, and the nature of the task the agent is performing. Additional complexity is introduced by interactions between the agents within the system and by the distributed nature of MASs.

MASs basically consist of a number of locations and agents. Locations are (logical) abstractions for (physical) hosts in a computer network. The network of locations serves as a unique and homogeneous platform, while the underlying network of hosts may be heterogeneous and widely distributed. Locations therefore have to guarantee independence from the underlying hardware and software.

Agents are active, autonomous software objects that reside (and are processed) on locations. They can communicate with other agents either locally inside one location or globally with agents on

other locations. Multi- Agents furthermore can migrate from one location to another. Mechanisms for the communication between agents and for the migration of agents have to be provided by the Multi- Agent System.

4.3 Byzantine Agreement in AgentScape/DARX Integrated Framework

Figure 2.5, in chapter 2, shows how agents, objects and locations are organized in the AgentScape middleware. Figure 2.6, in chapter 2 again, shows how a replication group in DARX framework is managed. The aim of this work is to integrate these two systems and make them tolerate Byzantine failure by applying the protocol designed and presented in section 4.1.

DARX uses a failure detector to keep track of dynamic list of replicas participating in the application. Failure detectors are organized in groups [1]. These failure detectors exchange heartbeats and maintain a list of processes which are suspected of having crashed. Heartbeats are subsequently released by the replica as a proof of its continued presence in the group.

Every DARX entity integrates an independent thread which acts as a failure detector. It sends heartbeats to replicas and collects the incoming heartbeats.

Using additional entity, i.e, failure detector, and sending heartbeats now and then undermines the performance of a multi-agent system. Broadcasting failure detector heartbeats several times in a row introduces high network traffic. Thus delay, collision, and corruption of message may occur – that produce latency, low throughput and low performance of the application.

This thesis work incorporates the way of detecting failures of replicas using the algorithm presented in section 4.1. Clients are used to detect failures of replicas in multi-agent systems. Replica to replica broadcast messages will be avoided to reach to agreement. Such kind of replica to replica

broadcast is well known in traditional Byzantine Fault Tolerance (BFT) systems, which are not fault-scalable. We define a fault-scalable service to be one in which performance degrades gradually, if at all, as more server faults are tolerated. Experience shows that Byzantine fault-tolerant agreement-based approaches are not fault-scalable: their performance drops rapidly as more faults are tolerated because of server-to-server broadcast communication and the requirement that all correct servers process every request.

A replica obtains a stable application state snapshot from the observation service that acquires its information from the AgentScape management system. Monitoring information such as load of host, date of creation of replicas, communication latencies to other replicas in the group, state of database locks, etc. are some to mention that are offered by the observation service.

DARX handles replication groups (RGs). Each of these groups consists of software entities (replicas) which represent the same agent. Maintenance of consistency in RG is done by either active, passive, or mixed strategies that has a potential to lead to byzantine failure.

Replicas are listed in the replication policy according to their potential of leadership. Hence, the first replica from the list can be considered to be the primary of the group.

Figure 4.3 shows the proposal to the modification on the DarxMessage entity that has to be incorporated to the replication management scheme of DARX in Figure 2.5 of Chapter 2. The modification is shown in bold in the DarxMessage entity in Figure 4.3..

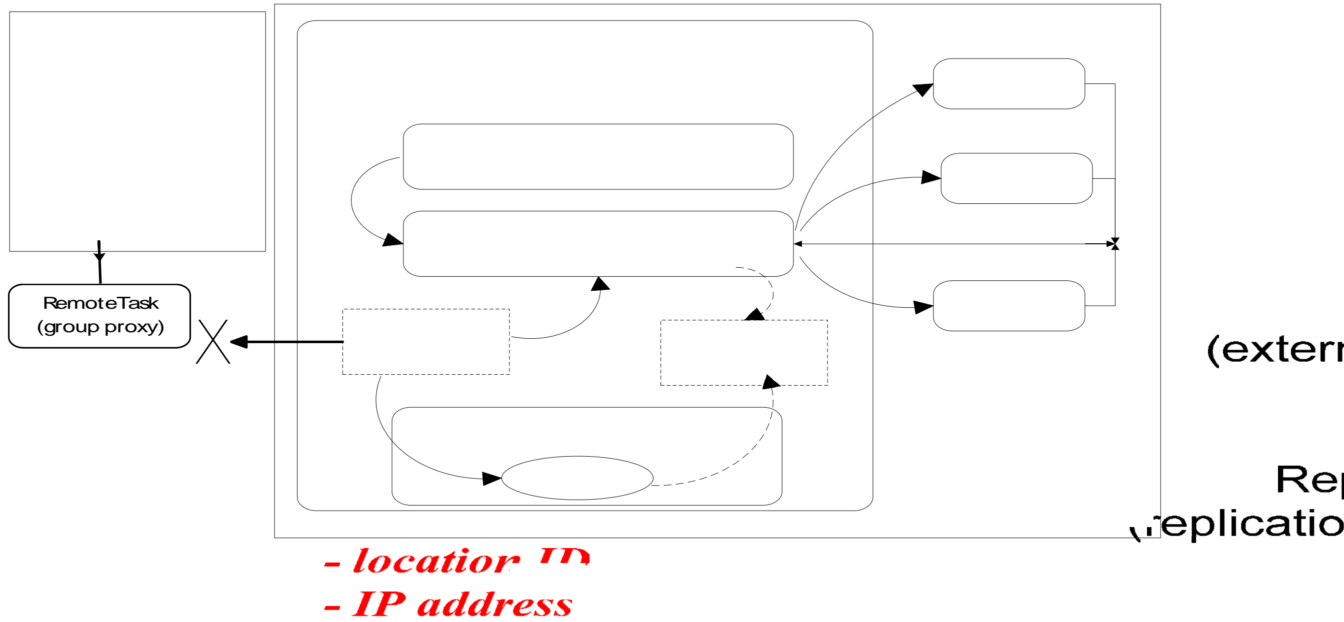


Figure 4. 3: Modified Replication Management Scheme

A primary in DARX is elected within the RG for consistency maintenance and updating version of the replication policy. Its objective is to adapt the replication policy to the criticity of the associated agent as a function of the characteristics of its context – the information obtained through observation. In our model, the primary has to do the task that already DARX framework assigned to and the activities they are listed in section 4.1.

4.4 Design Issues

When we start to design some model to solve a given problem there should be a goal against which results have to be compared to. This section address the design issues considered while designing the proposed solution.

4.4.1. Safety

We first show that the agreement sub-protocol is safe within a single location.

The proof proceeds in two cases. First, we show that no two requests complete with the same sequence number n. Second, we show that l_n is a prefix of $l_{n'}$ for $n < n'$ and completed requests req and req'.

Case 1: A request completes at a client when the client receives $2f_q + 1$ quorums, each of the quorums having 3f + 1 matching responses in phase 2. If a request completes in phase 1 with sequence number n, then no other request can complete with sequence number n because correct replicas send only one response for a given sequence number. Similarly, if a request completes with sequence number n in phase 2, no other request can complete since correct replicas only send one commit message for sequence number n.

Case 2: For any two requests req and req' that complete with sequence numbers n and n' and logs l_n and $l_{n'}$ respectively, there are at least 2f + 1 quroums, each having 3f +1 replicas that ordered each request. Because there are only 3f + 1 replicas in total, at least one correct replica ordered both req and req'. If $n < n'$, it follows that l_n is a prefix of $l_{n'}$.

4.4.2. Liveness

The protocol guarantees liveness only during periods of synchrony. To show that a request issued by a correct client eventually completes, we first show that if the primary is correct when a correct client issues the request, then the request completes. We then show that if a request from a correct client does not complete during the current location, then an election of a new primary will occur.

Case 1: If the client and the primary are correct, then protocol steps 1 through 3 ensure that the client receives response messages from all correct replicas. If the client receives 2f + 1 matching response messages from quorums, the request completes—and so does the proof.

Case 2: Assume the request from correct client, c, does not complete. Client, c, resends the request message to all replicas when the request has not completed for a sufficiently long time. A correct replica, upon receiving the retransmitted request from client c, contacts the primary for the corresponding order-req message. Any correct replica that does not receive the order-req message from the primary initiates the election sub-protocol by sending a new_primary message to all other replicas. Either at least one correct replica receives at least f + 1 new_primary messages, or no correct replica receives at least f + 1 new_primary messages. All correct replicas that did not receive the order-req message from the primary receive it from another replica. After receiving an order-req message, a correct replica sends a response to client c. Because all correct replicas send a response message to c, c is guaranteed to receive at least 2f + 1 such messages. If however, c does not receive 2f +1 matching response messages, then c is able to form a proof of inconsistency message: c relays this message to the replicas which in turn initiate and commit to an election, completing the proof.

4.5 Summary

We started how to deal with Byzantine failures using minimum round of information between a client, the primary, and replicas. The algorithm has two sub-protocols: agreement and election sub-protocols. The *agreement* protocol focuses on how to achieve common value from quorums and between quorums. The *election* protocol has to conduct election among the existing replicas in case the current primary fails. This algorithm combines the agreement-based and quorum-based techniques.

The approach used avoids the traditional server-to-server broadcast of messages to process requests. This obviously solves the scalability problem of byzantine fault tolerance. The traditional BFT protocols suffer form such fault scalability problem.

We have also mentioned how our protocol can be incorporated in the AgentScape/DARX integrated framework to handle byzantine failures in multi-agent systems. We have shown where the failure detector of DARX fails to detect byzantine failure and how it introduces performance loss due to its repeatedly sent heartbeat messages to check the presence to a replica. Modification of the replication management scheme were shown and discussed by introducing new entries to the DARX message format.

5 Implementation and Experiment

This chapter discusses the implementation and testing of our protocol. The pseudo-codes for the algorithms shown in the following pages do not contain all the details of the running algorithms. They show only the major functionalities incorporated in the protocols. Some methods are left only on their declaration level to obtain simplicity of this document.

The first section of this chapter shows the pseudo-codes for client-side, primary-side, and replica-side. The next section shows the results obtained from tests conducted from the implementation of the proposed protocol. Conclusion on the implementation and experiment issues are presented in the last section.

5.1 Implementation Issues

As discussed in the previous chapter, the proposed design has two major protocols, namely the agreement and election protocols. This section illustrates the pseudo-codes used to implement the protocols. To implement the protocols, we have to deal with the client, primary, and replicas. The pseudo-codes are described for client-side, primary-side, and replica-side for the agreement protocol in this section.

The implementation is developed using the integration of AgentScape and DARX. The integration does not contain the full version of DARX. This is due to the incomplete development of the DARX prototype. Thus, we were forced to study how DARX is designed and implement important services to the proposed algorithm.

We developed the naming and localization, system observation, replication, and interfacing services of DARX. These services are directly and indirectly in the proposed system.

Client-side

Pseudo-code for client-side functions is given in Figure 5.1. The pseudo-code includes client initialization, from line 1000 to 1003, and collects replicas information from the AgentHistory array. Clients read replica histories from servers. Clients store replica histories from servers in an array called the agent history (AgentHistory). The agent history is indexed by server, for example, ReplicaHistory[r] is equal to the last replica history returned from server r (i.e., r.ReplicaHistory).

Lines 1100 to 1102 show the client request pseudo-code. These lines describe how request is done by a client after being initialized. After requisition to the replicas, the client gathers response from the primary and replicas, lines 1200 to 1211. Starting from line 1300 through 1303, pseudo-code for making decision by the client is listed down.

```
/*Client Initialization*/
1000: ClientInitialize():
1001: for each (r ∈ U) do        /*U – the set of replicas running*/
1002:  AgentHistory[r],ReplicaHistory := {(0,0)}
1003: end for

/*Client Request*/
1100: ClientRequest():
1101: Req:=(msg,t,primaryID)
1102: clientRequest(primaryID, req, ClientID, IPaddress,portno, locationID, t);

/*Client Gather Responses/
1200: ClientGatherResponses():
1201: Number_of_replicas = replicationManager(cirticity);
1202: Responses_list[number_of_replicas];
1203: Quorum_list[number_of_replics]
1204:  i:=0
1205:  Do
1206:       Responses_list[i]=get_response(replicaID,
1207:               IPaddress, portno, groupID, locationID)
1208:        Quorum_list[i] =
1209:  get_quorum(quorumID,replicaID, locationID)
1210:        i:= i+1
1211:  while( i < number_of_replicas)

/*Client Make Decision*/
1300: ClientMakeDecision():
1301: Commit := make_decision(responses_list, groupID,
1302:                            quorum_list)
1303: ClientCommit( primaryID, commit, t, clientID,groupID, IPaddress, portno,
locationID)
```

Figure 5. 1: Pseudo-code for client side

Primary Side

Figure 5.2 lists the pseudo-code for the primary side. Every replication group (RG) must have a unique primary. Its role is to assess the replication policy; to make sure every replica in the group has a consistent knowledge of the policy, and most importantly to accept messages (request and commit) from client and forward to replicas.

Originally, the leadership is given to the initial replica: the first to be created within the RG. However, this may not be the best choice all along the computation: the host supporting this replica may be experiencing CPU or network overloads, or its mean time between failures may be exceedingly high. Therefore, the primary has the possibility to appoint another replica as the new

RG primary. If the primary happens to fail and be replaced, a new election is initiated by the naming service through a failure notification to the remaining replicas.

The pseudo-code listed in Figure 5.2 shows primary initialization from line 2000 to line 2003. Lines 2100 through 2110 list how a primary process request sent from the client, and forwards it to the rest of replicas in its RG. Commit message sent from a client is accepted by the primary and forward it to the replicas in the RG, as shown from line 2200 to line 2205.

```
/*Primary Initialization*/
2000: Primary_initialize():
2001: ReplicsInfo(replicaID, groupID, IPaddress,
2002:                      portno, locationID)
2003: r.ReplicaHistory := {(0, 0)}

/*Primary Process Request*/
2000: Primary_Process_Request():
2101:  n:= 0 //sequence number
2102:        AcceptRequest:= ClientRequest(primaryID, req,
2103:        t, clientID, groupID, IPaddress, portno,
2104:        locationID)
2105: j = 0
2106: Do
2107:  ForwardRequest(req, replicaID, groupID,
2108:               IPaddress, portno, clientID, locationID)
2109:  j:=j+1
2110: While(j < number_of_replicas)

/*Primary Sends Commit Message*/
2200:  i:=0
2201:  Do
2202:        ForwardCommit(commit, replicaID, groupID, 2203:
             IPaddress, portno, locationID)
2204:        i:= i+1
2205:  while( i < number_of_replicas)
```

Figure 5. 2: Pseudo-code for primary side

Replica Side

A replica can be added to or removed from a group during the course of the computation. For example, if a replica crashes, a new replica can be added to maintain the level of reliability within the group; or if the criticity of the associated agent decreases, a replica has to be removed. Replicas

may come and go in an RG, for their existence depends entirely on the dynamic characteristics of the application and of the underlying distributed system.

Figure 5.3 lists the pseudo-code for the replica side. Lines 3000 makes a call to ReplicasInfo function to initialize the replica. Lines 3100 to 3106 show how a process is handled. An active replica will try processing a direct request – a request that is forwarded by the primary, shown in lines from 3200 to 3209.

```
/*Replica Initialization*/
Replica_initialize():
3000: ReplicsInfo(replicaID, groupID, IPaddress, portno, locationID);

/*Process Request*/
3100: Process_Request();
3101: n:= 0; //sequence number
3102: AcceptRequest:= ClientRequest(replicaID, req, t, clientID,
locationID);
3103: if(replicaID=primaryID)
3104:  Primary_Process_Request();
3105: Else
3106:  Replica_Process_Request();

/*Replica Process Request*/
3200: Replica_Process_Request():
3201: n:= getSequenceNumber(msg, clientID);
3202: If(n <= max(SequenceNumberValue)
3203:  AbortRequest(msg)
3204: Else
3205:  If(n > max(SequenceNumberValue)+1)
3206: MissingSequenceNumber(max(SequenceNumberValue) + 1, n, replicaID,
primaryID)
3207:  Else
3208:        ExecureRequest(req)
3209: SendResponse(response, replicaID, quorumID, t, groupID, IPaddress,
portno, locationID)
```

Figure 5. 3: Pseudo-code for replica side

5.2 Experiment

The purpose of this experiment is to test one of the replicant group configuration, to show that replication improves the system failure rate due to Byzantine failure and fault scalability of the proposed system.

The experiment is conducted on AgentScape/DARX integrated environment. AgentScape 9.0 Beta version is used. Among the different components of DARX: naming and localization, system observation, replication, and interfacing services have been developed and integrated with AgentScape 9.0 Beta version open source middleware.

The experiment, presented in this section, was carried out on a machine with Intel ® Pentium ® 4 CPU at 2.8 GHz and 256Mb of RAM.

To carry out the experiment, we have made use of the StartUpGUI of AgentScape, as shown in Figure 5.4, after establishing a lookup server by specifying the IP address and port number on which a location has to exist on. The location name and location address has to be specified to startup a location.

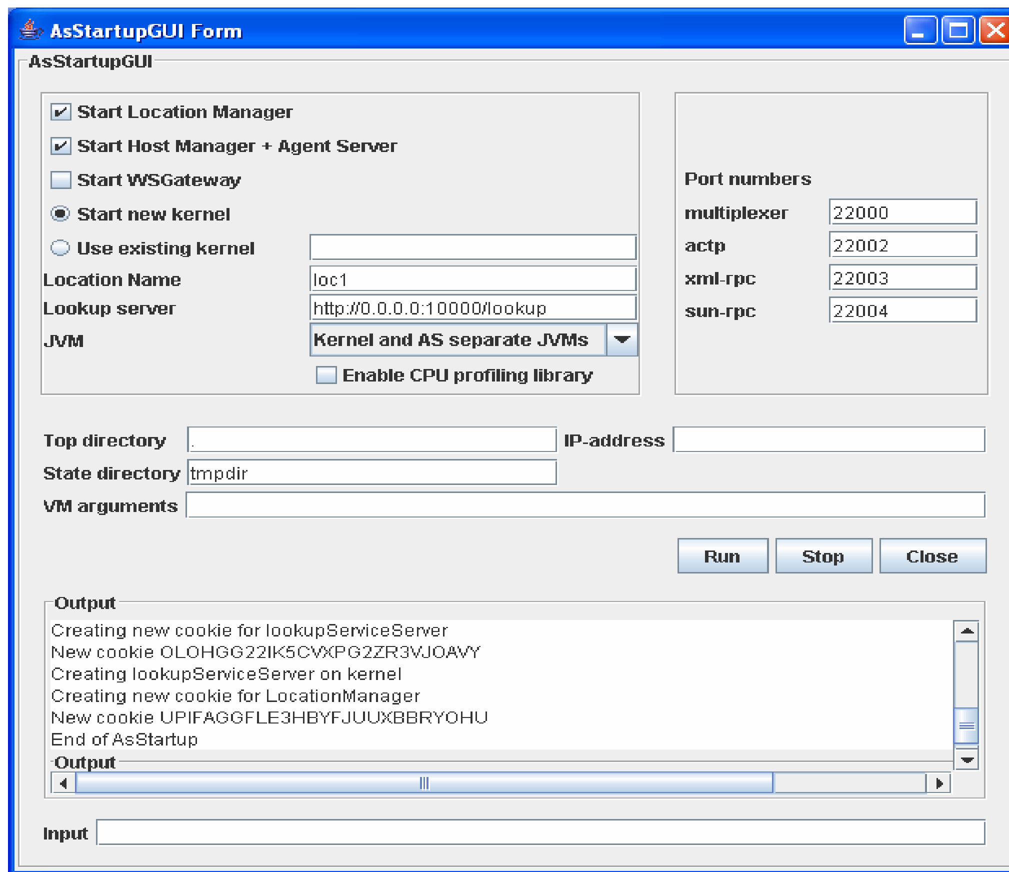


Figure 5. 4: StartUpGUI to launch a location.

A location can host a number of agents. Figure 5.4 shows a location *loc1* being executed at an address 0.0.0.0 and port number 10000, i.e., the lookup server URL.

Hence, to run the agents one has to make use of InjectorGUI interface shown in Figure 5.5 by specifying the look up server URL, selecting the agent to be launched, and selecting the location from the location frame; agents are launched and join the location. Based on the criticity value of an agent a replication is done using the replication service component of DARX.

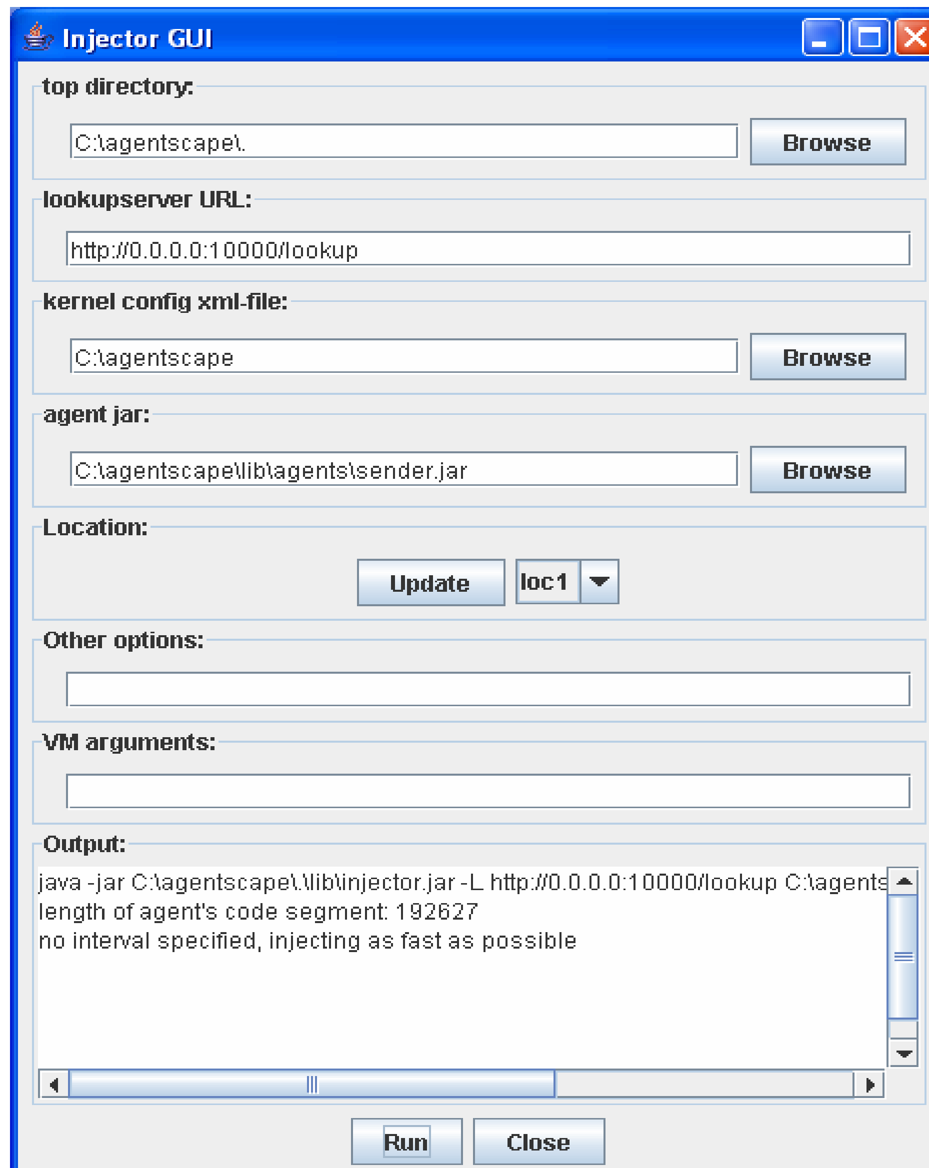


Figure 5. 5: InjectorGUI used to inject an agent to a location (*loc1*)

Currently, running agents on specific locations can be viewed using the AsAdminGUI. Figure 5.6 shows an instance where agents are running at location *loc1*. In Figure 5.6, the Agent list table shows agent id, location address at the Host field, location name, and the identifier of the exception handler for a specific agen. Lists of location can be viewed on a given server with their corresponding running replicas. The AsAdminGUI can also be used to terminate a running replica.

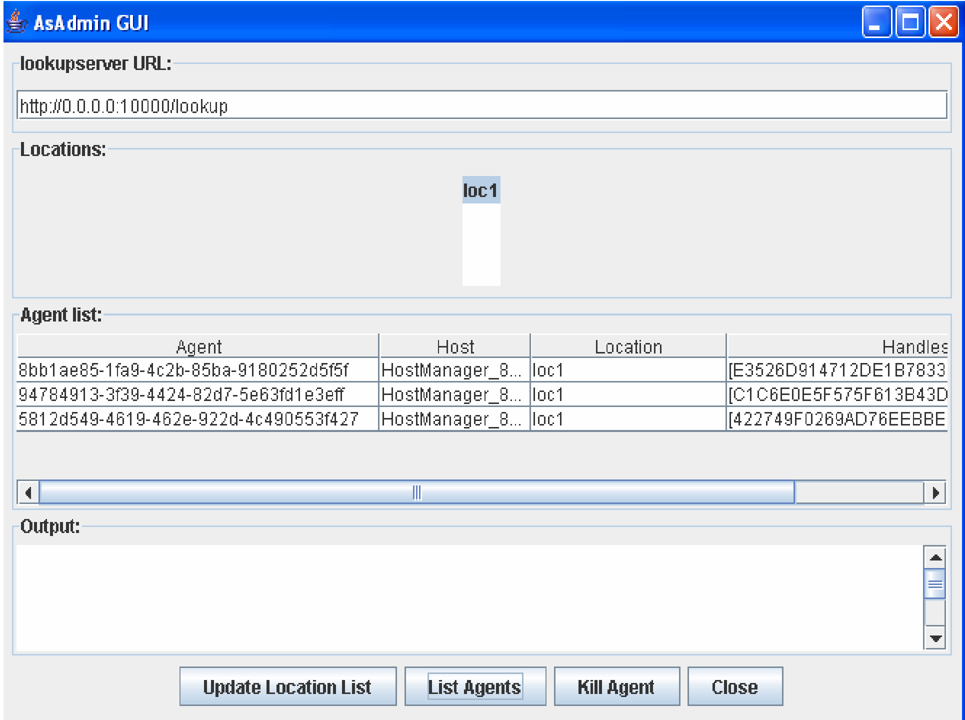


Figure 5. 6: AsAdminGUI used to show replicas running in a given location (*loc1*)

The multi – agent system implemented, for the purpose of experiment, is an information query system consisting of a client agent, which presents a request, a primary which is able to collect data and respond to queries regarding that data. Agent replication is introduced into the system by replicating the primary type agents. The group proxy is used to handle the communication in the RG, as shown in Figure 4.3. The group proxy passes the requests that it receives from the primary to all members of its RG.

To simulate Byzantine faults, replicas have a built-in Byzantine fault. Each time a replica receives a request it will return an erroneous result. A number of replicas have been designed to show Byzantine behavior for this particular experiment.

The experiment consists of running the system for a request, varying the number of replicas in the RG. For each run, the mean request response time and the system failure rate is calculated.

5.2.1. Results

The system was run with different numbers of replicas, n, in the agent RG, having values 3, 5, 7, 9, and 11. These specific values are chosen according to the quorum formation cardinality. Since every quorum has the primary as a member of the set, every quorum will have two other replicas as an element of the quorum. Results showing mean request response time versus size of RG in Figure 5.7. The mean response time increases linearly as the size of the RG increases.

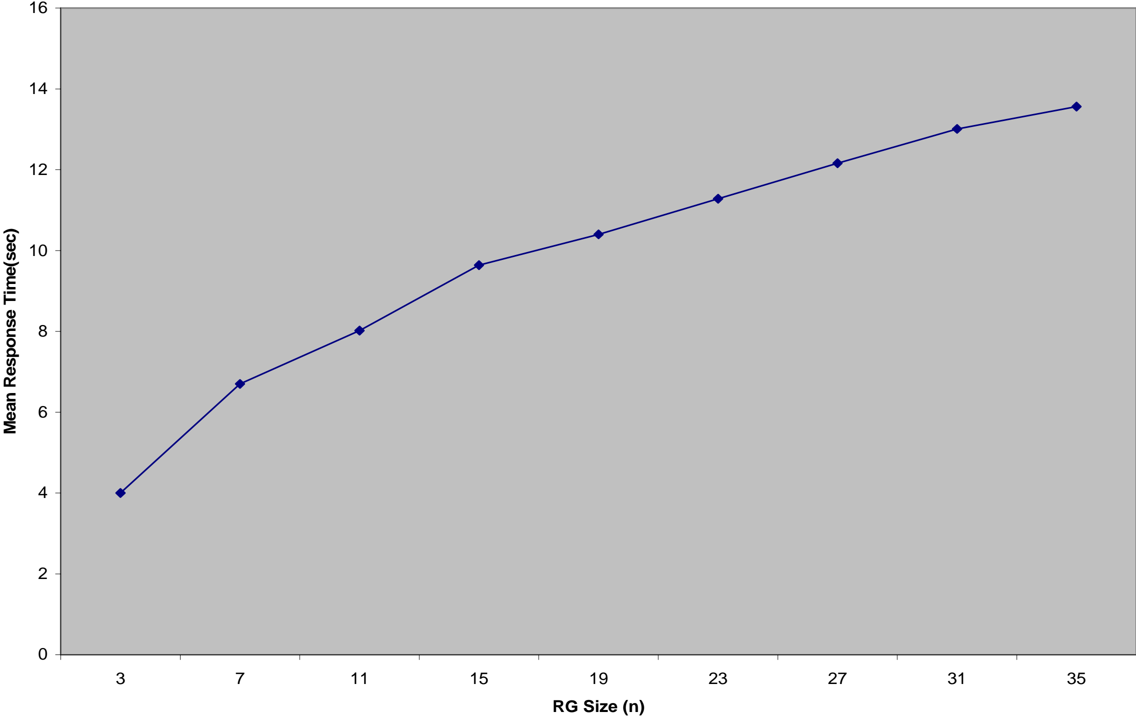


Figure 5. 7: Mean response time vs. RG size

Scalability is the other issue to be addressed in the fault tolerance support system. In Figure 5.8, we have presented how our protocols behave as f, the number of tolerated Byzantine faults, increase. As the number of faults tolerated increases, clients require more time to perform decision making. The protocol is robust to increasing the number of tolerated Byzantine faults and continues to provide significantly higher throughput. We believe that the protocol has two advantages over Q/U:

1. Fewer replicas: Q/U requires 5f + 1 replicas, while the protocol needs 2f + 1 replicas
2. Improved throughput

With respect to fault scalability, the throughput that depends on f, degrades as slowly or more slowly in the protocol compared to Q/U protocol as shown in Figure 5.8.

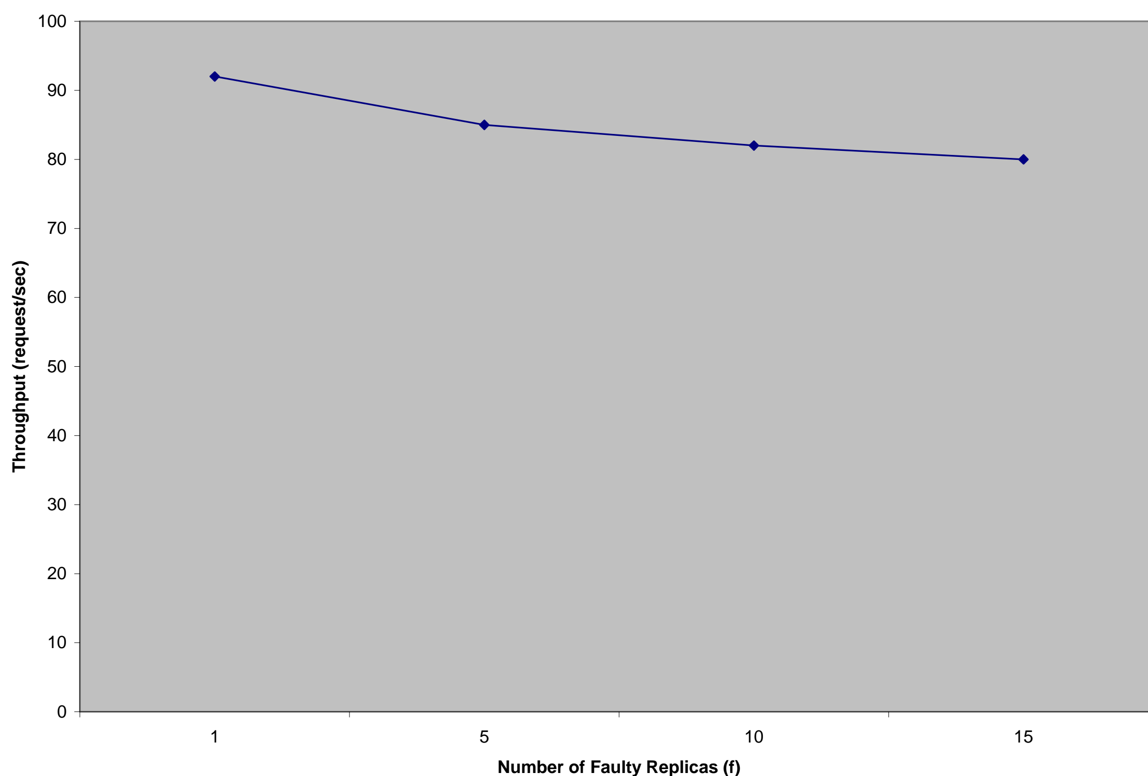


Figure 5. 8: Throughput (request/sec) Vs. Number of faulty replicas

5.3 Summary

The implementation issue covers the pseudo-codes for the client, primary and replica. The pseudo-codes do not contain the entire list to avoid complexity. Major components and method calls are shown and described.

AgentScape and DARX are the tools used in the implementation of the protocol. AgentScape is a middleware layer that supports multi-agent system support environment. AgentScape middleware is designed to provide a minimal but sufficient support for agent applications, and to be adaptive or reconfigurable such that it can be tailored to a specific application (class) or operating system/hardware platform.

DARX is a framework designed to support the development of fault-tolerant applications built upon multi-agent systems. The integration of AgentScape and DARX is used in our experiment to test the fault tolerance in scalable agent support systems. Performance evaluation through experimentation was done to assess the scalability and performance of the fault tolerance support system. The results ware shown using charts. The protocol is robust to increasing the number of tolerated Byzantine faults and continues to provide significantly higher throughput.

6 Conclusion and Future Works

A distributed system is a system with components that must work together where those components are distributed over a private or public network. A reliable distributed system should provide services even when partial failure is exhibited. A partial failure may happen when some components in a distributed system fail. A partial failure may be observed due to different failure types: like crash failure, omission failure, timing failure, response failure and arbitrary (Byzantine) failure.

A failed component (process) may show a behavior that is often overlooked, sending conflicting information to different parts of the system. This behavior is referred as Byzantine Generals problem. Note that for a system to exhibit a Byzantine failure there must be a system-level requirement for consensus.

Business applications, like air traffic controllers, stock exchange systems, etc., are now being developed as distributed systems. This is because the need to give service to the customer has been facilitated by the networking technology available. Such distributed system applications, currently, are being developed by agent-oriented software development, which is a shift from object oriented software development. A distributed application might be composed of multiple agents that are autonomous and communicate with each other to perform a task.

The focus of this thesis is to reduce computational effort exerted for fault tolerance in scalable agent support systems. The fault tolerance, basically, emphasized is the Byzantine faults as mentioned in the introductory chapter.

Many researches show different techniques to address this problem. Among the techniques applied, replication of the failed agent (process) has been found the better way to tolerate Byzantine faults.

Since distributed systems run on a networked environment having heterogenous platforms, we need to hide this heterogeneity by making use of a middleware. Different middleware systems have been developed. We have made use of the AgentScape middleware that has been used with many distributed systems. Also, additional Java framework, known as Dynamic Agent Replication eXtensible (DARX), which is designed to support the development of fault-tolerant applications built upon multi-agent systems has been used. DARX has a failure detection algorithm. But, it does not consider the Byzantine failure.

To address the Byzantine failure two approaches has been used: the agreement protocol and the quorum protocol. The quorum protocol has been proven to be fault scalable and efficient. The protocol developed in this work has used the combination of the agreement based and quorum based approaches for the agreement protocol. The agreement sub-protocol focuses on how to achieve a common value from quorums and between quorums. The election sub-protocol has to conduct election among the existing replicas in case the current primary fails.

The protocol developed avoids the traditional server-to-server broadcast of messages to process requests. This obviously solves the scalability problem of Byzantine fault tolerance.

The protocol is executed by n replicas, having $[(n-1)/2]$ quorums. Every replica is a member of a single quorum, except the primary which is member of all the $[(n-1)/2]$ quorums. Every quorum has 3 members: 2 replicas and a primary as an element. A client receives n responses from the replicas of $[(n-1)/2]$ quorums. If the client receives 2f responses, then response from the primary is the decision that the client has to consider. The reason to take the response from the primary is due to the fact that our algorithm uses the DARX replication service that calculates the degree of consistency (DOC) for all replicas. The DOC value for the primary is the highest value in the RG.

With respect to latency, the protocol executes requests in three one-way message delays, which is acceptable for agreement on a client request. The protocol is robust to increasing the number of tolerated Byzantine faults and continues to provide significantly higher throughput, achieving scalability feature of the fault tolerant multi-agent support system.

The experiment shows the protocol is better than the Q/U protocol in at least two ways: it requires fewer replicas and it scales better.

This work can be extended by incorporating the re-election of the primary in case the primary of an RG is suspected of failure. This re-election algorithm ensures the delegation of a new primary in the RG. Additionally, this work can have a better efficiency by considering the degree of consistency (DOC) value of each replica. Making the client consider the response along the DOC value, efficiency can be improved.

6.1 Future Work

The future work for this thesis has to focus on how to collect an orphan agent. To demonstrate what an orphan agent is and how an agent becomes orphan, refer to the following paragraphs for further description.

If agents explicitly refer to each other, it becomes relatively easy to impose a (possibly hierarchical) structure by which one agent is responsible for managing other agents, notably its siblings [25]. Management in this context generally refers to life cycle management: creating and destroying agents. However, what happens to an agent that is no longer referenced by other agents?

In many ways, this problem is similar to garbage collection in distributed systems, a notoriously hard problem to solve when there are many passive objects wondering around. Considering that agents act autonomously, the situation becomes somewhat different. For example, an anonymous

agent that is actively collecting information should most likely be allowed to survive. However, it is probably not acceptable to allow such an agent to live forever in a more or less anonymous way.

Every agent must have a lifetime to determine the period of its existence. Different agents have different lifetimes. Some static agents have long lifetime. When its resident host machine does not shut down, it always exists. However, some agents have short life. It is necessary to know; a dispatched agent without lifetime could become an “agent orphan” roaming in a network after it has done its mission. This will cause network problem.

Orphan detection in an agent system is very important to handle these orphan agents. A running agent uses resources which are valuable to both users and the system. Since there are limited available resources, these orphan agents have to be removed from the network. So if the user does not need the results of a distributed computation in progress anymore, s/he wants to be able to terminate the computation to minimize the resulting shortage of resources. With an orphan detection mechanism the user simply declares the agents to be terminated as orphans. Orphan detection guarantees that the now useless agents can be determined by the system and terminated, thus freeing the resources they have bound. Additional protocol is needed to minimize the orphan agents that are roaming around the network by enforcing criticity of components in a system.

Thus orphan detection and handling is a crucial activity for the fault tolerant multi-agent support systems. We introduced the problem, like performance degradation by taking part in sharing the limited resources available, on the MAS that can potentially be created by orphaned agents. Thus, had we have handled these orphaned agents; our protocol certainly should show better performance. Our protocol does not handle these orphan agents. It can be considered as future work for this thesis.

References

[1] Benno Overeinder, Frances Brazier, and Olivier Marin: Fault Tolerance in Scalable Agent Support Systems: Integrating DARX in the AgentScape Framework, In 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003), pp 688--695, Japan, 2003.

[2] Bracha, G. ``An $O(\log n)$ Expected Rounds Randomized Byzantine Generals Protocol," J. ACM 34, 4 (October 1987), 910--920.

[3] Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E. ``Reaching Approximate Agreement in the Presence of Faults," J. ACM 33, 3 (July 1986), 499--516.

[4] Dolev, D., Ruediger, R., and Strong, H. R. ``Early Stopping in Byzantine Agreement," J. ACM 37, 4 (October 1990), 720--741.

[5] Hadzilacos, V. and Halpern, J. Y. ``Message-Optimal Protocols for Byzantine Agreement," ACM Symposium on Principles of Distributed Computing, 1991, 309--323.

[6] Halpern, J. Y., Moses, Y., and Waarts, O. ``A Characterization of Eventual Byzantine Agreement," ACM Symposium on Principles of Distributed Computing, 1990, 333--346.

[7] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg: Byzantine Fault Tolerance, from Theory to Reality, Proc. 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP03), pp.235-248, UK, 2003.

[8] Leslie Lamport, Robert Shostak, and Marshall Pease: The Byzantine Generals Problem,

ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, 1982, pp 382-401.

[9] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodsony, Michael K. Reiter, and Jay J. Wylie: Fault Scalable Byzantine Fault Tolerant Services, In Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP '05), pp. 59--74, United Kingdom, 2005.

[10] Miguel Castro and Barbara Liskov: Practical Byzantine Fault Tolerance, In the Proceedings of the Third Symposium on Operating Systems Design and Implementation, USA, 1999.

[11] Z. Guessoum, N. Faci, and J. P. Briot: Adaptive Replication of Large-Scale Multi-Agent Systems – Towards a Fault-tolerant Multi-Agent Platform, SELMAS'05 at ICSE'O5, USA, 2005.

[12] N. J. E. Wijngaards, B. J. Overeinder, M. van Steen, and F. M. T. Brazier. Supporting Internet-scale multi-agent systems. Data and Knowledge Engineering, 41(2–3):229–245, June 2002.

[13] F. M. T. Brazier, D. G. A. Mobach, B. J. Overeinder, and N. J. E. Wijngaards. Managing agent life cycles in open distributed systems. In Proceedings of the ACM Symposium on Applied Computing (SAC 2003), Melbourne, FL, Mar. 2003.

[14] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards adaptive fault-tolerance for distributed multi-agent systems. In Proceedings of the European Research Seminar on

Advances in Distributed Systems (ERSADS'2001), pages 195– 201, Bertinoro, Italy, May 2001.

[15] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In Proceedings of the International Conference on Dependable Systems and Networks, Washington, DC, June 2002.

[16] J.-P. Briot, Z. Guessoum, S. Charpentier, S. Aknine, O. Marin, and P. Sens. Dynamic adaptation of replication strategies for reliable agents. In Proceedings of the 2nd Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS-2), London, UK, Apr. 2002.

[17] Andrew S. Tanenbaum and Maarten Van Steen: Distributed Systems Principles and Paradigms, Prentice-Hall of India, New Delhi, 2002.

[18] Leslie Lamport: Time, Clocks, and the Ordering of Events in a Distributed System, Massachusetts Computer Associates, Inc, 1978.

[19] Ian Sommerville: Software Engineering, 7th edition, Addison-Wesley Longman Publishing Co., Inc. Boston, 2004.

[20] Dahila Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright: Probabilistic Quorum Systems, Academic Press, 2001

[21] Rachid Guerraoui and Marko Vukolic: Refined Quorum Systems, ACM, 2007

[22] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault

tolerance. In Proc. SOSP, October 2001.

[23] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In Proc. SOSP, October 2003.

[24] Alan Fedoruk and Ralph Deters. Improving fault-tolerance by replicating agents. In AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multi-agent systems, pages 737–744, New York, NY, USA, 2002. ACM Press.

[25] Niek Wijngaards, Maarten van Steen, Frances Brazier. On MAS Scalability. Proc.2nd Int'l Workshop on Infrastructure for Agents, MAS and Scalable MAS. May 2001.

[26] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. Journal of Distributed Computing, 11(4), Florham Park, NJ USA, 1998.

[27] L. Alvisi, D. Malkhi, E. Pierce, and M. Reiter. ªFault Detection for Byzantine Quorum Systems,º Proc. Seventh IFIP Int'l Working Conf. Dependable Computing for Critical Applications, pp. 357-371, Jan. 1999.

[28] Frances Brazier, David Mobach, and Benno Overeinder. AgentScape Demonstration. http://www.iids.org/, last visited on May 7, 2008.

[29] L. Lamport. Lower bounds for asynchronous consensus. In Proc. FUDICO, pages 22–23, June 2003.

[30] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Common Knowledge Revisited.

[31] Vinod Vaikuntanathan. Distributed Computing with Imperfect Randomness. MIT 2005

[32] Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. Journal of the Association for Computing Machinery, Vol. 32, No. 4, October 1985, pp. 824 – 840.

[33] Miguel Correia, Nuno Ferreira Neves, And Paulo Verissimo. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures.

[34] Valerie King , Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable Leader Election.

Declaration

I, the undersigned, declared that this thesis is an original work and it has not been presented for a degree in any other university. All the sources of the materials used in this thesis are acknowledged.

Full Name	**Signature**
Ashenafi Kassahun	____________________

Advisor Confirmation

Full Name	**Signature**
Mulugeta Libsie (Ph. D.)	___________________

Department of Computer Science

Addis Ababa University

Addis Ababa

Ethiopia