UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

ACHIEVING FAULT-TOLERANT CONSENSUS IN AD HOC NETWORKS

David Rogério Póvoa de Matos

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

ACHIEVING FAULT-TOLERANT CONSENSUS IN AD HOC NETWORKS

David Rogério Póvoa de Matos

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

Dissertação orientada pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

2013

This work was partially supported by:
CE through project FP7-257475 (MASSIF)
FCT through the multianual program (LaSIGE)
PTDC/EIA-EIA/113729/2009 (SITAN)

Acknowledgments

First, I would like to thank my advisor, prof. Nuno Neves, for accepting me as his student, helping me with every issue I faced during the research period, establish a high bar for every task assigned to me, improving the overall quality of my work and for his patience in clarifying every question I had.

I want to acknowledge Faculty of Sciences, particularly the Department of Informatics. During this years I had the opportunity to learn so much, acquire lots of good experiences, met good colleagues and Professors who were very helpful in every issue I faced. I also want to thank the Navigators research team for their welcoming, support and fellowship.

I also want to thank my close friends for the moral support and company during hard times.

A very special thank goes to my lovely girlfriend, Raquel, who were always there for me and encourage me to never give up every time I had second thoughts.

Finally, I want to thank my mother and my father for giving me the opportunity to study away from home and for giving me everything I needed to complete my academic course.

À Raquel, à minha família e amigos

Abstract

Consensus plays a fundamental role in distributed systems. This operation consists in having every process in a distributed system, or a subset of processes, to agree upon a value that was proposed by any of the processes. Consensus is used to solve several problems in distributed systems computation, such as: state machine replication, leader election and atomic broadcast, allowing the coordination of the network. Its applicability becomes more important and difficult to implement in wireless ad hoc networks that are vulnerable to intrusions. When dealing with a wireless ad hoc network, specially one composed by mobile devices that are constantly moving, there are several obstacles that have to be overcome such as the unreliability in the communication, the hardware limitations of the devices, the limited communication range and the exposure to malicious users.

The project consists in the design, implementation, test and analysis of Byzantine fault-tolerant consensus protocols for wireless ad hoc networks. It is assumed that the number of participants is unknown and the consensus protocols execute in a group of processes called sink. The protocols are byzantine fault-tolerant and circumvent both FLP and Santoro-Widmayer impossibility results. Three forms of consensus protocols were considered: binary, multivalued and vector. The protocols were organized in a stack, where lower level protocols were used to build higher ones. The stack was implemented as a library and was tested in a simulation environment. Some preliminary tests were also performed with Android devices. The evaluation of the protocols shows that they exhibit good performance in several scenarios and even under attack.

Keywords: Distributed systems, dependability, security, fault tolerance, intrusion tolerance, agreement, consensus, ad hoc wireless networks

Resumo

O consenso tem um papel fundamental em sistemas distribuídos. Esta operação consiste em ter todos os processos num sistema distribuído, ou um subconjunto de processos, a acordar num valor que foi proposto por algum dos processos. O consenso é usado para resolver vários problemas na computação de um sistema distribuído, como por exemplo: máquina de estados replicada, eleição de líder e difusão atómica, permitindo a coordenação da rede. A sua utilidade torna-se mais importante e difícil de implementar em redes ad hoc móveis sem fios que estão vulneráveis a intrusões. Quando se está a lidar com uma rede ad hoc sem fios, especialmente uma composta por dispositivos móveis que apresentam uma mobilidade constante, existe um conjunto de obstáculos relacionados com a falta de fiabilidade na comunicação, as limitações dos equipamentos, o seu reduzido alcance de comunicação e a exposição a utilizadores mal intencionados.

O projecto consiste no desenho, implementação, teste e análise de protocolos de consenso tolerântes a faltas bizantinas para redes ad hoc sem fios. É assumido que o número de participantes é desconhecido e os protocolos de consenso são executados num grupo de processos denominado poço. Os protocolos são tolerântes a faltas bizantinas e contornam of resultados de impossibilidade de FLP e de Santoro-Widmayer. Foram considerados três tipos de protocolos de consenso: binário, multi-valor e vector. Os protocolos estão organizados numa pilha, onde protocolos de baixo nível foram usados para construir os protocolos de níveis superiores. A pilha foi implementada como uma biblioteca e foi testada em ambiente de simulação. Alguns testes preliminares foram também efectuados com dispositivos Android. A avaliação dos protocolos revela que estes exibem um bom desempenho em vários cenários e mesmo sobre ataque.

Palavras-chave: Sistemas distribuídos, confiabilidade, segurança, tolerância a faltas, tolerância a intrusões, acordo, consenso, redes ad hoc sem fios

Resumo Alargado

As redes ad hoc sem fios são um importante avanço na evolução das redes sem fios. As suas características permitem a operabilidade da rede em caso de catástrofes naturais, em operações militares, ou no dia a dia em situações em que o acesso a um meio de controlo centralizado é escasso ou até mesmo inexistente. Ao contrário das redes tradicionais, os dispositivos integrantes de uma rede ad hoc sem fios apresentam limitações na capacidade de processamento, na energia disponível, no alcance da transmissão e recepção e na largura de banda. Estas limitações têm de ser tomadas em conta quando se constrõem protocolos de comunicação e coordenação.

As redes ad hoc sem fios podem ser compostas por dispositvos pessoais móveis: telémoveis, pdas, computadores portáteis, *tablets*, entre outros, que já possuem algum poder computacional mas que apresentam um comportamento móvel e podem não estar continuamente ligados em rede. Outro tipo de redes ad hoc sem fios são as redes de sensores que são compostas por dispositivos de baixa capacidade de processamento e memória. O uso de operações que exigem elevado processamento por parte dos dispositivos, e protocolos que efectuam demasiadas transmissões de mensagens estão fora de questão. Em qualquer um dos casos não sabemos ao certo quantos dispositivos fazem parte da rede num dado instante uma vez que alguns deles podem se mover ou simplesmente desligar da rede e mais tarde voltar.

Para além das restrições associadas às limitações energéticas e de processamento dos dispositivos também temos de lidar com a hipótese de existirem utilizadores mal intencionados, que se interponham na rede com vista a adulterar o processamento do sistema ou simplesmente a quebrar a prestação do serviço. Esta hipótese é bastante provável uma vez que os nós comunicam através de uma redes sem fios, sendo possível que qualquer dispositivo que se encontre na vizinhança se interponha na rede.

Uma vez que este tipo de redes não tem controlo centralizado, por vezes é necessário que todos os intervenientes mantenham uma coordenação para garantir um uso adequado e eficiente da rede. A coordenação pode ser necessária por várias razões, por exemplo: para garantir que não estejam vários nós simultâneamente a transmitir na rede o que pode causar ruído e impedir qualquer comunicação naquele instante; para garantir a operabilidade do sistema, se o sistema for uma rede de sensores e actuadores é fundamental que

os actuadores actuem com base nos dados adquiridos pelos sensores; para eleição de um coordenador da rede, um líder, que tenha uma tarefa especial na gestão da rede.

Para conseguir a coordenação da rede recorre-se a uma operação de *consenso* em que todos os nós escolhem o mesmo valor dentro de um conjunto de valores propostos. A concretização do consenso não é trivial e está provado que não tem solução em sistemas assíncronos se houver a possibilidade de um processo falhar (resultado de impossibilidade de FLP [8]). Outro resultado de impossibilidade é o de Santoro e Widmayer que estabelece que não existe solução determinística para o problema de *consenso* mesmo com fortes pressupostos de sincronismo se pelo menos n-1 mensagens se perderem [22], o que é bastante provável neste tipo de redes (se um nó falhar perdem-se n mensagens num sistema com n nós).

Num cenário realista em que ocorreu uma catástrofe natural ou durante uma complexa operação militar, pode não se saber ao certo quantos ou quais os intervenientes que fazem parte da rede, e não existe uma forma simples de os reunir fisicamente para se trocar os parâmetros de segurança (chaves usadas na autenticação). Nesta situação é importante que os dispositivos cooperem para que se consiga coordenar a rede. Cavin, Sasson e Schiper apresentaram um algoritmo de *consenso* para esta situação denominado *consenso com participantes desconhecidos* [4] em que não se sabe quem são os participantes nem quantos são. Os participantes da rede conhecem apenas parte de rede e quando é necessário atingir um *consenso* apenas um núcleo da rede executa o algoritmo de consenso e transmite o valor decidido ao resto da rede.

Esta tese tem o objectivo estudar algoritmos de *consenso* e a sua concretização numa pilha de protocolos especializada para redes *ad hoc* sem fios que seja tolerante a faltas bizantinas. A pilha pressupõe a existência de um conjunto de protocolos de suporte e comunicação segura em redes ad hoc, pois tira partido de uma operação de difusão fiável autenticada, e fornece uma primitiva de *consenso*. Da mesma maneira que esta pilha também está preparada para que outros protocolos sejam construídos no topo dela, por exemplo para a difusão atómica.

Palavras-chave: Sistemas distribuídos, confiabilidade, segurança, tolerância a faltas, tolerância a intrusões, acordo, consenso, redes ad hoc sem fios

Contents

List of Figures **xxii**

List of Tables **xxv**

1 Introduction **1**
1.1 Motivation 2
1.2 Objectives 2
1.3 Contributions of the Thesis 2
1.4 Document Structure 3

2 Related Work **5**
2.1 The Consensus Problem 5
2.1.1 System Model 6
2.1.2 Impossibility Results 7
2.1.3 The Byzantine Generals Problem 9
2.2 Ad hoc Wireless Networks 9
2.2.1 Common Attacks in Wireless Ad hoc Networks 9
2.2.2 Consensus in Ad hoc Wireless Networks 11

3 Binary, Multivalued and Vector consensus protocols **13**
3.1 System Model 13
3.2 Protocol Stack 15
3.2.1 Binary Consensus 15
3.2.2 Multivalued Consensus 23
3.2.3 Vector Consensus 25

4 Implementation and Evaluation **29**
4.1 Design and Implementation of the Protocol Stack 29
4.1.1 Single-threaded and Multi-threaded Mode 29
4.1.2 Protocol Context 31
4.1.3 Library Architecture 31
4.1.4 Application Programming Interface 32

4.2 NS-3 Simulation 38
4.3 Android Implementation 39
4.4 Simulation Results 40
4.4.1 Experimental Environment 40
4.4.2 Binary Consensus 41
4.4.3 Multivalued Consensus 49
4.4.4 Vector Consensus 51

5 Conclusion **55**
5.1 Discussion of the results 55
5.2 Future Work 56

Abbreviations **57**

Bibliography **61**

List of Figures

3.1 Protocol stack 14
3.2 Interactions between the protocols 16
3.3 Execution example of Algorithm 1 19

4.1 Class diagram of the android network library 33
4.2 Sequence diagram of a vector consensus execution 34
4.3 Latency of binary consensus with divergent values; byzantine faults affect the identity of the sender 42
4.4 Latency of binary consensus with divergent values; byzantine faults affect the phase number 42
4.5 Latency of binary consensus with divergent values; byzantine faults affect the status value 43
4.6 Latency of binary consensus with divergent values; byzantine faults affect the proposal value 43
4.7 Latency of binary consensus with unanimous values; byzantine faults affect the identity of the sender 44
4.8 Latency of binary consensus with unanimous values; byzantine faults affect the phase value 44
4.9 Latency of binary consensus with unanimous values; byzantine faults affect the status value 45
4.10 Latency of binary consensus with unanimous values; byzantine faults affect the proposal value 45
4.11 Number of phases of binary consensus with divergent values; byzantine faults affect the identity of the sender 46
4.12 Number of phases of binary consensus with divergent values; byzantine faults affect the phase value 46
4.13 Number of phases of binary consensus with divergent values; byzantine faults affect the status value 47
4.14 Number of phases of binary consensus with divergent values; byzantine faults affect the proposal value 47
4.15 Exchanged messages of binary consensus with unanimous values; byzantine faults affect the proposal value 48

4.16 Exchanged messages of binary consensus with divergent values; byzantine faults affect the proposal value 48
4.17 Latency of multivalued consensus with divergent values; byzantine faults affect the identity of the sender 49
4.18 Latency of multivalued consensus with divergent values; byzantine faults affect the proposal value 50
4.19 Latency of multivalued consensus with unanimous values; byzantine faults affect the identity of the sender 50
4.20 Latency of multivalued consensus with unanimous values; byzantine faults affect the proposal value 51
4.21 Latency of vector consensus with divergent values; byzantine faults affect the identity of the sender 51
4.22 Latency of vector consensus with divergent values; byzantine faults affect the proposal value 52
4.23 Latency of vector consensus with unanimous values; byzantine faults affect the identity of the sender 52
4.24 Latency of vector consensus with unanimous values; byzantine faults affect the proposal value 53

List of Tables

3.1 Example of table VK_i with the messages signatures for process p_i 19

List of Algorithms

1 Turquois: a Byzantine k-consensus algorithm 17
2 $validate(m)$ 20
3 $validatePhase(m, count)$ 21
4 $validateStatus(m, dCount_0, dCount_1, dCount_\perp, dCount_{total})$ 21
5 $validateValue(m, count_0, count_1)$ 23
6 Multivalued consensus protocol 25
7 Vector consensus protocol 26
8 $selectValue(j, array_i)$ 27

Chapter 1

Introduction

In the last years there has been a proliferation of mobile devices: laptops are increasingly lighter and powerful, and smartphones and tablets are becoming much more common. All these devices share a group of characteristics: wireless communication, limited processing, memory and battery power, and some level of mobility.

There is a range of distributed applications that can be developed for this kind of devices, such as: productivity tools for groups, schedulers, chat applications, save and rescue applications for emergency scenarios. These distributed applications take advantage of the wireless communication capacity and mobility of the devices, but must deal with the processing, memory and autonomy limitations.

A network composed only by mobile devices (from now on designated processes) is called Mobile Ad hoc NETwork (MANET). These networks are self organized because there is no centralized unit to configure and manage them. These networks exhibit several limitations: unreliability of the communication medium, due to noise in the communication channel or obstacles to messages transmission; unreliability of the processes, as they may disconnect at any time or be apart from the network; and asynchrony of message delivery.

The lack of a central coordinator to configure and manage the network hampers their management. Every management decision must be taken by the group and they must agree on the decision taken. The management of the network requires several coordination actions, such as: agreeing on a unique process to be leader; deciding the ordering of the received messages; assigning roles to processes; deciding routes. Also, several distributed applications need the agreement on some specific values.

A fundamental operation used to coordinate the network is called consensus. The consensus problem attempts to reach an agreement on some data value among a group of processes given some information that is proposed by each one of them. Consensus solves several problems, such as: leader election, state machine replication and atomic broadcast. It provides a means of agreement to a distributed application.

1.1 Motivation

Consensus is a crucial operation to coordinate and configure a MANET. However, achieving consensus in such system model it is a very difficult operation. Most consensus algorithms do not consider the system model of a MANET, making them useless in this kind of networks. The mobility of the processes causes message loss, the wireless communication is unreliable and asynchronous. It was proven that it is impossible to reach agreement in asynchronous systems in the presence of certain faults [8][22]. Having a consensus protocol specific for this kind of network and that is able to circumvent [8][22] would be very helpful to manage and configure such network and to assist the development of distributed applications.

Developing distributed applications can be a very difficult task, specially in a MANET system. The MANET system presents several obstacles due to the unreliability of wireless communication and mobility and fault model of the processes. Additionally, a programmer must assume the existence of byzantine processes, since the wireless communication is exposed. The programmer needs to ensure the network is correctly configured and must cope with the unreliability of the communication medium and the possible faults the processes may suffer. Having a library that deals with these problems would assist the developers work and would allow the implementation of other protocols that need a consensus primitive, such as atomic broadcast.

1.2 Objectives

Developing distributed applications for mobile devices assuming the presence of byzantine faults is very difficult and there is no framework providing a coordination primitive. Also the existent multivalued and vector consensus protocols do not consider the chosen system model and fault model. So the objectives of this work are:

- the study of three byzantine fault-tolerant consensus protocols: binary, multivalued and vector;
- the adaptation of existing protocols [17] [6] so they can fit the MANET system model;
- the implementation of a protocol stack with the three consensus protocols;
- the evaluation of the protocol stack in the Network Simulator-3 (NS-3) simulator [10].

1.3 Contributions of the Thesis

The contributions of this thesis are:

- a new protocol for vector consensus and the adaptation of the multivalued consensus protocol of [6] to MANETs. A new message validation process was also created for the binary consensus of [17];
- the implementation and evaluation with the network simulator NS-3 [10] of three consensus protocols for MANETs: binary, multivalued and a vector;
- the preliminary implementation of the studied protocols in the mobile platform Android[20], as a network library for distributed applications.
- some of the results of the thesis were presented in a paper in the INFORUM'13 Simpósio de Informática, with the title "MiCCS4Mobile - Middleware para Comunicação e Coordenação Segura em Redes Ad-hoc".

1.4 Document Structure

This document is organized as follows:

- Chapter 2 – Related work - short description about the research and developed work made so far in this area: presents the consensus problem and the Michael J. Fischer, Nancy Lynch, and Mike Paterson (FLP) and Santoro-Widmayer impossibility results; describes some of the research in ad hoc wireless networks explaining its limitations, respective attacks and the consensus protocols specific for this kind of networks.
- Chapter 3 – Binary, Multivalued and Vector consensus protocols - a formal definition of the considered system model, the fault model and the guarantees provided by each of the protocols. The modifications to the validation process of binary consensus protocol are described. The adaptations to the multivalued and the new vector consensus protocols for MANET are also discussed.
- Chapter 4 - Implementation and evaluation - A full description of the design and implementation of the stack of consensus protocols. The simulation and the execution environment used to evaluate the protocol stack are also explained; the results, organized by protocol and execution scenarios, are presented and discussed.
- Conclusion - reviews of the developed work listing the accomplished results and the aspects that could be improved. Future work is presented with possible features that can be implemented.

Chapter 2

Related Work

This chapter explains the related work made concerning byzantine fault-tolerant consensus for wireless ad hoc networks. Section 2.1 describes in detail the consensus problem, the system model and its particularities and, the main impossibility results and how it is possible to circumvent them. Section 2.2 explains the characteristics of a wireless ad hoc network, the specific attacks to these networks, and how consensus can be achieved by introducing two protocols.

2.1 The Consensus Problem

Every distributed system needs its processes to agree on one or more values. Agreement must be an unanimous operation among the correct processes and must resist to the failures that may occur, whether they are of accidental causes or a malicious attempt to corrupt the system.

The agreement problem in distributed systems is generalized into a *consensus* abstraction [25]. The consensus problem consists in having every process from the group to agree on a value which was previously proposed by these processes. The consensus problem is equivalent to the state-machine replication problem and atomic broadcast.

A formal presentation of the consensus problem assumes a known set of n processes. If a process executes the protocol from beginning until the end without failing or deviating from the protocol it will be considered correct, otherwise it will be faulty. Each process p_i has a unique identifier i which is known by every process in the group. The operation starts when every correct process proposes a value v_i and terminates when the all the processes decide the same value v.

A typical consensus protocol provides the following properties:

- **Validity.** If every correct process proposes the same value v, then any process that decides, decides v.

- **Agreement.** No two correct processes decide differently.

- **Termination.** Every correct process decides.

Depending on the system model these properties may vary to adjust to the system's limitations. The *validity* property guarantees that the processes decision is based on the proposal values and it is not a deterministic value that relive the interaction between the processes. The *agreement* property if essential for the purpose of the operation, without this guarantee a consensus operation would be useless. Finally, the *termination* property ensures that the protocol does something.

The properties can be grouped into two categories. Validity and agreement are safety properties, i.e., they ensure that the protocol does not do unwanted things. *Termination* is a liveness property which ensures that something is done during the execution.

2.1.1 System Model

The system model influences how the protocol works in order to provide a consensus operation. The system model is defined in terms of how activities proceed in the system and the kind of faults that may occur.

Timing Model

The timing model defines how the system executes its tasks and the processing duration. There are two main timing models: synchronous and asynchronous.

Synchronous system. Assumes that there is a known upper bound in the message transmission and in the processing time of the processes. Formally [9], a synchronous system is one that guarantees:

- there is a known upper bound on the time required by any process to execute a computational step;
- there is a known upper bound on message delay; this consists in the time it takes to send, transport, and receive a message over any link;
- every process has a local clock with a known bounded rate of drift with respect of real time.

These properties allow the system to detect failures by measuring the time that is taken by a the process to transmit a message. If the process exceeds the maximum delay of message transmission and processing time then one can assume that there was a failure. A failure detection mechanism is a very helpful tool to achieve consensus. When a process fails, it is excluded from the group of correct processes, so it does not violate the termination property.

Asynchronous system. An asynchronous system is the opposite of the synchronous. In this system model there is no assumption about the amount of time needed for message transmission or processing time. This characteristic makes impossible to know when processes fail. In an asynchronous system the consensus operation is more difficult to accomplish (see below).

Fault Model

A fault model defines the kind of faults that may occur in the system [9]. A fault occurs when a process fails to do what it is expected to do, which can be related to failing to send a message or to deviating from the protocol. The faults can be classified in two groups:

- **omissive faults**: when an expected message from a process fails to be delivered. These faults can be caused by broken communication channels or by processes that crashed;

- **byzantine faults**: it is an arbitrary fault that occurs when a process does something that varies from the protocol specification. This fault can be caused intentionally to corrupt the system by breaching the protocol properties.

2.1.2 Impossibility Results

Achieving consensus is a difficult problem. In some system models with certain characteristics it was proved that it has no deterministic solution. This section describes two impossibility results: the FLP and the Santoro-Widmayer impossibility results.

The FLP Impossibility Result

Fischer, Lynch and Paterson established an impossibility result [8] to solve deterministic *consensus* in an asynchronous system if only one process fails. The impossibility result assumes an asynchronous system composed by n processes with reliable channels and assume that processes can fail by crashing. This impossibility result affects systems that assume byzantine faults because a byzantine process may fail arbitrarily. The result is related to the fact that in an asynchronous system there is no way to know, for sure, if processes failed. When a process does not receive a message from a process p this can be caused by the fact that p failed (by crashing) or because it took longer to transmit the message. This condition can cause the processes to wait indefinitely.

Circumventing the FLP impossibility result. Although it is impossible to achieve consensus in asynchronous systems if only one process fails, there are some techniques that allow to circumvent the FLP result [25]:

- **Partial synchrony**: although it is proven that it is impossible to achieve consensus in asynchronous systems, that does not necessarily means that a complete synchrony is needed. The solution consists in declaring that some parts of the system can be synchronous while the remaing parts are asynchronous and defining which parts of the system need synchrony in order to achieve consensus;

- **Failure detector**: this technique consists of having a special failure detector that provides information about which processes have failed. The consensus protocol can take advantage of this information to ensure that a decision is taken.

- **wormholes**: the wormhole technique is an architectural hybridization that consists in enhancing certain parts of the system in order to provide stronger properties not provided by the standard system. In this technique some support services can execute inside a wormhole, which can be built to guarantee timely behavior and be immune to byzantine faults. The consensus protocol then uses the service to ensure termination. This mechanism requires implementation of such a wormhole in the system, which can be difficult in some environments like an ad hoc wireless network;

- **randomization**: the randomization technique circumvents the FLP impossibility result by discarding the deterministic requirement. The protocols considered in the impossibility result assume that the algorithms are deterministic, which means that for a given input the system will always return the same output. If instead of having a fully deterministic algorithm, some parts of it are random, then it would be possible to achieve consensus with a probability. The technique consists in having a protocol that makes progress in rounds where some value of the protocol is generated randomly. The advantage of this approach is that it does not need to alter any assumption about the environment.

  Randomized consensus protocols use a virtual coin to generate random values to the process. There are two kinds of randomized consensus protocols: local virtual coin, each process has an internal mechanism that returns a random value; shared virtual coin, the mechanism generates the same random value to be used by every process. The binary consensus protocol considered in this work is a local virtual coin protocol.

The Santoro-Widmayer Impossibility Result

The Santoro-Widmayer impossibility result [22] is associated with systems in which failures may occur in the communication links. This result assumes a synchronous system in which both the transmission and processing delay are bound by a known constant. It

concerns a specific instance of consensus called k-agreement. A k-agreement consensus requires that at least $k > \frac{n}{2}$ out of n processes agree on a value $v \in \{0, 1\}$. The result states that there is no deterministic solution to achieve k-agreement in a system composed by n processes if more than $n - 2$ messages can be lost during a communication step. It only takes one process to fail to lose $n - 1$ messages. Most of the techniques utilized to circumvent the FLP impossibility result are not effective to circumvent the Santoro-Widamyer impossibility result because it assumes a completely synchronous system. Nevertheless, the protocol used in this project circumvents both impossibility results using randomization.

2.1.3 The Byzantine Generals Problem

The Byzantine Generals Problem [15] is an agreement problem in which generals of the of Byzantine army have to make a decision about attacking or not a castle. The generals communicate with each other through messengers. Each general chooses to attack or not. All the generals must agree with the same action but some of them are traitors and they may mislead other generals to perform the wrong action. There is no solution to this problem for $n < 3 * f + 1$, where n is the total number of generals and f the total number of traitors. This problem is similar to the binary consensus problem and it helps to define the correct f (number of Byzantine faulty processes) in our system.

2.2 Ad hoc Wireless Networks

Ad hoc wireless networks [27] are self organized networks composed by mobile devices such as laptops, smartphones, tablets, Personal Digital Assistant (PDA)s and other mobile devices with wireless communication capabilities. They dispense the existence of a centralized device to control and manage the network. This characteristic makes them useful to scenarios of disaster relief, conference, and battlefield environments.

The nature of ad hoc networks and the wireless and mobile communications exhibit several issues and limitations such as: lack of bandwidth optimization and power control, poor quality of transmission due to noise or obstacles. Also the inexistence of a central infrastructure to configure the network, discover the devices, maintain the topology, assign addresses to the devices and resolve routing aspects makes these networks a specific topic of research and development. There are several protocols for this kind of networks that deal with different problems from the standard networks to accomplish the same goals.

2.2.1 Common Attacks in Wireless Ad hoc Networks

The nature of ad hoc wireless networks and the exposure of the communication reveal many vulnerabilities and consequently several possible attacks that can be performed.

At the route discovery phase and during maintenance, malicious devices can monitor the traffic and use this information to deceive the remaining nodes by transmitting false information of the network topology, tampering the Internet Protocol (IP) Addresses of the data packets and also identifying resources ID. Attacks can be grouped into two categories: passive attacks and active attacks.

Passive attacks. In this kind of attacks the adversary monitors the network traffic and it does not disturb the network operations. It is very difficult to discover such an attack since the attacker does not perform any detectable action. By monitoring the traffic in the network, a malicious user may obtain sensitive information that it can use to perform active attacks. Passive attacks are usually the first step of an attacker, where it collects sensitive information, like cryptographic and network parameters.

Active attacks. It is called an active attack when an attacker attempts to modify or destroy the data packets being exchanged between source and destination. By doing this an attacker is able to manipulate the protocol execution. These attacks are easier to detect than the passive attacks. Active attacks can be:

- based on modification: malicious processes may fake routes so that the packets do not reach the final destination. This attack can be classified as a redirection and a Denial of Service (DOS) attack;

- impersonation attacks: malicious processes perform these attacks in order to impersonate other processes by changing their IP or Media Access Control (MAC) addresses. This is also called *spoofing*.

- creating information: there are several variants of this attack [19], which can be performed by fabricating information. Three examples of this attacks are:

  - falsification of route error messages: a malicious process sends fake and incorrect route error messages from some correct process to force the routing protocol to remove it;
  - route cache poisoning: when the routes are being calculated, a malicious process sends spoofed packets in order to congest the network, make it inaccessible or make it slower by suggesting a least advantageous path;
  - routing attacks: a malicious process inserts itself between correct processes to absorb the communication. Later it can use the received packets and resend them and waste battery power and network bandwidth. Moreover, it is able to flood the network by broadcasting the received packets to every process [26].

Other attacks. There is another group of attacks which are more sophisticated.

- Wormhole attack: connects two parts of the network via one exclusive link controlled by the attacker. A malicious process is placed in the link and tunnels the communication [12], [23] and [14].
- Black hole attack: a malicious process inserts itself between the communication by sending incorrect information during the routing phase, it then absorbs all the packets and stops forwarding them [11];
- Gray hole attack: it is similar to a black hole attack but the attacker only discards some data packets. This may happen selectively or statistically;
- Resource consumption attack: a malicious process can consume other correct processes battery by sending unwanted packets and creating a route loop.

2.2.2 Consensus in Ad hoc Wireless Networks

The consensus protocols for ad hoc wireless networks must take in consideration all the limitations in these networks. Typically they only use the broadcast operation for communication, which saves energy, and they use less expensive processing operations both for authentication and for the protocol operation itself. The binary consensus protocol considered for this work is Turquois [17], which is a k-consensus protocol designed for MANETs.

Turquois: a Byzantine Fault-tolerant Binary Consensus

Turquois is a byzantine fault-tolerant binary k-consensus algorithm for wireless ad hoc networks. It takes advantage of the processing and memory limitations of the mobile devices in a MANET and it only uses the wireless broadcasting medium as the communication channel. It also avoids using expensive asymmetric cryptography during the normal execution of the algorithm. It circumvents the FLP [8] and the Santoro-Widmayer [22] impossibility results by applying randomization. The algorithm works in two tasks that execute concurrently. The first task periodically broadcasts the internal state of the process. The second task handles the received messages. Progress is ensured in rounds of three phases. Once a process collects a quorum of $\frac{n+2}{2}$ valid messages in its phase it makes progress to the next phase (more details are provided in the next chapter).

Byzantine Fault-tolerant - Consensus wiht Unknown Participants (BFT-CUP)

BFT-CUP [1] - Byzantine Fault-Tolerant Consensus with Unknown Participants is an algorithm that solves consensus in a network in which the number of participants is unknown and some of them may behave maliciously. It assumes an asynchronous system

model composed by a set of Π processes from a larger universe with n processes. There are two kinds of networks: *known networks* in which every process knows n and Π and *unknown networks* in which a process $p_i \in \Pi$ may only be aware of a subset $\Pi_i \subseteq \Pi$. The protocol uses *authenticated and reliable point to point channels* established between known processes. The set of known processes is not fixed: if a process p_i sends a message to p_j and $p_i \notin \Pi_j$ then p_j adds p_i to its Π_j. A *participant detector* (Participant Detector (PD)) is used to provide to each process the set of known processes it can communicate with. The participant detector provides the following two guarantees:

- **Information inclusion:** the information returned by the participant detectors is not decreasing over time;
- **Information accuracy:** the participant detector does not make mistakes

The PD gives each process a context about which processes are reachable other by returning a directed knowledge connectivity graph. The graph is directed because the set of processes a process knows may be different from the group of processes the other process knows. Processes are connected through more than one disjoint paths. A process p_j is k-strongly connected to another process p_i if p_j can reach p_i from k-disjoint paths. Gdi is the directed graph representing the knowledge relation determined by the PD oracle. A component *Gs* of *Gdi* is a *sink component* when there is no path from a process in *Gs* to other processes of *Gdi*, except processes from *Gs* itself. The BFT-CUP uses the weakest participant detector defined to solve Fault-tolerant - Consensus wiht Unknown Participants (FT-CUP), which is called k-One Sink Reducibility (OSR). The knowledge connectivity graph *Gdi* satisfies the following conditions:

- the undirected knowledge connectivity graph *G* obtained from *Gdi* is connected;
- the directed acyclic graph obtained by reducing *Gdi* to its *k*-strongly connected components has exactly one sink;

The consensus protocols described on the next chapter were developed to be run among the processes that belong to the sink. Once they reach a decision, they propagate this value to the remaining processes.

Chapter 3

Binary, Multivalued and Vector consensus protocols

This chapter describes the binary, multivalued and vector consensus algorithms used to implement the protocol stack. It is divided in three sections. The first section makes a formal presentation of the system model. The second section explains in detail the protocol stack, the architecture and the interactions between the protocols. It includes a description of each consensus protocol in detail, presenting the algorithms so that they fit the system model.

3.1 System Model

The algorithms are designed to be executed in wireless ad hoc networks composed by mobile devices (MANETs). The communication channels are unreliable and, to take advantage of the wireless capacity of the devices, all the messages are transmitted through a broadcast. The system model for the complete protocol stack (represented in Figure 3.1, and divided in support layer and consensus layer) is the one defined in [1], which considers an universe $\cup$ of unknown participants.

The support layer main task is to discover and provide a group of known processes in the network, called the *sink*, in which the consensus protocols will execute. The *sink* is composed by a set of $\Pi = \{p_0, p_2, p_3, ..., p_{n-1}\}$ processes. Each process p_i has a unique identifier $i \in \{0, ..., n-1\}$ and every process knows the identifiers of the other ones. The fault model assumes the existence of byzantine faults, meaning that up to f process may fail arbitrarily. A byzantine process can become silent, send messages with wrong values, or work together with other byzantine processes to corrupt the properties of the system. The fault model also assumes that correct process may fail to transmit messages as the network is unreliable (i.e., suffer an omission fault in the communication).

All the consensus protocols make progress in rounds and tolerate dynamic omission faults from correct processes, meaning that the safety properties are ensured no matter how many omission faults occur. However, in order to make progress, the number of

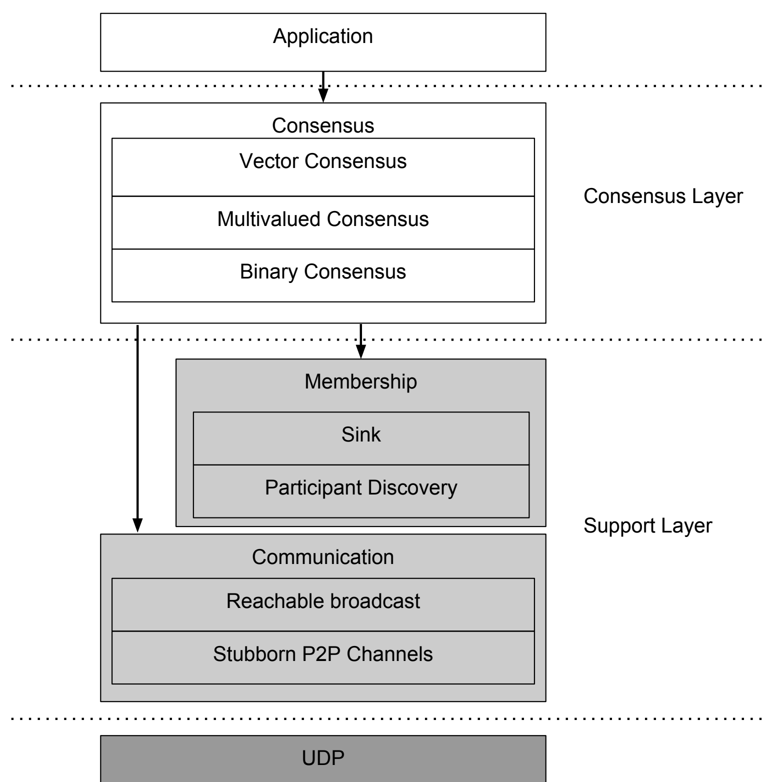


Figure 3.1: Protocol stack

omissions must never exceed a certain bound (see the protocols description). A process that executes the protocol from the beginning to the end without failing or deviating from the original algorithm is correct. If it fails during the execution, the process is considered incorrect. The algorithm tolerates up to f faulty processes as long as $f < \frac{n}{3}$.

A process p_i has a pair of public (pu_i) and private (pr_i) keys [24]. Every process knows the public keys from the other process, but the private key is only known to the owner. Some protocols in the stack use a cryptographic hash function $H(m)$ with the following characteristics:

- for every message m, regardless of $size(m)$, $H(m)$ will return a $hash_m$ value with a constant size s;
- it is impossible to find m given a $hash_m$;
- given m, it is impossible to find m', such that $H(m) = H(m')$.
- it is impossible to find m and m', such that $H(m) = H(m')$;

The protocols use an unreliable broadcast primitive that appends to the messages a Message Authentication Code (MAC) for each potential received. This MAC allows the receiver to authenticate the sender of the message and check the integrity of its contents.

Each process has a local virtual coin that returns a random bit with uniform distribution probability observable only by the process and a local clock used to trigger periodic tasks of the protocols.

3.2 Protocol Stack

The consensus protocols were developed to be installed on the top of a communication and a membership support for ad hoc networks, as depicted in Figure 3.1. The support layer is based on the User Datagram Protocol (UDP) and was developed in [3]. The consensus protocols run among the *sink* and once a decision is reached, the decided value is propagated to the rest of the network. The broadcast primitive used by consensus is the reachable broadcast, which only broadcasts in the *sink*. The binary consensus protocol is the lowest protocol on the consensus layer. Next, there is the multivalued consensus protocol that executes an instance of the binary consensus and decides on non-binary values. The top protocol is vector consensus that decides on sets of values. Above the vector consensus protocol is the application level where other protocols can use the primitives provided by the stack.

Figure 3.2 displays the interactions among the protocols of the stack. The membership level finds the group Π of known processes located at the *sink* of the network. The communication level offers a broadcast primitive that transmits a message in the *sink* of the network. Binary consensus uses the group membership to get the set of n processes in Π and the communication primitives from the support layer. It provides propose and decide operations for multivalued consensus and the application level. Simultaneously, the multivalued and vector consensus implement propose and decide operations, and call the protocols below in the stack.

3.2.1 Binary Consensus

The binary consensus protocol is a k-consensus protocol [17]. Every process p_i proposes a value v_i, such that $v_i \in \{0, 1\}$, and decides on a value $v \in \{0, 1\}$. Only k processes, such that $\frac{n+f}{2} < k \leq n - f$, are expected to decide. The remaining correct processes (at most $n - k$) might not decide, but if they do, the same value must be selected as the rest.

The binary consensus ensures the following properties:

- **BC1 Validity.** if all processes propose the same value v, then any correct process that decides, decides v.

- **BC2 Agreement.** no two correct processes decide differently;

- **BC3 Termination.** at least k correct processes eventually decide with probability 1.

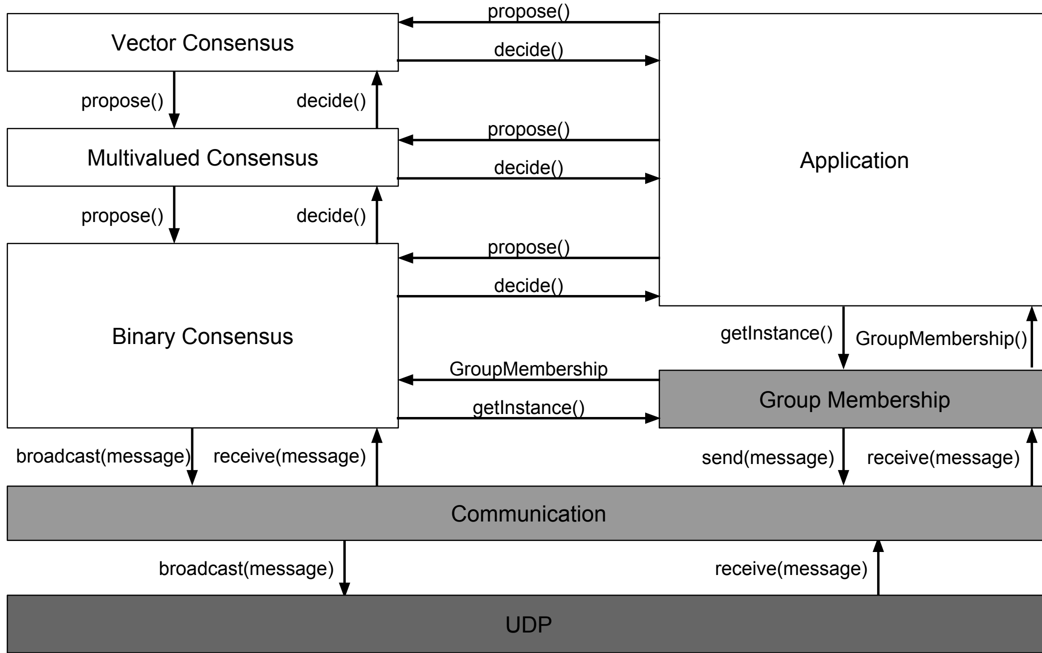


Figure 3.2: Interactions between the protocols

Algorithm

Each process p_i has an internal state containing: the current phase number ϕ_i; its current value v_i; a boolean flag $deterministic_i$ that indicates if v_i was generated randomly or deterministically; a double dimension array $R[p][3]$ with random numbers only visible by p_i with size p x 3 (p is an upper bound of the number of phases a consensus execution might have; if the estimation is wrong and more phases have to be run, then a new array of random values R_i needs to be generated and exchanged; see later on in this section) and a boolean value, $status_i$, saying if p_i has decided yet (see Algorithm 1). Additionally, the process has a vector V_i with every valid received message.

The algorithm works with two parallel tasks. Task T1 is triggered by a local timer to broadcast the process internal state. A process p_i broadcasts the internal state in a message m of the form $\langle i, \phi_i, v_i, status_i, deterministic_i, R[\phi_i][v_i], J_i \rangle$. J_i is an array with the messages that justifies m and allow its validation. J_i includes the received messages in the previous phase and the messages in the last decided phase $\phi_d < \phi_i$. To optimize the communication, the first time p_i sends a message in the current phase (when $\phi_{last} < \phi_i$) it does not append J_i (lines 10-12). Then, ϕ_{last} is updated (line 12). J_i is only sent when p_i is retransmitting its state, in other words, when $\phi_{last} = \phi_i$ (lines 13-16).

Algorithm 1: Turquois: a Byzantine k-consensus algorithm

Input: Initial binary proposal value $proposal_i \in \{0, 1\}$
Output: Binary decision value $decision_i \in \{0, 1\}$

```
1  φ_i ← 1;
2  v_i ← proposal_i;
3  status_i ← undecided;
4  V_i ← 0;
5  R_i ← (rand(), ..., rand());
6  φ_last ← 0;
7  deterministic_i ← 1;
8  Task T1
9  when local clock tick :
10     if φ_last < φ_i then
11         broadcast(⟨i, φ_i, v_i, status_i, deterministic_i, R_i[φ_i][v_i], []⟩);
12         φ_last ← φ_i;
13     else
14         φ_d ← last DECIDE phase φ < φ_i ;
15         J_i ← {⟨∗, φ, ∗, ∗, ∗, ∗, ∗⟩ ∈ V_i : φ ∈ {φ_i − 1, φ_d}};
16         broadcast(⟨i, φ_i, v_i, status_i, deterministic_i, R_i[φ_i][v_i], J_i⟩);
17 Task T2
18 when m = ⟨j, φ_j, v_j, status_j, deterministic_j, r_j, J_j⟩ is received :
19     V_i ← V_i ∪ {m : m is valid};
20     if ∃⟨∗, φ, v, status, ∗, ∗, ∗⟩ ∈ V_i : φ > φ_i then
21         φ_i ← φ;
22         if φ(mod 3) = 1 ∧ deterministic_i = 0 then
23             v_i ← coin_i();
24             deterministic_i ← 0;
25         else
26             v_i ← v;
27             deterministic_i = 1;
28         status_i ← status;
29     if |{⟨∗, φ, ∗, ∗, ∗, ∗, ∗⟩ ∈ V_i : φ = φ_i}| > (n+f)/2 then
                                          /* phase CONVERGE */;
30         if φ_i(mod 3) = 1 then
31             v_i ← majority value v in messages with phase φ = φ_i;
32                                        /* phase LOCK */;
33         else if φ_i(mod 3) = 2 then
34             if ∃v ∈ {0, 1} : |⟨∗, φ, v, ∗⟩ ∈ V_i : φ = φ_i| > (n+f)/2 then
35                 v_i ← v;
36             else
37                 v_i ← ⊥;
38                                        /* phase DECIDE */;
39         else
40             if ∃v ∈ {0, 1} : |{⟨∗, φ, v, ∗, ∗, ∗, ∗⟩ ∈ V_i : φ = φ_i}| > (n+f)/2 then
41                 status_i ← decided;
42             if ∃v ∈ {0, 1} : |{⟨∗, φ, v, ∗, ∗, ∗, ∗⟩ ∈ V_i : φ = φ_i}| ≥ 1 then
43                 v_i ← v;
44                 deterministic_i = 1;
45             else
46                 v_i ← coin_i();
47                 deterministic_i = 0;
48         φ_i ← φ_i + 1;
49     if status_i = decided then
50         decision_i ← v_i;
```

Task T2 handles the received messages. Every message must pass the validation procedure of Algorithm 2 before it is stored in vector V_i (line 19). Lines 20 and 21 apply a technique to jump phases. When a process receives a valid messages with phase ϕ greater than ϕ_i, it updates ϕ_i, v_i and $status_i$ (lines 20-28). This technique allows processes that have lost communication for a certain amount of time (and stood some phases behind) to catch up with the rest of the group.

Once a process collects more than $\frac{n+f}{2}$ messages in the current phase (line 29), it updates the internal state in the following way: if it is at a *CONVERGE* phase then it picks the majority value among the messages received at that phase (line 31); if it is at a *LOCK* phase, it picks the value proposed by more than $\frac{n+f}{2}$ (line 35) or $\perp$ if not enough processes chose the same value (line 37); if it is at a *DECIDE* phase, it updates the proposal value v_i if there are one or more messages with the (same) value $v \in \{0, 1\}$, otherwise it updates the value v_i with the result of a coin flip (lines 43 and 46). Additionally, it can update the status value $status_i$ to $decided$ if there more than $\frac{n+f}{2}$ messages received with the same value v different from $\perp$ (line 41). Finally, it updates the phase value by incrementing it (line 48).

Figure 3.3 represents an execution example of the Algorithm 1. In the example $n = 4$ and p_0, p_1 and p_2 are correct and p_3 is byzantine. At the beginning p_0 and p_2 propose 1, p_1 and p_3 propose 0. Once they receive the initial proposals, p_3 chooses an incorrect value. At phase $LOCK$ the message from p_3 is discarded because it does not pass validation. At phase $DECIDE$, p_0, p_1 and p_2 decide 1. p_3, which is incorrect, decides 0. The fact the p_3 sent wrong messages did not affect any of the properties of the algorithm.

Message Validation

As indicated in line 19 of Algorithm 1, every received message is validated before being treated by Task T2. If it fails the validation process then the message is not used by the protocol. There are two complementary methods of validation: authentication and semantic.

Authentication. The authentication method validates messages using cryptographic techniques. At the beginning of execution, the processes exchange a table VK with hash values (see Table 3.1). The table is signed by an one-way function *F* (e.g., RSA [21]). The lines of the table correspond to phases and the columns to the possible proposal values.

The table VK_i for a process p_i is created using an one-way hash function H (e.g. SHA-256 or RIPEMD-160)[16] to calculate the hash values of the table. H takes as input a value composed by: the proposal value ($v \in \{\perp, 0, 1\}$), the phase number ϕ and a different random value $R_i[\phi][v]$ for each v_i and ϕ_i. If the number of phases exceeds the projected upper bound of random values generated in R, then a new array of random vales R' must be generated and a new table VK' must be exchanged by the processes.

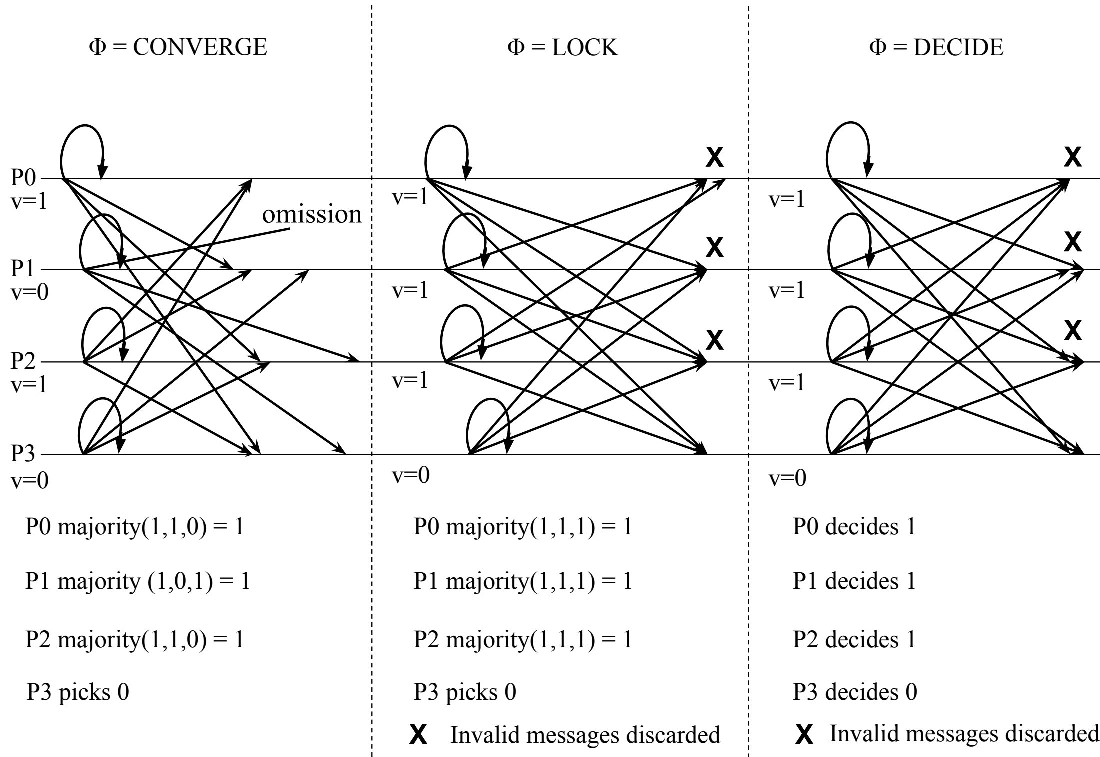


Figure 3.3: Execution example of Algorithm 1

When p_i broadcasts a message with phase ϕ_i and value v_i, it appends the random value, $R_i[\phi_i][v_i]$, used to generate the corresponding h in the table VK_i. With this random value, along side with the phase value and the proposal value, the receiver is able to get the hash value h'. h' is generated with the hash function H using as inputs: v_i, ϕ_i and $R_i[\phi_i][v_i]$. If h' matches $VK_i[\phi_i][v_i]$ then the message is authenticated. This method ensures the receiver that p_i sent the message because, according to the properties of H, it is impossible to generate h' without knowing the input value used by H. The security of this mechanism lies with the secrecy of the random values.

Phase	$v = \perp$	$v = 0$	$v = 1$
ϕ_1	$H(\langle\phi_1, \perp, R_i[1][\perp]\rangle)$	$H(\langle\phi_1, 0, R_i[1][0]\rangle)$	$H(\langle\phi_1, 1, R_i[1][1]\rangle)$
ϕ_2	$H(\langle\phi_2, \perp, R_i[2][\perp]\rangle)$	$H(\langle\phi_2, 0, R_i[2][0]\rangle)$	$H(\langle\phi_2, 1, R_i[2][1]\rangle)$
...	...	...	...
ϕ_p	$H(\langle\phi_p, \perp, R_i[p][\perp]\rangle)$	$H(\langle\phi_p, 0, R_i[p][0]\rangle)$	$H(\langle\phi_p, 1, R_i[p][1]\rangle)$

Table 3.1: Example of table VK_i with the messages signatures for process p_i

Semantic validation. The semantic validation checks if the values in the received message are congruent with the previous computation, i.e., the previously transmitted messages. This validation complements the authentication method by checking if the sender behaved correctly, i.e., it did not deviate from the algorithm.

Depending on the set of messages used to validate the received messages, the semantic validation can be implicit or explicit:

- **Implicit:** the validation is implicit when the received message does not include other messages to justify its values. In this case the receiver needs to compare it with the previous messages in the messages vector V_i.

- **Explicit:** if a process did not receive enough messages to validate the arriving message m, or if, by some reason, m appears to be invalid when comparing to the V_i, then the sender must resend m with a vector J_i containing the messages that justifies m.

Algorithm 2 starts the validation process. Messages with phase $\phi_j = 1$ are valid since there are no previous messages to compare with (lines 1 and 2). If J_j is not empty then it completes V_i with the missing messages from J_j (lines 3-5). Then it sets the counters $count_0$, $count_1$, $dCount_0$, $dCount_1$ and $dCount_\perp$ according to value $v \in \{0, 1, \perp\}$ and phase number ϕ, and counters $count_{total}$ and $dCount_{total}$ with the total number of messages with phase $\phi_j - 1$ and in the last decided phase respectively (lines 6-12). Then it executes the methods to validate each of the fields in the message (lines 13-15). Finally it returns the conjunction of the validation results (line 16).

Algorithm 2: $validate(m)$

Input: Message $m = \langle j, \phi_j, v_j, status_j, deterministic_j, r_j, J_j \rangle$
Output: A boolean value indicating if the message is valid

1 **if** $\phi_j = 1$ **then**
2   **return** $true$;
3 **if** $|J_j| > 0$ **then**
4   **forall the** $m_k = \langle k, \phi_k, v_k, status_k, deterministic_k, r_k, * \rangle \in J_j$ **do**
5         $V_i \leftarrow V_i \cup \{m_k : m_k$ is valid$\}$;
6 $count_0 = |\{\langle *, \phi, 0, *, *, *, * \rangle \in V_i : \phi = \phi_j - 1\}|$;
7 $count_1 = |\{\langle *, \phi, 1, *, *, *, * \rangle \in V_i : \phi = \phi_j - 1\}|$;
8 $count_{total} = |\{\langle *, \phi, *, *, *, *, * \rangle \in V_i : \phi = \phi_j - 1\}|$;
9 $dCount_0 = |\{\langle *, \phi, 0, *, *, *, * \rangle \in V_i : \phi =$ highest $\phi < \phi_j : \phi (mod\, 3) = 0\}|$;
10 $dCount_1 = |\{\langle *, \phi, 1, *, *, *, * \rangle \in V_i : \phi =$ highest $\phi < \phi_j : \phi (mod\, 3) = 0\}|$;
11 $dCount_\perp = |\{\langle *, \phi, \perp, *, *, *, * \rangle \in V_i : \phi =$ highest $\phi < \phi_j : \phi (mod\, 3) = 0\}|$;
12 $dCount_{total} = |\{\langle *, \phi, *, *, *, *, * \rangle \in V_i : \phi =$ highest $\phi < \phi_j : \phi (mod\, 3) = 0\}|$;
    /* Validates each field separately */;
13 $phase \leftarrow validatePhase(m, count_{total})$;
14 $status \leftarrow validateStatus(m, dCount_0, dCount_1, dCount_\perp, dCount_{total})$;
15 $value \leftarrow validateValue(m, count_0, count_1)$;
16 **return** $phase \wedge value \wedge status$;

Phase value. Algorithm 3 validates the phase value. A phase value ϕ_j requires more than $\frac{n+f}{2}$ valid messages with phase $\phi_j - 1$, otherwise there are not enough messages to justify the remaining fields of the message (lines 1 and 2).

If the phase value ϕ_j is greater than ϕ_i then the message must be treated differently. According to the system model, a process may miss necessary messages to make progress.

Algorithm 3: $validatePhase(m, count)$

```
Input: Received message m = ⟨j, φ_j, v_j, status_j, deterministic_j, r_j, J_j⟩
Input: count with the number of message with phase φ_j − 1
Output: A boolean value valid indicating if φ_j is valid
1  if count ≤ (n+f)/2 then
2  │  return false ;
3  return true;
```

To recover from this problem it must resume the protocol by jumping to the current phase. A process p_i jumps to a phase when it receives a valid message m with ϕ_j such that $\phi_j > \phi_i$ (lines 20-28). To ensure that the state maintains its consistency the received message m must carry an array J_j with enough messages to validate m. The messages from J_j cannot be validated semantically if the phase $\phi_j > \phi_i + 1$ because there are no previous messages. It is only possible to authenticate the sender and to guarantee that it generated v_j and ϕ_j. However, as long as the number of byzantine processes does not exceed f, the group of messages in J_j contains enough correct messages to validate m and update the internal state without corrupting any of the properties.

This jumping method allows processes that become disconnected for some considerable time to catch up the group, and is also a good technique to guarantee property **BC3 Termination**. As long as there is at least one correct process broadcasting the set of messages that led to the decision, a recovering process is able to decide.

Algorithm 4: $validateStatus(m, dCount_0, dCount_1, dCount_\perp, dCount_{total})$

```
Input: Received message m = ⟨j, φ_j, v_j, status_j, deterministic_j, r_j, J_j⟩
Input: dCount_0, dCount_1, dCount_⊥: the number of messages with the highest φ(mod
       3) = 0 < φ_j and values 0, 1 and ⊥, respectively
Input: dCount_total: the total number of messages received in the last phase DECIDE
Output: A boolean value valid indicating if status_j is valid
1  if status_j = decided then
2  │  if φ_j ≤ 3 then
3  │  │  return false;
4  │  if (dCount_0 ≤ (n+f)/2 ∧ v_j = 0) ∨ (dCount_1 ≤ (n+f)/2 ∧ v_j = 1) then
5  │  │  return false;
6  else                                          /* status_j = undecided */
7  │  if φ_j > 3 then
8  │  │  if dCount_total ≤ (n+f)/2 then
9  │  │  │  return false;
10 │  │  if dCount_⊥ ≤ ((n+f)/2)/2 then
11 │  │  │  return false;
12 return true;
```

Status value. Algorithm 4 validates the status value. It takes as input the message m to be validated and counters, $dCount_0$, $dCount_1$ and $dCount_\perp$ with the number of messages with values 0, 1 and $\perp$ respectively and $dCount_{total}$ with the total number of

messages in phase ϕ' such that ϕ' is the highest $DECIDE$ phase and $\phi' < \phi_j$. The status can only assume one of two values: $\{decided, undecided\}$.

- $status = decided$: messages with phase ϕ_j must pass these two conditions (lines 1-5):
  - $\phi_j > 3$ which means, in other words, that no process can decide prior to phase 3;
  - the message must carry a value v_j chosen by more than $\frac{n+f}{2}$ processes in the last decide phase (lines 4-5).
- $status = undecided$: if $\phi_j \leq 3$ then the message is valid. If $\phi_j > 3$, in order to have enough messages to validate, $dCount_{total}$ must be greater than $\frac{n+f}{2}$ (line 8 and 9). m is valid if $dCount_{\perp}$ is less or equal than $\frac{\frac{n+f}{2}}{2}$ (lines 10 and 11). Since the value v_j and the phase number ϕ_j are authenticated, there is no way a byzantine process could tamper more than $\lfloor\frac{\frac{n+f}{2}}{2}\rfloor$ proposal values.

Algorithm 5 validates the deterministic flag and the proposal value. It takes as input the message m to be validated and counters, $count_0$ and $count_1$, with the total number of messages with values 0 and 1 respectively and phase $\phi_j - 1$.

Deterministic flag. This bit indicates if the proposal value was decided deterministically or randomly. It can only assume one of two values $\{0, 1\}$. This flag is validated along side with the proposal value (lines 4-11).

Proposal value. The proposal value v_j is selected according to the phase, so its validation depends on ϕ_j.

- Phase $CONVERGE$ ($\phi_j(mod\ 3) = 1$): the validation depends if the proposal value v_j was generated deterministically (line 43 of Algorithm 1) or randomly (line 46 of Algorithm 1).
  - Deterministically: if there are no messages with phase $\phi_j - 1$ and value $v \in \{0, 1\}$ then the message is invalid (lines 4-8);
  - Randomly: there can be no message from last phase with $v \in \{0, 1\}$ for this message to be valid (lines 10 and 11).
- Phase $LOCK$ ($\phi_j(mod\ 3) = 2$): the voted value must be the one chosen by the majority. A majority in a quorum of $\frac{n+f}{2}$ is more than $\frac{\frac{n+f}{2}}{2}$. A message with value v_j is valid if there are at least $\frac{\frac{n+f}{2}}{2}$ messages with value v_j (lines 12-16).

- Phase $DECIDE$ ($\phi_j (mod\ 3) = 0$): the validation depends if the value is in $\{0, 1\}$ or if it is $\perp$.
  - $v_j \in \{0, 1\}$: the message is valid if there are more than $\frac{n+f}{2}$ messages with the same value $v_j \in \{0, 1\}$ (lines 18-20).
  - $v_j = \perp$: the message is invalid if all messages have the same value $v \neq \perp$. (lines 22 and 23).

Algorithm 5: $validateValue(m, count_0, count_1)$

```
   Input: Received message m = ⟨j, φ_j, v_j, status_j, deterministic_j, r_j, J_j⟩
   Input: count_0, count_1 with the number of message with phase φ_j − 1 and values 0, 1
   Output: A boolean value indicating if the v_j and deterministic_j are valid
                                    /* phase CONVERGE */;
 1 if φ_j(mod 3) = 1 then
 2     if (v_j = ⊥) then
 3         return false;
 4     if deterministic_j = 1 then
 5         if count_0 = 0 ∧ count_1 = 0 then
 6             return false;
 7         if (count_0 = 0 ∧ v_j = 0) ∨ (count_1 = 0 ∧ v_j = 1) then
 8             return false;
 9     else                                              /* deterministic_j = 0 */
10         if count_0 ≥ 1 ∨ count_1 ≥ 1 then
11             return false;
                                    /* phase LOCK */;
12 if φ_j(mod 3) = 2 then
13     if v_j = ⊥ then
14         return false;
15     if (count_1 ≤ ((n+f)/2)/2 ∧ v_j = 1) ∨ (count_0 ≤ ((n+f)/2)/2 ∧ v_j = 0) then
16         return false;
                                    /* phase DECIDE */;
17 if φ_j(mod 3) = 0 then
18     if v_j ∈ {0, 1} then
19         if (v_j = 0 ∧ count_0 ≤ (n+f)/2) ∨ (v_j = 1 ∧ count_1 ≤ (n+f)/2) then
20             return false;
21     else                                              /* v_j = ⊥ */
22         if count_0 = 0 ∨ count_1 = 0 then
23             return false;
24 return true;
```

3.2.2 Multivalued Consensus

A multivalued consensus protocol allows processes to agree on an arbitrary value. The proposed algorithm solves the multivalued consensus problem in the presence of byzantine faults. It was adapted from [6] to ensure that the protocol can be used in ad hoc

networks. The multivalued consensus protocol provides the following properties:

- **MVC1 Validity 1.** If all correct processes propose the same value v, then any correct process that decides, decides v.
- **MVC1 Validity 2.** If a correct process decides v, then v was proposed by some process or $v = \perp$.
- **MVC1 Validity 3.** If a value v is proposed only by corrupt processes, then no correct process that decides, decides v.
- **MVC1 Agreement.** No two correct processes decide differently.
- **MVC1 Termination.** Every correct process eventually decides.

Algorithm

Algorithm 6 solves the multivalued consensus problem. Each process p_i has an internal state with: the current phase number ϕ, the value v_i and a vector with the received messages V_i. The algorithm works in three independent tasks.

Task T1 is triggered by a local timer (line 5). It broadcasts the internal state of p_i in a message m in the form $\langle i, \phi, v_i, J_i \rangle$, being i the identifier of the sender, ϕ the current phase number and J_i the array of messages that justifies the proposal of p_i (line 6).

Taskt T2 handles received messages. When a message $m = \langle j, \phi_j, v_i, J_j \rangle$ is received (line 8) it is validated and if valid it is stored in V_i (line 9).

Task T3 updates the internal state depending on ϕ:

- $\phi = 0$: once it collects more than $\frac{n+f}{2}$ messages with the same phase $\phi = 0$ it sets the variable maj with the most voted value (line 12). Then, if there are more than f messages with the same value $v = maj$ (line 13), it sets v_i with maj (line 14). This way it is guaranteed that at least one correct process voted for maj. Finally, the phase number ϕ is incremented.
- $\phi = 1$: it waits until there are more than $\frac{n+f}{2}$ messages with the same phase number $\phi = 1$ (line 16). Then, if there are more than $\frac{n+f}{2}$ messages with the same value v and phase value $\phi = 1$, v_i is updated to v and variable $propose$ is set to 1 (lines 17-19). Otherwise v_i is set to $\perp$ and $propose$ to 0 (lines 20-22). The $propose$ is then used as proposal value of binary consensus (line 23). The decided value of multivalued consensus depends on the decision of binary consensus:
  - binary consensus decided 1: if v_i is $\perp$ then p_i must wait until one of the other processes sends the decided value in order to update v_i with v (lines 24-27);
  - binary consensus decided 0: v_i is set to $\perp$ (lines 28-29).

The phase number ϕ is incremented to prevent the protocol from continuing execution (line 30) and the decided value v_i is returned;

- $\phi = 2$: in background, Task T1 continues to broadcast the decided value, to ensure that the remaining processes can also terminate.

Algorithm 6: Multivalued consensus protocol

Input: Instance id of this execution
Input: Initial proposal value $proposal_i$
Output: Decision value v_i

1. $\phi \leftarrow 0$;
2. $v_i \leftarrow \langle proposal_i, sig_i \rangle$;
3. $V_i \leftarrow 0$;
4. **Task T1**
5. **when** *local clock tick* **:**
6. $broadcast(\langle i, \phi, v_i, J_i \rangle)$;
7. **Task T2**
8. **when** $m = \langle j, \phi_j, v_i, J_j \rangle$ *is received* **:**
9. $V_i \leftarrow V_i \cup \{m : m\ is\ valid\}$;
10. **Task T3**
11. **when** $|\{*, \phi, *, *\} \in V_i : \phi = 0| > \frac{n+f}{2}$ **:**
12. $maj \leftarrow majority\ value\ in\ V_i\ for\ \phi = 0$;
13.   **if** $|\{\langle *, \phi, v, * \rangle \in V_i : \phi = 0 \wedge v = maj\}| > f$ **then**
14. $v_i \leftarrow maj$;
15. $\phi \leftarrow 1$;
16. **when** $|\{\langle *, \phi, *, * \rangle \in V_i : \phi = 1\}| > \frac{n+f}{2}$ **:**
17.   **if** $|\{\langle *, \phi, v, * \rangle \in V_i : \phi = 1\}| > \frac{n+f}{2}$ **then**
18. $v_i \leftarrow v$;
19. $propose \leftarrow 1$;
20.   **else**
21. $v_i \leftarrow \perp$;
22. $propose \leftarrow 0$;
23. $result \leftarrow BConsensus(propose)$;
24.   **if** $result = 1$ **then**
25.   **if** $v_i = \perp$ **then**
26.   **wait until** (it is delivered $\langle *, \phi, v, * \rangle \in V_i : \phi = 2$)
27. $v_i \leftarrow v$;
28.   **else** /* $result = 0$ */
29. $v_i \leftarrow \perp$;
30. $\phi_i \leftarrow 2$;
31. **return** v_i;

For the multivalued consensus protocol we will not describe the validation procedure as it is relatively simple. This occurs because values are signed (see line 2), and therefore the malicious processes cannot tamper with the values from correct processes.

3.2.3 Vector Consensus

In the vector consensus problem each process p_i proposes a value v_i and all the processes decide upon a vector V of size n. For every correct process p_i, $V[i]$ contains v_i proposed

by p_i or $\perp$. The vector consensus protocol provides the following properties:

- **VC1 Vector Validity.** every correct process that decides, decides on a vector V of size n:
  - $\forall_{pi}$ if p_i is correct, then either $V[i]$ is the value proposed by p_i or $\perp$;
  - at least $(f+1)$ elements of V were proposed by correct processes.
- **VC2 Agreement.** no two correct processes decide differently.
- **VC3 Termination.** every correct process eventually decides.

Algorithm

Algorithm 7 solves the vector consensus problem. Every process p_i has an internal state with the value v_i, a double dimension array $array_i$ with pairs with every received proposal and its correspondent signature, and the current round number r. The value v_i is initialized with the $proposal_i$ and its signature. The protocol works in three tasks.

Algorithm 7: Vector consensus protocol

```
   Input: Initial proposal value proposal_i
   Output: Decision vector V_i
1  r ← 0;
2  array_i ← ((⊥, ..., ⊥), ..., (⊥, ..., ⊥));
3  array_i[i][i] ← ⟨proposal_i, sig_i⟩;
4  Task T1
5  when local clock tick :
6      broadcast(⟨i, array_i[i]⟩);
7  Task T2
8  when m = ⟨j, vector_j⟩ is received :
9      count ← 0;
10     forall the (v_k, sig_k) ∈ vector_j do
11         if v_k ≠ ⊥ then
12             if verifySig(k, v_k, sig_k) = invalid then
13                 break to when;
14             count ← count + 1;
15     if count = 2f + 1 then
16         array_i[j] ← vector_j;
17     if |{v_k = array_i[i][k] : k ∈ {0, ..., n − 1} ∧ v_k ≠ ⊥}| < 2f + 1 then
18         array_i[i][j] ← vector_j[j];
19 Task T3
20 when ∃vector ∈ array_i : |{v_k ∈ vector : k ∈ {0, ..., n − 1} ∧ v_k ≠ ⊥}| = 2f + 1 :
21     repeat
22         vector ← selectValue(r(mod n), array_i);
23         V_i ← MConsensus(r, vector);
24         r ← r + 1;
25     until V_i ≠ ⊥;
26 return V_i;
```

Algorithm 8: $selectValue(j, array_i)$

```
   Input: j
   Input: array_i
1  value ← ⊥;
2  for index ∈ {0, .., n − 1} do
3      k ← (index + j)(mod n);
4      if |{v_l = array_i[k][l] : l ∈ {0, ..., n − 1} ∧ v_l ≠ ⊥}| = 2f + 1 then
5          return array_i[k];
6  return ⊥;
```

Task T1 is triggered by a local clock (line 5) to broadcast a message m in the form $\langle i, array_i[i] \rangle$ (line 6). The process sends $array_i[i]$ which corresponds to the single dimension array (or vector) with the valid received proposal values by p_i.

Task T2 handles received messages. When p_i receives a message $m = \langle j, vector_j \rangle$ sent by p_j (line 8), it verifies if the signature of each value is valid (lines 9-12). Function $verifySig(k, v_k, sig_k)$ verifies the digital signature of v_k by using the public key pu_k. If at least one value of the $vector_j$ is invalid then the message is not used (line 13). As the signatures are verified, the variable $count$ (line 9) is incremented for every value different than $\perp$ in $vector_j$ (lines 11 and 14). If the number of values in $vector_j$ different from $\perp$ is equal to $2f + 1$ then the $array_i$ is updated to include the received $vector_j$ in the jth position (lines 15 and 16). If the number of entries in $array_i[i]$ with a value different from $\perp$ is less than $2f + 1$, then $array_i$ is updated to include the proposal of p_j (lines 17 and 18).

Task T3 executes the required multivalued consensus executions to achieve a vector consensus. It is triggered when there is a vector with $2f+1$ values different than $\perp$. Then it enters in a loop and in each iteration it invokes the function $selectValue()$, implemented in Algorithm 8, and uses its result to execute a multivalued consensus (lines 22 and 23). Then it increments the round number r (line 23). When the result of the multivalued consensus is different than $\perp$ it returns the decided vector (line 26).

Algorithm 8 consists in an auxiliar function used by vector consensus. It runs the $array_i$ from j to $j - 1$ (similar to a circular array) (lines 2-5) until it finds a k such that the number of values in $array_i[k]$ different from $\perp$ are equal to $2f + 1$. If it does then it returns that vector, otherwise it returns $\perp$.

Chapter 4

Implementation and Evaluation

This chapter describes the implementation and evaluation of the protocol stack. Section 4.1 explains the design and implementation of the protocol stack. Section 4.2 describes the simulation configuration and specific details of using the network simulator NS-3 [10]. Section 4.3 describes the implementation of the protocol stack in the mobile platform Android[20]. Section 4.4 presents the execution environment and the results obtained with the simulations.

4.1 Design and Implementation of the Protocol Stack

There are several aspects that had to be taken into account when designing the stack, which was implemented as a network library. The library provides several primitives for consensus and must interfere as less as possible with the applications.

4.1.1 Single-threaded and Multi-threaded Mode

There are two possible approaches to implement the protocol stack related to the execution flow: single-threaded and multi-thread. Both approaches have their advantages and disadvantages. In the single-threaded approach, the protocol stack runs in an independent thread from the application and every protocol in the stack executes in the same thread. This approach has the advantage that it does not need synchronization techniques to keep the state consistent, but it may have a performance degradation because only a protocol may execute at a time. The multi-threaded approach allows for each protocol instance to run in a separated thread concurrently, but this approach requires a synchronization mechanism to keep the library state consistent.

After analyzing the protocols to understand which approach would be advantageous, the single-threaded was chosen. The binary consensus, at the base of the protocol stack, provides a consensus primitive for the multivalued consensus which in turn offers a primitive for vector consensus. Both multivalued and vector consensus, which are dependent from the lower protocols, do not take significant advantage from multiple threads because

they only make progress once a decision is reached. Also the synchronization mechanisms required to keep the system consistent would potentially create overheads. Although the stack runs in single-threaded mode, each protocol offers callbacks to the application level so it can run concurrent instances of consensus without blocking.

The consensus protocols periodically broadcast messages with the state of the processes. In order to not interfere with the execution flow of the application, the protocol stack runs in two independent threads from the application thread. One thread deals with message transmission and the other thread deals with the consensus protocol.

Communication Thread

The communication thread carries out message transmission and it is independent from the protocol stack. It works the following way: when the protocol stack sends a message, it is stored in a queue. Periodically the communication thread consults the queue and if it is not empty it broadcasts the entire queue. Since none of the protocols require point-to-point channels, every message is broadcast, this way less transmissions are made saving the battery power of the mobile device.

When the communication thread receives a message it saves it in a list and waits for the protocol stack (or the application) to check for received messages. The protocol stack does not use callbacks because they may corrupt the consistency of the internal state of the protocols, instead it uses the timer that triggers a check for received messages.

Protocols Thread

The protocols run in a specific thread that is independent from the application thread. This way the application runs without interruption and once it needs to initiate a consensus execution a new thread is created. The consensus starts and when a decision is reached it returns the decided value to the application.

The application, however, may need to execute several consensus instances simultaneously without waiting for decisions. The protocols offer non-blocking calls to execute consensus, and, for every consensus instance is initiated a new thread. The application may use a callback to obtain the consensus decision or it can pool the consensus instance to determine if a decision was reached.

Messages. The communication thread groups the messages from several protocols and protocol instances in a queue and sends them at once. This requires a mechanism to manage the delivery of the messages. Each message, independently of the protocol, has two IDs: one identifies the protocol and the other the instance. When a group of messages arrive at a node, the communication layer groups the received queue by their protocol and instance IDs.

The IDs allow the communication thread to deal with messages from protocol instances that are not running locally (e.g., because they have not started yet). These messages are queued for later processing.

4.1.2 Protocol Context

Each protocol allows several instances to be executed. Each instance is identified by a unique ID and the ID of the protocol. Every consensus instance is saved in a per protocol list.

The protocol stack tolerates arbitrary faults. Therefore, process may lose communication for some time and recover later but meanwhile the remaining process reached a decision. It is essential that the correct process save the decisions made so that when a process recovers it is able to learn the decided value. The library saves every results of consensus executions in a list for as long as the application is running (and while there is space in memory).

4.1.3 Library Architecture

The classes of the library are organized by packages representing a logic part of the stack.

The package - pt.fcul.consensus.network - includes the classes related to the network tasks:

- **Communication**: Contains all the methods and attributes necessary to establish a connection between peers and do broadcast and receive messages;

- **CommunicationCallback**: Interface to be implemented by classes to receive messages;

- **Message**: Generic type of message. Every message exchanged by the stack of protocols must extend this type;

- **MessageBinaryConsensus**: Specific type of message for BinaryConsensus protocol

- **MessageMultivaluedConsensus**: Specific type of message for Multivalued Consensus protocol

- **MessageVectorConsensus**: Specific type of message for Vector consensus protocol.

The most important package - pt.fcul.consensus.protocols - includes the classes which implement the protocols and the interfaces needed to implement the callbacks.

- **BinaryConsensus**: Implements the binary consensus protocol described in Section 3.2.1.

- **BinaryConsensusCallback**: An interface to be implemented by a class to deal with the result of the consensus execution.

- **MultivaluedConsensus**: Implements the multivalued consensus protocol explained in Section 3.2.2.

- **MultivaluedConsensusCallback**: An interface to be implemented by a class to deal with the result of the multivalued consensus.

- **VectorConsensus**: Implements the vector consensus protocol explained in Section 3.2.3.

- **VectorConsensusCallback**:An interface to be implemented by the application to deal with the results of vector consensus executions.

The pt.fcul.consensus.utils package has the classes with utilities and constants of the library:

- **Constants**: contains all the constant variables commons to all project;

- **FileUtils**: a set of file manipulation methods.

Figure 4.1 presents the class diagram of the network library. The Communication class is instantiated by the Consensus class and is used by all three consensus protocols. Using only one instance of the Communication class allows a better use of the wi-fi interface of the mobile device. Each consensus protocol has its own message type that extends the main message. The consensus class is the class that saves the internal state of the protocol stack and manages the instances of the protocols.

Figure 4.2 represents a vector consensus execution started by the application. This example illustrates a complete use of the protocol stack. As we can see, every interaction made is unilateral. In the example it is used the blocking proposal call for every consensus protocol. This call does not return until a decision is reached and is the one used internally in the stack.

4.1.4 Application Programming Interface

The network library offers a set of operations to the application level. The Application Programming Interface (API) of both the simulation and the android network library contains the following methods.

Figure 4.1: Class diagram of the android network library

Communication

```
void broadcast(Message message)
```

Broadcasts a messages through Wi-Fi direct [2], in case of android, and in Wi-Fi 802.11g in case of the simulation. This broadcast primitive does not give any acknowledgement about the delivery of the message and does not offer any guarantee that the message is delivered.

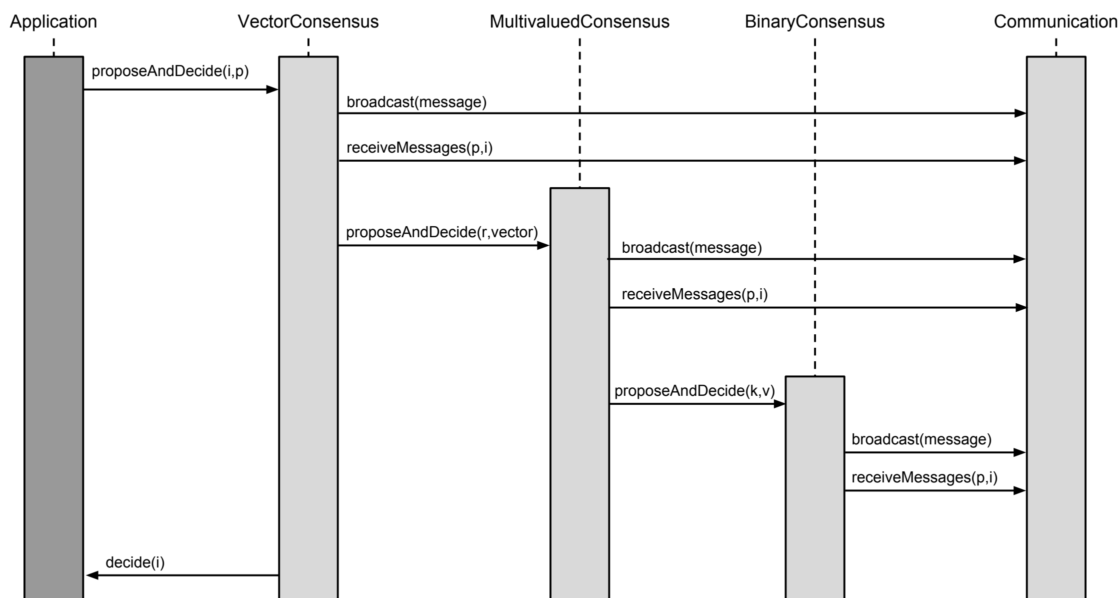


Figure 4.2: Sequence diagram of a vector consensus execution

Parameters:

- `message`: the message to broadcast.

```
ArrayList<Message> receiveMessages(int protocol, String
   instance)
```

Checks for received messages. If a callback is not set, the protocol must call this method to receive messages.

Parameters:

- `protocol`: the protocol that is the receiver of the message;
- `instance`: the instance of the protocol.

Returns:

- An `ArrayList<Message>`: containing the received messages of the protocol instance.

```
void setCaller(CommunicationCallback caller)
```

Sets a callback (an implementation of `CommunicationCallback`) to handle received messages. If no callback is set, the received messages are saved in a list and can be obtained by the `receiveMessages()` method.

Parameters:

- `caller`: the CommunicationCallback invoked that will be triggered when a message arrives.

Binary Consensus

```
void propose(String instance, int value)
```

Non-blocking proposal method. The result of consensus execution can be retrieve through `getDecision()`. It takes as input:

Parameters:

- `instance`: a String value to identify the instance of this consensus execution;
- `value`: the proposal value $v_i \in \{0, 1\}$.

```
int proposeAndDecide(String instance, int value)
```

Blocking proposal call. This method does not return until a decision is reached. Once a decision is reached it returns the decided value $decision_i \in \{0, 1\}$.

Parameters:

- `instance`: the instance of this consensus execution;
- `value`: the proposal value $v_i \in \{0, 1\}$.

Returns:

- `decision`: the decided value $decision_i \in \{0, 1\}$.

```
int decide(String instance)
```

Method used to consult the result of a previous consensus execution. For a given instance `instance` it returns the decided value $decision_i \in \{0, 1\}$ or -1 if the process did not yet decided.

Parameters:

- `instance`: the identifier of the consensus instance to consult the decided value.

Returns:

- `decision`: the decided value $decision_i$ of the binary consensus execution.

Multivalued Consensus

```
void propose(String instance, String value)
```

Non-blocking proposal method. The result of consensus execution can be retrieve through `getDecision()`.

Parameters:

- `instance`: the instance of this consensus execution;
- `value`: the proposal value $proposal_i$, the empty string (" ") is used to refer to the $\perp$ value.

```
String proposeAndDecide(String instance, String value)
```

Blocking proposal call. This method does not return until a decision is reached.

Parameters:

- `instance`: the instance of this consensus execution;
- `value`: the proposal value v_i.

Returns:

- `decision`: the decided value $decision_i$. If `decision` equals an empty string (" ") then the decided value was $\perp$.

```
String decide(String instance)
```

Method used to collect a decided value. For a giver instance `i` of a multivalued consensus protocol it returns the decided value $v \geq 0$, -1 for $\perp$ or -2 if it has not decided yet.

Parameters:

- `instance`: the identifier of the instance.

Returns:

- `decision`: the decided value $decision_i$. If `decision` equals an empty string (" ") then the decided value was $\perp$.

Vector Consensus

```
void propose(String instance, String value)
```

Non-blocking proposal method. The result of consensus execution can be retrieve through `getDecision()`.

Parameters:

- `instance`: the instance of this consensus execution;
- `value`: the proposal value.

```
String[] proposeAndDecide(String instance, String value)
```

Blocking proposal call. Until a decision is reached this method does not return. When they reach a decision it returns the decided vector V. It takes as input:

Parameters:

- `instance`: the instance of the protocol
- `value`: the proposal value $v_i \geq 0$ of this process.

Returns:

- `decision`: the decided vector V.

```
String[] decide(String instance)
```

Method used to collect a decided value. For a giver instance `instance` of a vector consensus protocol it returns the decided vector V.

Parameters:

- `instance`: the String with the instance of the consensus execution.

Returns:

- `decision`: the decided vector V.

4.2 NS-3 Simulation

The protocol stack was implemented in the network simulator NS-3 [10]. The simulation can be configured with the following parameters:

- **n**: the total number of processes n executing the protocol stack;
- **diverge**: if true then each process proposes a different value, otherwise every process proposes the same value;
- **protocol**: the top protocol of the stack being tested. It can be: binary, multivalued or vector;
- **grid**: the distribution of the processes: if true they are set in a grid otherwise they assume a random distribution;
- **byzantine_identity**: if true, byzantine processes try to impersonate correct processes;
- **byzantine_value**: if true, byzantine processes send a different value than they should;
- **byzantine_status**: if true, byzantine processes send incorrect status value;
- **byzantine_phase**: if true, byzantine processes send incorrect phase value;
- **num_sims**: the number of simulations being executed with this configuration.

The simulation is composed by the following additional classes:

- **ProtocolStackSimulator:** main class executed when the simulation starts. It installs the network, sets-up the protocols, sets the simulation configuration (based on the configuration file) and initiates the execution;
- **ProtocolStack:** Depending on the configuration file, this class instantiates the protocols to be tested;

- **Communication:** Every process instantiates this class to use a simple broadcast primitive. This class implements some mechanisms to optimize the network usage, such as: saving the messages to be sent in a list to be periodically sent, instead of sending the messages immediately.

- **ProtocolStackHelper:** Auxiliary class that installs the ProtocolStack application in the processes and passes the arguments from the ProtocolStackSimulator to the ProtocolStack instance;

- **ReadConfig:** Reads the configuration file of the simulation and keeps an internal state with the configuration parameters;

- **Sha256:** Implements the hashing algorithm SHA-256[16] used to authenticate the proposal value v and the phase number ϕ of the binary consensus protocol.

4.3 Android Implementation

The protocol stack was also implemented in android to be used as a library that provides consensus primitives for distributed applications. This library was implemented for Android 4.0 (API level 14 or later) because it uses Wi-fi direct [2] to broadcast the messages. To test the library, an application was built that only allows to execute consensus of each kind.

The package - pt.fcul.consensus - is only used in the android library and it includes the classes related to the application:

- **Consensus**: extends the Application class of android. It saves all the attributes of the protocols stack such as the number of processes n, the groupId, the communication instance, the connection status, the communication port, the currently connected peers and the consensus results;

- **ConfigureActivity**: Handles the layout of the Configuration activity;

- **MainActivity**: Class related with the first activity presented to the user. Contains a menu with the options to execute consensus and to configure the application.

- **NetworkActivity**: Class related with the activity to manage the network status of the device;

- **BinaryConsensusActivity**: Class related to the activity that starts the execution of a binary consensus instance;

- **MultivaluedConsensusActivity**: Class related to the activity that starts the execution of a multivalued consensus instance;

- **VectorConsensusActivity**: Class related to the activity that starts the execution of a vector consensus instance;

4.4 Simulation Results

Several simulations were executed with each protocol. Different data was collected depending on the configuration of the stack. All the results are organized in function of the number of processes. This exhibits how the performance changes when the number of processes increases, allowing the evaluation of the scalability of the stack.

4.4.1 Experimental Environment

An ad hoc wireless network with the communication protocol 802.11g was configured without the presence of a centralized unit (access point). The protocol runs as expected in a real world scenario, fully distributed. Depending on the distribution of the mobile devices the arrangement can be:

- **grid:** the processes are located in a grid. The distance between lines and columns is 1m. This scenario describes the execution of the consensus protocol in a totally crowded classroom where the users holding their mobile devices are near each other and do not move during the execution.

- **random:** the processes move randomly across a rectangular room with 50x50m. This scenario describes the execution of the consensus protocol when the users are moving, e.g, in the coffee break of a conference, walking randomly across the room and eventually stopping to talk to each other.

Depending on the proposed values the executions can be:

- **unanimous:** if every correct process proposes the same value ;

- **divergent:** the worst case of a consensus execution. In the binary consensus half of the processes propose 0 and the other half propose 1. In the multivalued and vector consensus protocols each process proposes a different value.

Depending on the fault mode the simulations can be:

- **fault-free:** every process behaves as it's expected to, i.e., each process executes the algorithm from beginning to the end and does not fail during the execution. Although there are no kind of process faults during the execution, the channels are not reliable, simulating a wireless network;

- **byzantine:** in this case $f = \lfloor \frac{n-1}{3} \rfloor$ processes will assume a byzantine behavior that can be one of the following:

– **byzantine value:** byzantine processes vote on different values than they are suppose to in order to delay or even fail the agreement.

– **byzantine status**: byzantine processes change the status flag that indicates if the process has already decided in order to make the correct process decide earlier or to delay the decision;

– **byzantine phase**: byzantine processes send a different phase value to make the correct processes jump phases and get stuck in an inconsistent state;

– **byzantine identity**: the id value of the sender is modified by byzantine processes to assume the identity of correct processes.

The timeout value used to trigger the periodic tasks was the number of processes n (in milliseconds). According to [17], this value gives the best performance results.

The data collected for all three protocols was:

- **latency**: is the duration it takes from the moment the first process starts a consensus execution until the last process decides. The latency reveals if the protocol is viable in the defined system model. It is collected through the start and end time recorded by the processes;

- **number of rounds**: all three protocols make progress in rounds (phases in the case of binary and multivalued consensus). This value allows the evaluation of how long it takes for the protocol to converge to the decided value;

- **number of sent and received messages**: evaluates the impact of the protocol stack in the network.

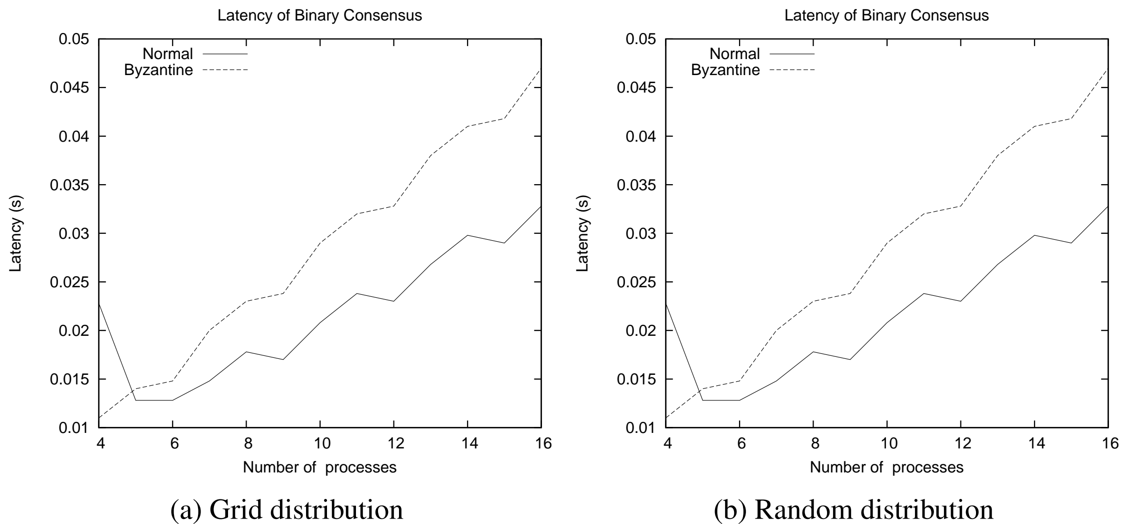
4.4.2 Binary Consensus

This section includes the simulations results of the binary consensus. During the simulation were collected the values of the latency, number of phases and amount of messages exchanged.

Latency

The following figures present the average latency of 10 instances with a variation on the number of processes from 4 to 16 in several scenarios of executions. In each scenario both the grid and the random distributions are presented, respectively, side by side.

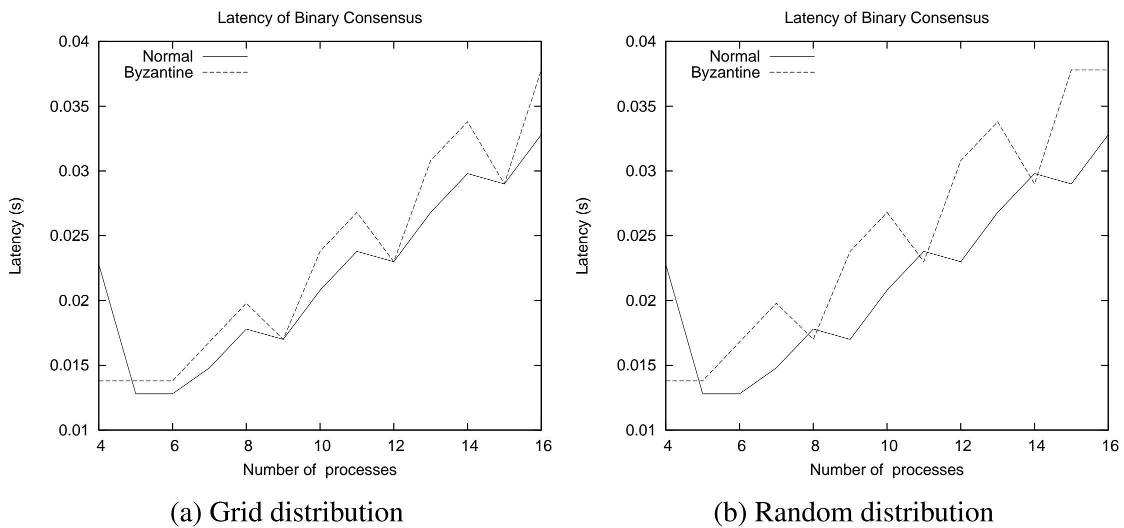
Figure 4.3 presents the latency of binary consensus when processes propose a divergent value. Results are displayed for fault-free runs and with byzantine faults that affect the identity of the sender of the messages. The results are very identical in both distributions. The byzantine mode has a worse performance than the fault-free scenario.



(a) Grid distribution (b) Random distribution

Figure 4.3: Latency of binary consensus with divergent values; byzantine faults affect the identity of the sender

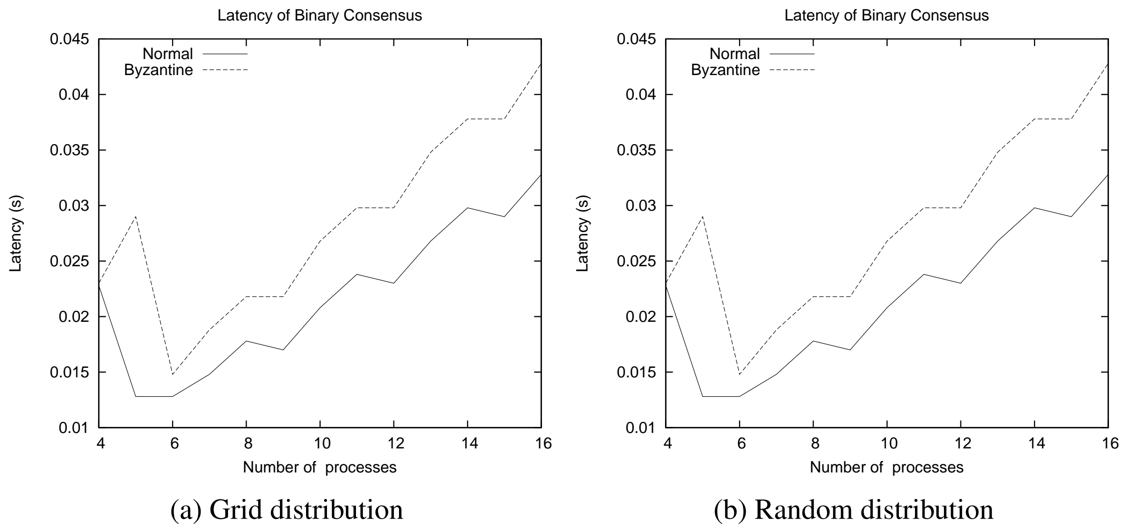
In Figure 4.4, processes propose divergent values, byzantine faults occur in the phase value. In this case the grid distribution has a slightly better performance than the random distribution and there are a few cases where the byzantine mode has a better latency than the normal mode.



(a) Grid distribution (b) Random distribution

Figure 4.4: Latency of binary consensus with divergent values; byzantine faults affect the phase number

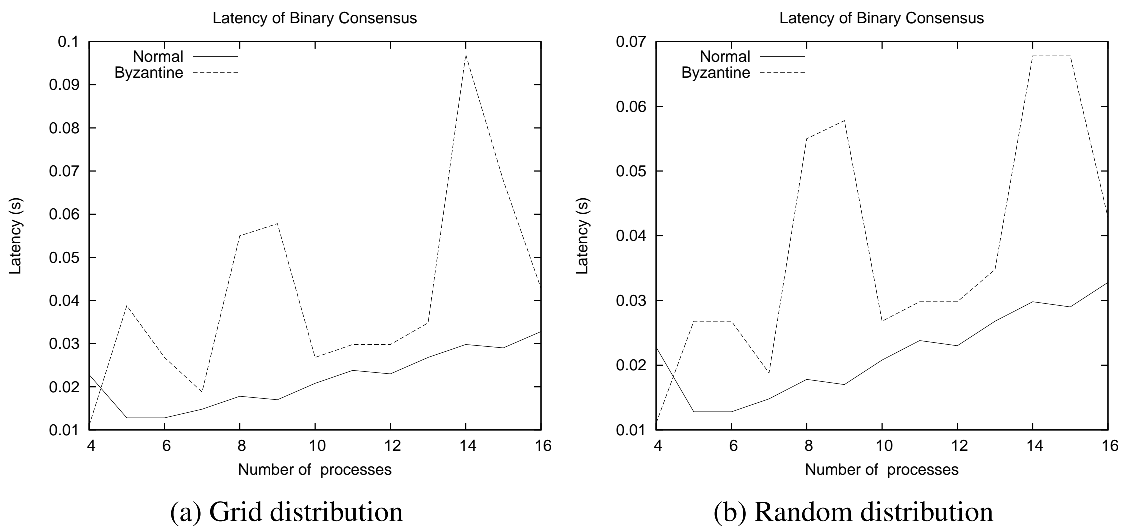
Figure 4.5 presents the scenario with divergent proposals and byzantine status faults. The results in both grid and random distributions are identical and the byzantine case reveals the worst results in every case.



(a) Grid distribution (b) Random distribution

Figure 4.5: Latency of binary consensus with divergent values; byzantine faults affect the status value

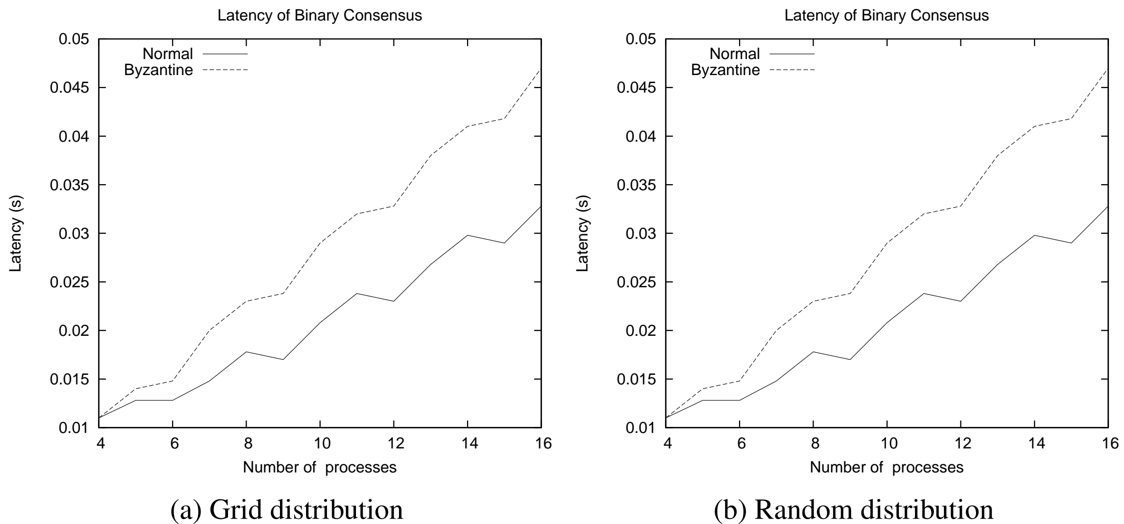
The byzantine value attack with divergent proposals is presented in Figure 4.6. This results are different in both grid and random distributions and present the greatest amplitude of values for the byzantine case, which is also the worst case.



(a) Grid distribution (b) Random distribution

Figure 4.6: Latency of binary consensus with divergent values; byzantine faults affect the proposal value

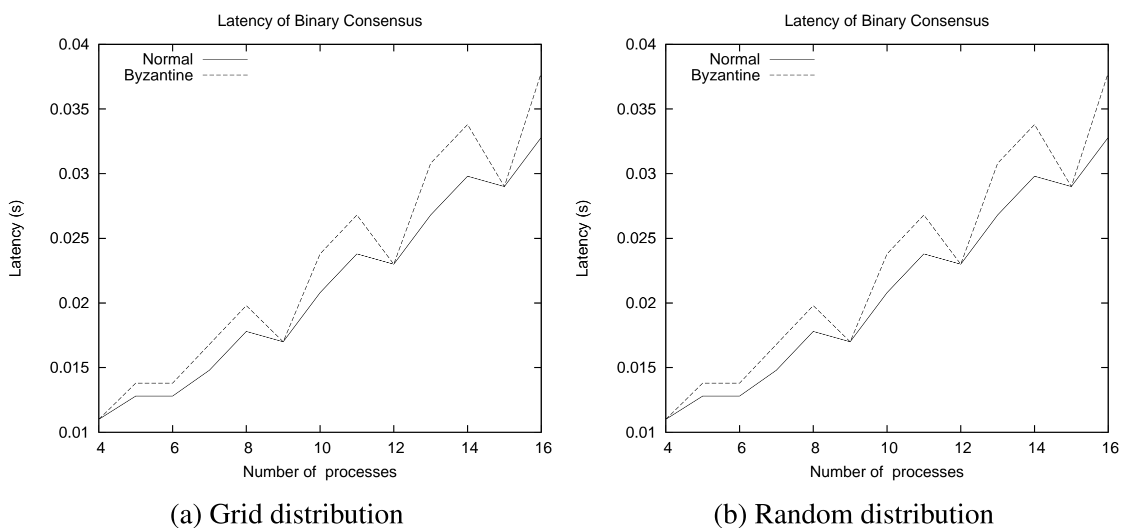
When the proposals are unanimous the results are better. Figure 4.7 presents the scenario where byzantine faults occur to the identity of the sender of the messages. The fault-free case is better and the difference between both cases increases as the number of processes increases. The results in both the grid and random distributions are identical.



(a) Grid distribution (b) Random distribution

Figure 4.7: Latency of binary consensus with unanimous values; byzantine faults affect the identity of the sender

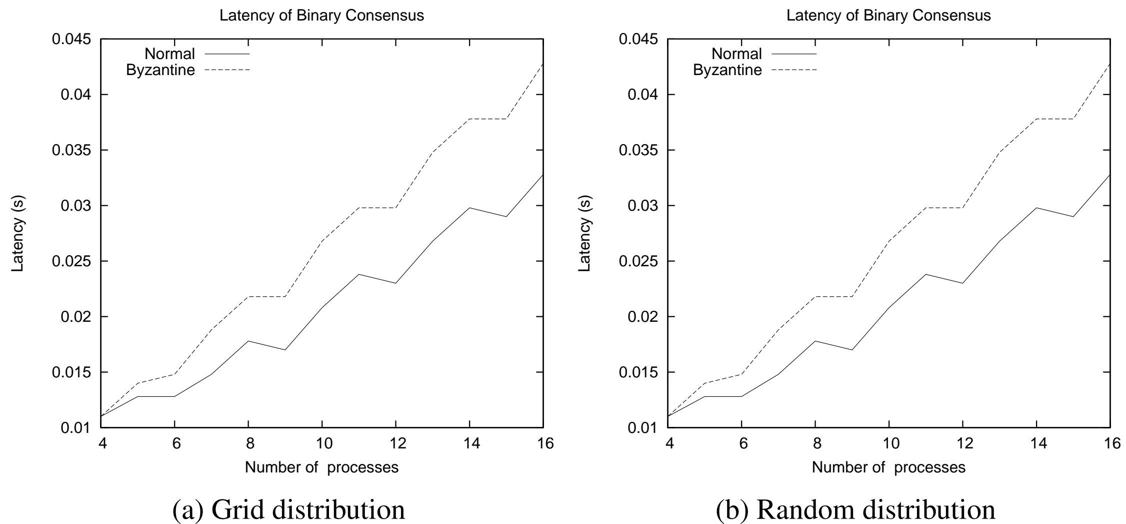
Figure 4.8 presents the byzantine phase scenario with unanimous proposals. The latency in both the fault-free and the byzantine scenarios is closer and in some cases it is equal. Once again, the physical distribution of the processes did not affect the performance.



(a) Grid distribution (b) Random distribution

Figure 4.8: Latency of binary consensus with unanimous values; byzantine faults affect the phase value

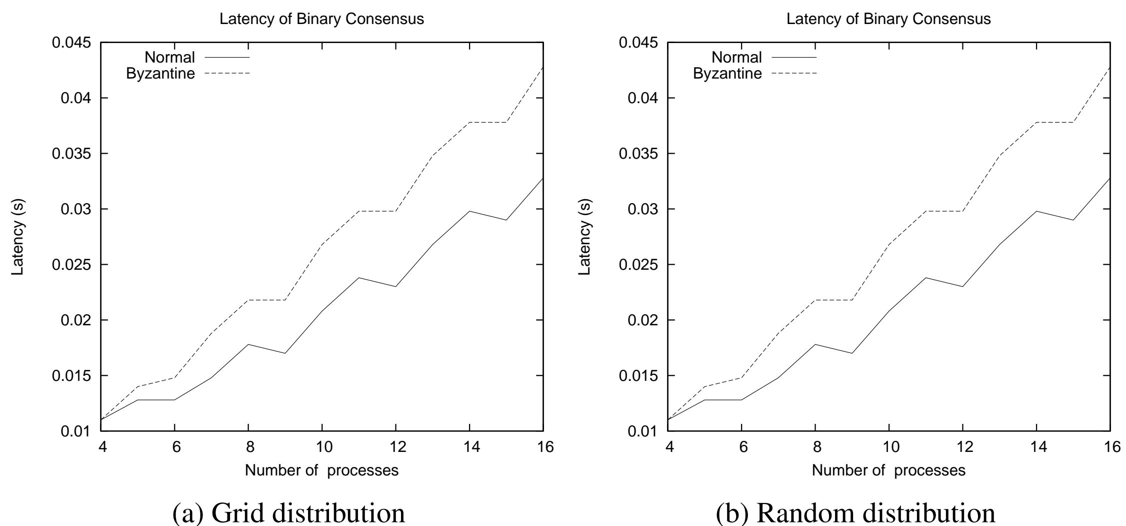
Figure 4.9 presents scenario with unanimous proposals and byzantine attacks to the status value. The latency of the byzantine mode is greater in every case and the difference between both modes increases with the number of processes.



(a) Grid distribution (b) Random distribution

Figure 4.9: Latency of binary consensus with unanimous values; byzantine faults affect the status value

Figure 4.10 presents the scenario with unanimous proposals and byzantine value attacks. The byzantine mode reveals greater latency values in every case. The difference between both modes increases with the number of processes.



(a) Grid distribution (b) Random distribution

Figure 4.10: Latency of binary consensus with unanimous values; byzantine faults affect the proposal value

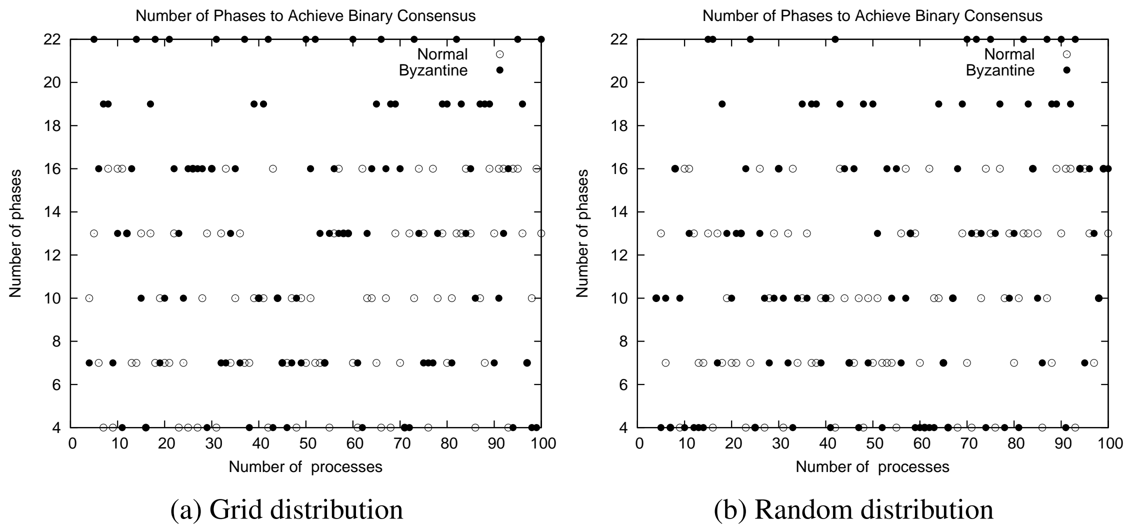
Number of phases

The binary consensus protocol progresses in rounds of three phases. The number of phases can be a measure of the performance of the algorithm since it shows how processes converge to the decided value when the number of nodes increases. This value was collected by the last process to terminate the execution of consensus by saving the phase value in the data file once it decided.

To collect the total number of phases to achieve consensus, simulations from 4 processes to 100 were performed. In the first experiment the proposal values of all processes

was 1. In every simulation every process achieved consensus in 4 phases (they decided at phase 3 and incremented before returning the decided value).

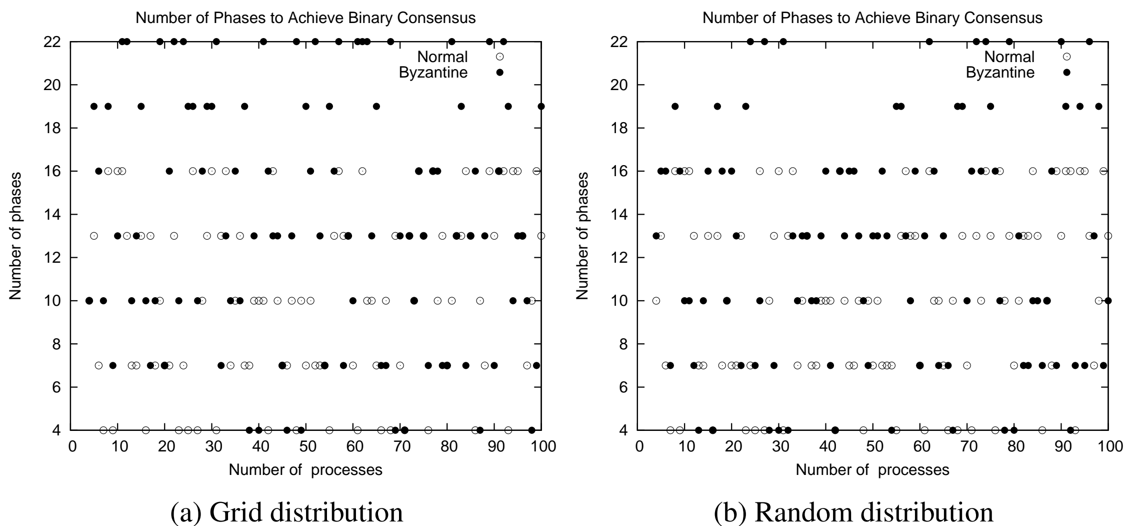
In the second experiment the proposal value was divergent. In this scenario the total number of phases needed to reach consensus varied from 4 to 22, but this value did not reveal a tendency to grow as the number of processes increased. The following Figures reveal the number of phases to achieve consensus when the proposal value is divergent.



(a) Grid distribution (b) Random distribution

Figure 4.11: Number of phases of binary consensus with divergent values; byzantine faults affect the identity of the sender

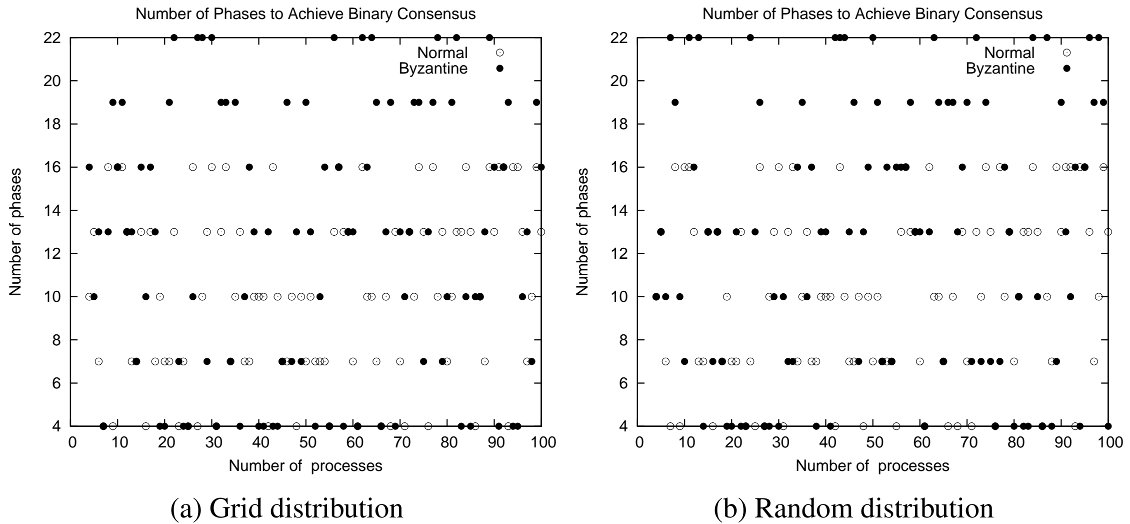
Figure 4.11 presents the number of phases to achieve consensus when byzantine faults affect the identity of the sender of the messages. The results are very identical in both distributions. The byzantine executions have a worse performance then the fault-free scenario. The fault-free modes never exceeds 16 phases, the byzantine mode reaches 22 phases.



(a) Grid distribution (b) Random distribution

Figure 4.12: Number of phases of binary consensus with divergent values; byzantine faults affect the phase value

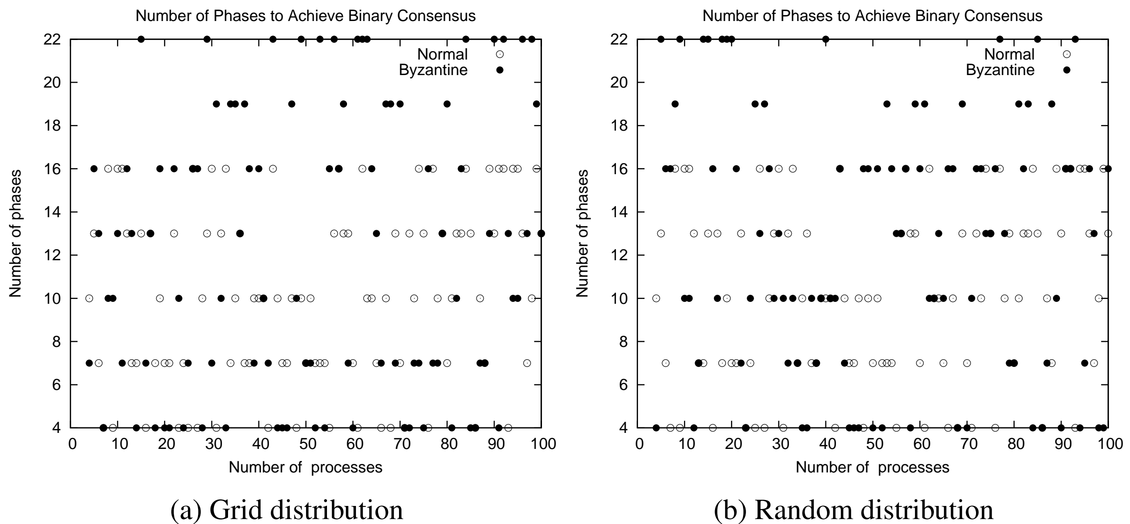
In Figure 4.12 byzantine faults occur to the phase value. In this case the random distribution has a slightly better performance than the grid distribution and there are some cases when the byzantine needs less phases to achieve consensus.



(a) Grid distribution (b) Random distribution

Figure 4.13: Number of phases of binary consensus with divergent values; byzantine faults affect the status value

Figure 4.13 present the scenario with byzantine faults affecting the status value. The results in both grid and random distributions are identical and once again the byzantine mode has a worst performance.

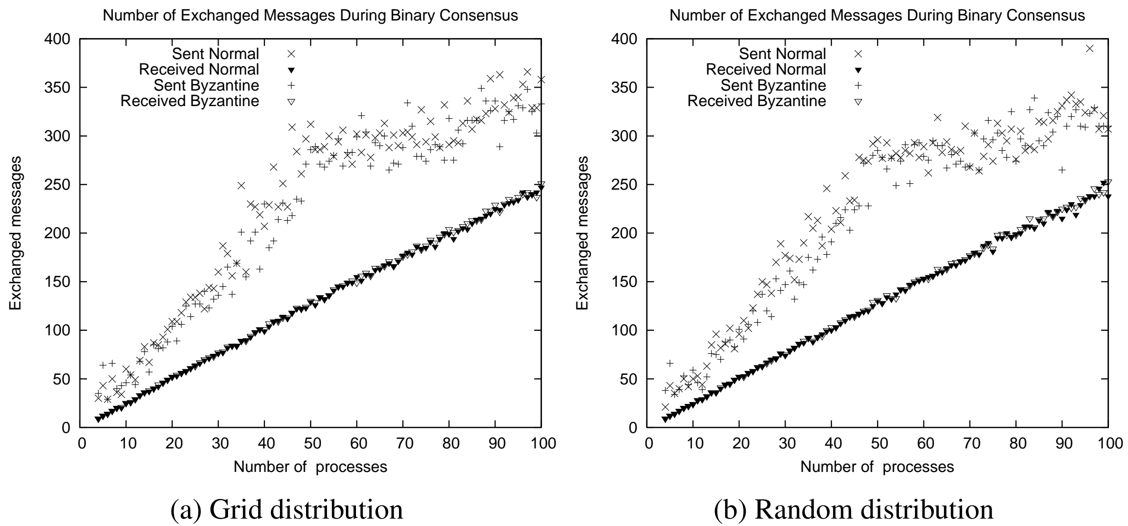


(a) Grid distribution (b) Random distribution

Figure 4.14: Number of phases of binary consensus with divergent values; byzantine faults affect the proposal value

The number of phases with byzantine value attack is presented in Figure 4.14. This results are different in both grid and random distributions. The random distribution reveals a n overall better performance.

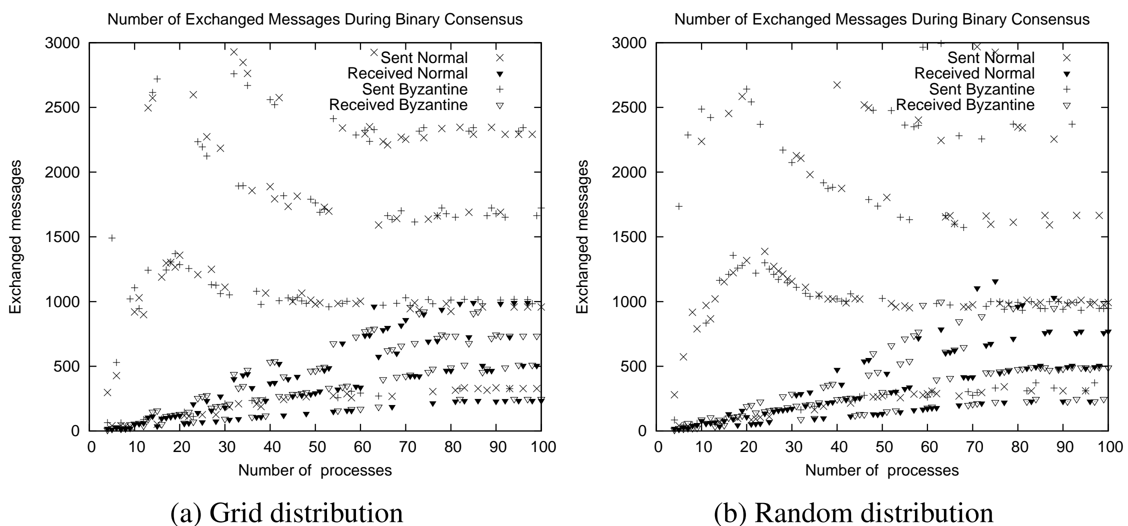
Number of sent and received messages



(a) Grid distribution (b) Random distribution

Figure 4.15: Exchanged messages of binary consensus with unanimous values; byzantine faults affect the proposal value

Figure 4.15 presents the number of sent and received messages in fault-free run and when byzantine processes temper the proposal value. The values proposed by correct processes are unanimous. The number of received messages is linear to the number of processes, which is explained by the fact that in every case it only took 4 phases to achieve consensus and it is required a certain number of messages to make progress. The number of sent messages is superior to the number of received messages and is not linear, that is because some messages are lost and processes need to resend the messages.



(a) Grid distribution (b) Random distribution

Figure 4.16: Exchanged messages of binary consensus with divergent values; byzantine faults affect the proposal value

Figure 4.16 presents the number of sent and received messages in fault-free run and when byzantine processes temper the proposal value. The proposal values are divergent.

Once again the number of sent messages is superior to the number of received messages. In this scenario there is not a linear relation between the number of processes and the number of received messages. When the proposal value is divergent it requires the processes to flip the coin to make progress which in turn requires more phases and consequently more messages to achieve agreement.

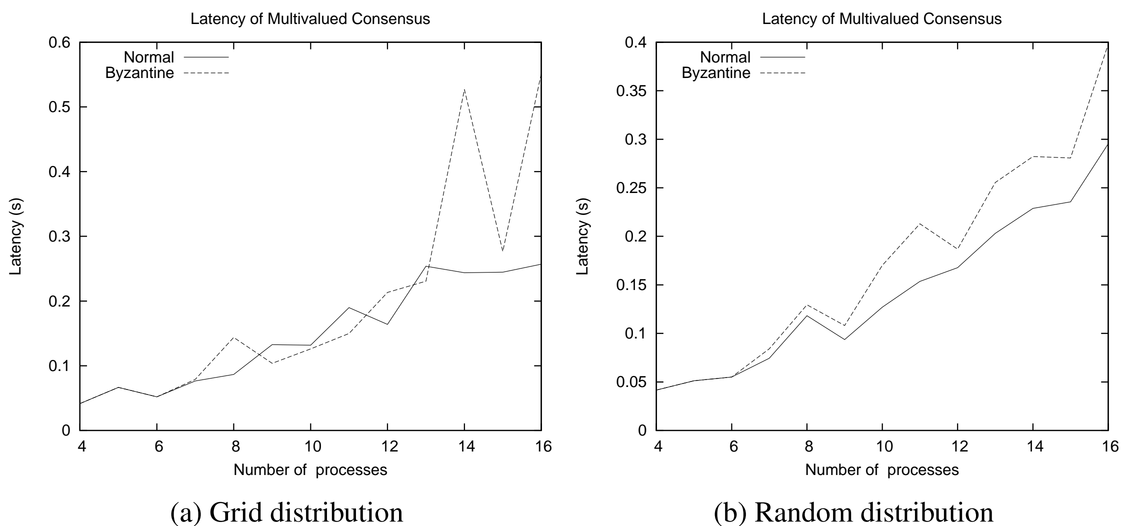
4.4.3 Multivalued Consensus

This section includes the simulations results of the multivalued consensus. There were executed 10 instances of multivalued consensus from 4 to 16 processes in several scenarios. The execution time was collected to evaluate the average latency of the protocol.

Latency

The following figures present the average latency of multivalued consensus. In each scenario both the grid and the random distributions are presented, respectively, side by side. The

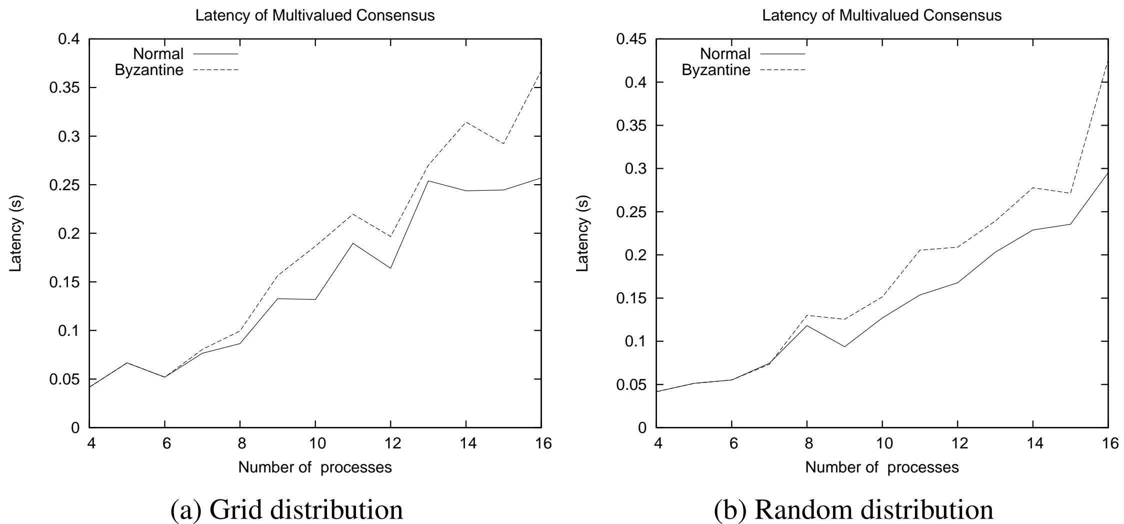
Figure 4.17 presents the latency of binary consensus when processes propose divergent value, byzantine faults occur to the identity of the sender of the messages. The results are very identical in both distributions. The byzantine executions have a worse performance then the fault-free scenario.



(a) Grid distribution (b) Random distribution

Figure 4.17: Latency of multivalued consensus with divergent values; byzantine faults affect the identity of the sender

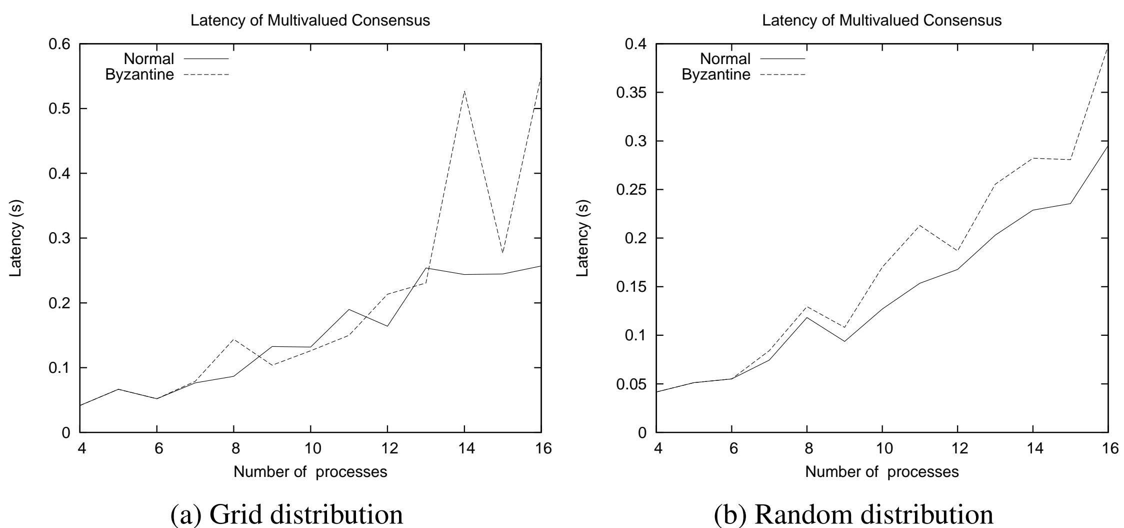
The byzantine value attack with divergent proposals is presented in Figure 4.18. This results are different in both grid and random distributions and present the greatest amplitude of values for the byzantine case, which is also the worst case.



(a) Grid distribution (b) Random distribution

Figure 4.18: Latency of multivalued consensus with divergent values; byzantine faults affect the proposal value

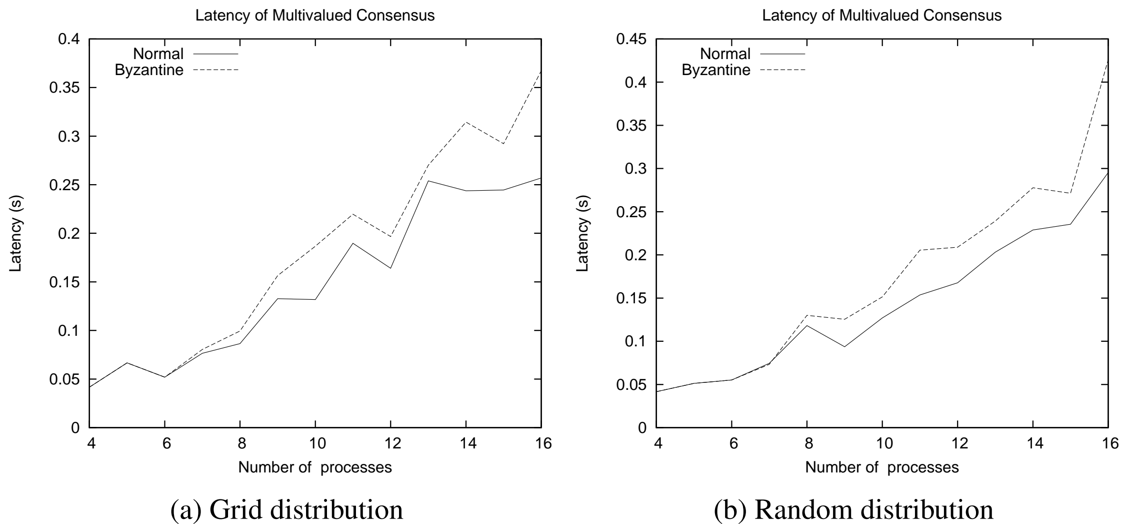
When the proposals are unanimous the results are better. Figure 4.19 presents the scenario where byzantine faults occur to the identity. The fault-free case is better and the difference between both cases increases as the number of processes increases. The results in both the grid and random distributions are very identical.



(a) Grid distribution (b) Random distribution

Figure 4.19: Latency of multivalued consensus with unanimous values; byzantine faults affect the identity of the sender

Figure 4.20 presents the scenario with unanimous proposals and byzantine value attacks. The byzantine mode reveals greater latency values in every case. The difference between both modes increases with the number of processes.



(a) Grid distribution (b) Random distribution

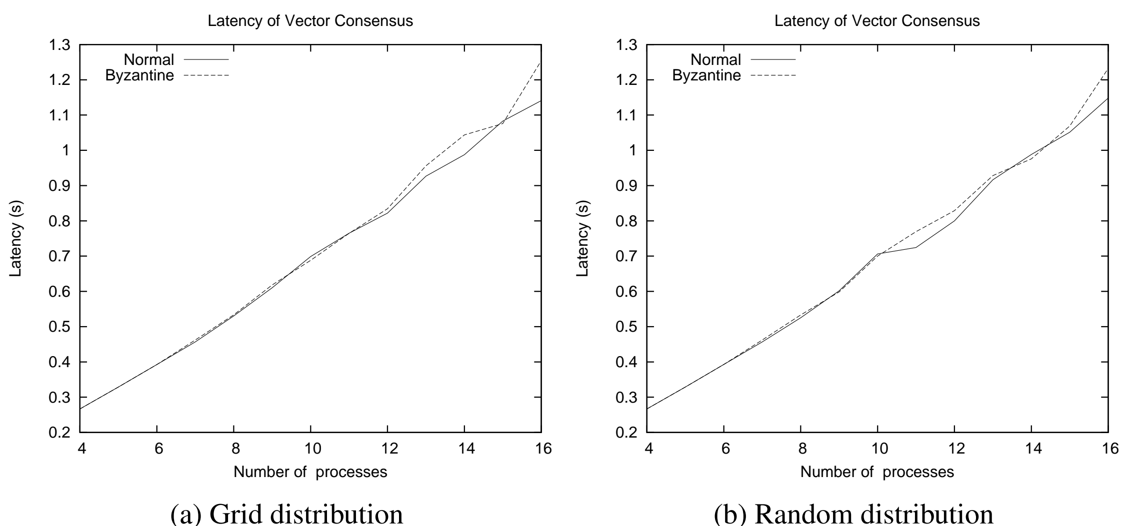
Figure 4.20: Latency of multivalued consensus with unanimous values; byzantine faults affect the proposal value

4.4.4 Vector Consensus

This section includes the simulations results of the vector consensus. There were executed 10 simulations from 4 to 16 processes. The execution was collected to evaluate the average latency of vector consensus execution.

Latency

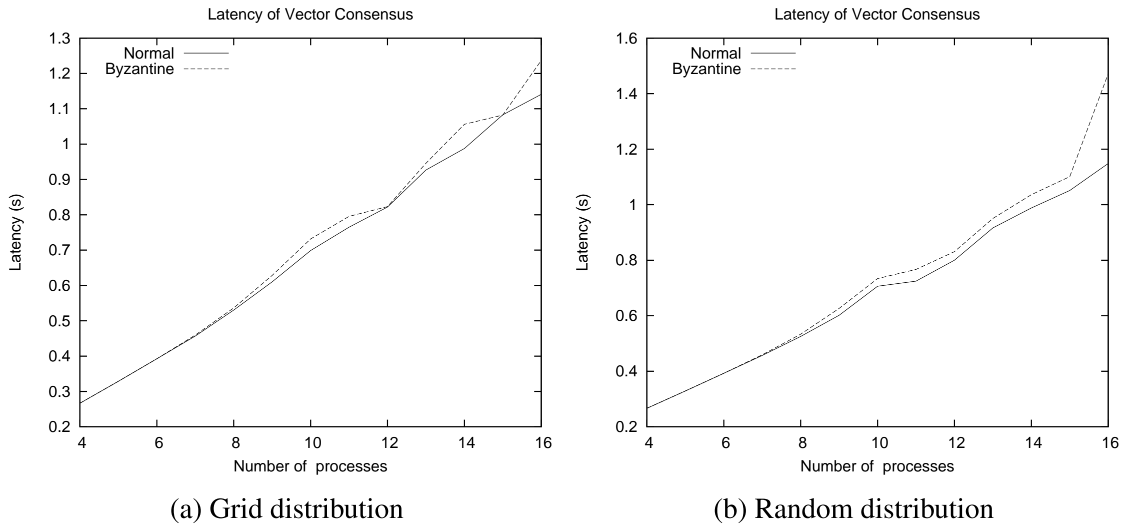
The following figures present the average latency of 10 executions with a variation on the number of processes from 4 to 16 with different scenarios of executions. In each scenario both the grid and the random distributions are presented, respectively, side by side. The results are different from multivalued and binary consensus, in this case there is a linear growth of the latency in function of the number of processes.



(a) Grid distribution (b) Random distribution

Figure 4.21: Latency of vector consensus with divergent values; byzantine faults affect the identity of the sender

Figure 4.21 presents the latency of binary consensus when processes propose divergent value, byzantine faults occur to the identity of the sender of the messages. The results are very identical in both distributions. The byzantine executions have a worse performance then the fault-free scenario.

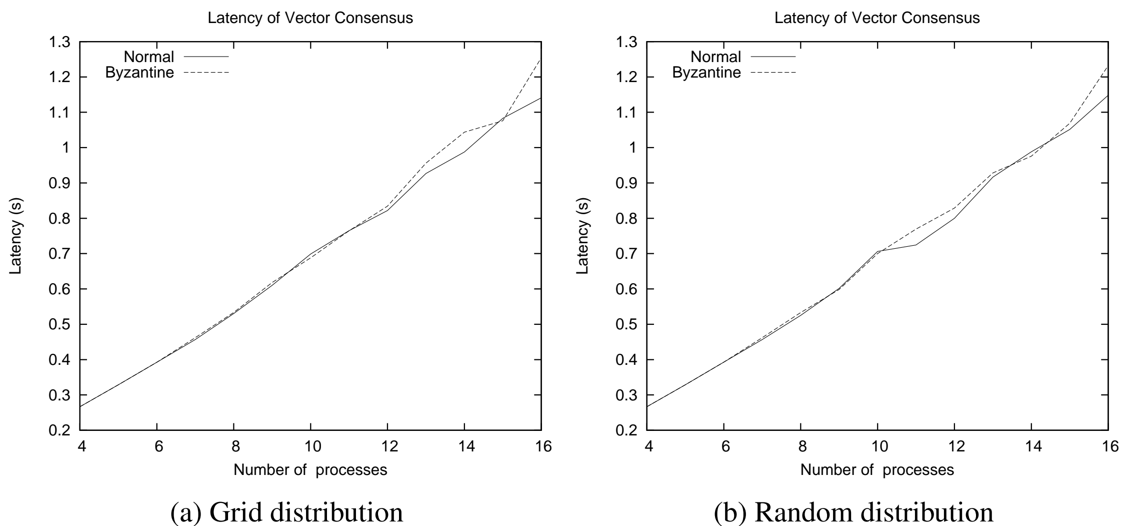


(a) Grid distribution (b) Random distribution

Figure 4.22: Latency of vector consensus with divergent values; byzantine faults affect the proposal value

The byzantine value attack with divergent proposals is presented in Figure 4.22. This results are different in both grid and random distributions and present the greatest amplitude of values for the byzantine case, which is also the worst case.

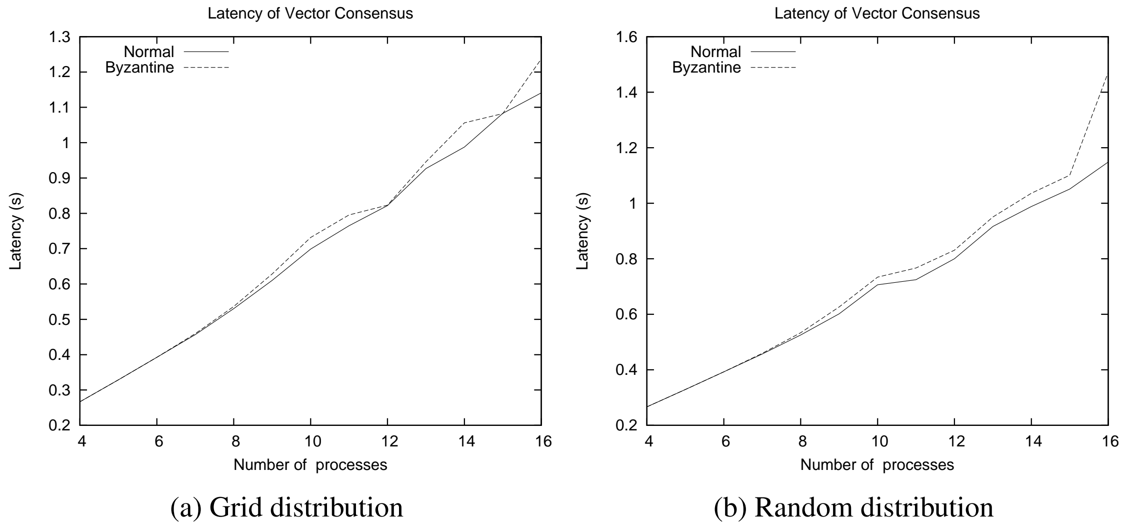
When the proposals are unanimous the results are better. Figure 4.23 presents the scenario where byzantine faults occur to the identity. The fault-free case is better and the difference between both cases increases as the number of processes increases. The results in both the grid and random distributions are very identical.



(a) Grid distribution (b) Random distribution

Figure 4.23: Latency of vector consensus with unanimous values; byzantine faults affect the identity of the sender

Figure 4.24 presents the scenario with unanimous proposals and byzantine value attacks. The byzantine mode reveals greater latency values in every case. There is a slightly difference between the two distributions, the random distribution is slower than the grid distribution.



(a) Grid distribution (b) Random distribution

Figure 4.24: Latency of vector consensus with unanimous values; byzantine faults affect the proposal value

Chapter 5

Conclusion

Achieving consensus in ad hoc wireless networks is a difficult yet fundamental task. Agreement is crucial to manage and configure the network. Consensus becomes even more difficult when dealing arbitrary faults. There are few consensus protocols for this kind of networks specially when considering arbitrary faults. This thesis gives a contribution by adapting two consensus protocols [17] [6] to fit the defined system model, and proposes a new vector consensus protocol. With these algorithms it was possible to develop a network library that provides three consensus primitives: binary, multivalued and vector consensus. The developed library allows the configuration and management of a network and provides agreement primitives for distributed applications.

5.1 Discussion of the results

The simulations of the binary consensus show similar latency results in both the normal operation mode and the byzantine mode, which reveals that this protocol was a good choice given the assumption of arbitrary faults. The average latency is very low, which was a requisite since the binary consensus protocol is the most used protocol in the stack. The number of messages is also low, which is an important fact, since it is important that the protocol stack does not affect the application execution. The number of phases needed to achieve consensus did not increased with the number of processes executing it, which shows that the algorithm is scalable, as long as there is a efficient communication support to manage exchanged messages. Another interesting result is the fact that in every simulation all the processes terminated the consensus execution, which was not a requirement since this is a k-consensus protocol. This result does not mean that in every case all the processes terminate, but that it is very probable that they do so.

The multivalued consensus protocol was able to achieve a decision in less than one second for up to 16 process. This is a very low value since the protocol uses binary consensus to achieve a decision. The injection of byzantine faults to the proposal value did not broke any of the properties of the protocol and did not decrease performance.

Vector consensus protocol is the slowest of all three, which was an expected result since it uses the multivalued consensus protocol which in turn uses binary consensus. However, it took less than two seconds to achieve vector consensus for up until 16 processes. This is a good value because, although 16 processes may be less than a normal ad hoc wireless network, in this case it could be the enough because a sink with 16 processes may be part of a much bigger network

The preliminary application for Android with the studied protocols show that it is possible to implement these kind of protocols in this kind of devices. The API provides blocking and non-blocking propose function calls and callbacks. A distributed application can use the most useful propose call for each kind of operation.

5.2 Future Work

The work presented in this thesis provides a study and evaluation of a stack with consensus protocols for MANETs. The stack was implemented in Android, however, the network library does not include the work of Emanuel Alves [3], instead it uses directly the network interface provided by Android. In the future, the protocol stack must be set up on top of the support layer.

Another interesting feature that come up during the development of the library was the possibility of detecting faulty processes. The detection could be made through the rejected messages by all three protocols. This mechanism could, in theory, increase performance since the malicious processes are automatically removed from the network and there are less messages to process at each round.

The protocol stack allows distributed applications or other protocols to achieve agreement. Developing other protocols that take advantage of consensus, such as: atomic broadcast, state machine replication or view synchronous, would improve the stack. There are also a range of distributed applications that can be implemented.

Abbreviations

API Application Programming Interface.

BFT-CUP Byzantine Fault-tolerant - Consensus wiht Unknown Participants.

DOS Denial of Service.

FLP Michael J. Fischer, Nancy Lynch, and Mike Paterson.

FT-CUP Fault-tolerant - Consensus wiht Unknown Participants.

IP Internet Protocol.

MAC Message Authentication Code.

MAC Media Access Control.

MANET Mobile Ad hoc NETwork.

NS-3 Network Simulator-3.

OSR One Sink Reducibility.

PD Participant Detector.

PDA Personal Digital Assistant.

UDP User Datagram Protocol.

Bibliography

[1] E. Alchieri, A. Bessani, and S.Fraga. Byzantine Consensus with Unknown Participants. *Proceedings of the 12th International Conference on Principles of Distributed Systems*, 2008, pp. 22–40.

[2] Wi-Fi Aliance. Wi-Fi CERTIFIED Wi-Fi Direct. *White Paper: http://www. wi-fi. org/news_articles. php*, 2010.

[3] E. Alves. Comunicação e Filiação em Redes Ad-hoc Móveis com Participantes Desconhecidos. Master's thesis, Department of Informatics, Faculty of Sciences, University of Lisbon, Oct. 2013.

[4] D. Cavin and Y. Sasson. Reaching Agreement with Unknown Participants in Mobile Self-Organized Networks in Spite of Process Crashes. *Technical Report IC/2005/026*, 2005.

[5] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2), 1996, pp. 225–267.

[6] M. Correia, N. Neves, and P. Veríssimo. From Consensus to Atomic Broadcast: Time-free Byzantine-resistant Protocols Without Signatures. *The Computer Journal*, 49(1), Jan. 2006, pp. 82–96.

[7] P. Devi and A. Kannammal. Security Attacks and Defensive Measures for Routing Mechanisms in MANETs A Survey. *International Journal of Computer Applications*, 42(4), 2012, pp. 27–32.

[8] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), Apr. 1985, pp. 374–382.

[9] V. Hadzilacos and S. Toueg. Fault-tolerant Broadcasts and Related Problems. In *Distributed Systems (2nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., 1993.

[10] T. Hendersona, S. Roy, S. Floyd, and G. Riley. NS-3 Project Goals. *Proceeding from the 2006 Workshop on NS-2: the IP Network Simulator*, 2006.

[11] Y. Hu and A. Perrig. A Survey of Secure Wireless Ad Hoc Routing. *IEEE Security and Privacy*, 2(3), Jun. 2004, pp. 28–39.

[12] Y. Hu, A. Perrig, and D. Johnson. Packet Leashes: A Defense Against Wormhole Attacks in Wireless Networks. *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, 3, Apr. 2003, pp. 1976–1986.

[13] Y. Hu, A. Perrig, and D. Johnson. Rushing Attacks and Defense in Wireless Ad Hoc Network Routing Protocols. *Proceedings of the 2nd ACM workshop on Wireless security*, Sep. 2003, pp. 30–40.

[14] M. Ilyas and R. Dorf. The Handbook of Ad Hoc Wireless Networks. *CRC Press*, 2003.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), Jul. 1982, pp. 382–401.

[16] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC press, 2010.

[17] H. Moniz, N. Neves, and M. Correia. Turquois: Byzantine Consensus in Wireless Ad Hoc Networks. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2010, pp. 537–546.

[18] A. Mostefaoui, M. Raynal, and F. Tronel. From Binary Consensus to Multivalued Consensus in Asynchronous Message-passing Systems. *Information Processing Letters*, 73(5), Mar. 2000, pp. 207–212.

[19] P. Ning and K. Sun. How to Misuse AODV: A Case Study of Insider Attacks against Mobile Ad-hoc Routing Protocols. *Ad Hoc Networks*, 3(6), Nov. 2003, pp. 795–819.

[20] Android Developers Reference. http://developer.android.com/guide/basics/what-is-android.html, 2011.

[21] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2), Feb. 1978, pp. 120–126.

[22] N. Santoro and P. Widmayer. Time is Not a Healer. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science*, February 1989, pp. 304–313.

[23] K. Sanzgiri, B. Dahill, B. Levine, C. Shields, and E. Belding-Royer. A Secure Routing Protocol for Ad Hoc Networks. *In Proceedinds of the 10th IEEE International Conference on Network Protocols*, Nov. 2002, pp. 78–87.

[24] W. Stallings. *Cryptography and Network Security - Principles and Practices*. Pearson Prentice Hall, 2006.

[25] J. Turek and D. Shasha. The Many Faces of Consensus in Distributed Systems. *Computer*, 25(6), Aug. 1992, pp. 8–17.

[26] B. Wu, J. Chen, J. Wu, and M. Cardei. A Survey on Attacks and Countermeasures in Mobile Ad Hoc Networks. *Wireless Network Security, Springer US*, 2006, pp. 103–135.

[27] J. Wu and I. Stojmenovic. Ad Hoc Networks. *Computer*, 37(2), 2004, pp. 29–31.