Design and Performance Analysis of Fail-Signal Based Consensus Protocols for Byzantine Faults

Thesis by

Qurat-ul-Ain Inayat Tariq

In Partial Fulfilment of the Requirements

For the Degree of

Doctor of Philosophy

University of Newcastle upon Tyne

Newcastle upon Tyne, UK

Submitted September 4th, 2007

Defended November 2nd, 2007

To ammi and papa

I owe you my every breath

Acknowledgements

I feel immense pleasure to reach the point in my PhD studies at which I am writing this document. Achieving this sense of satisfaction could not have been possible without the constant support of my thesis supervisor Dr. Paul Ezhilchelvan. Long technical discussions with him helped me to clarify many concepts. He always encouraged me to think critically and not to hesitate in asking questions. Also, him being easily reachable prevented all barriers to the thinking process. Thanks for being my mentor.

I feel greatly indebted to my mother. She has always encouraged me to take new initiatives and move forward ever since childhood. I could not have done anything without her backing and her day and night prayers. I owe her my every success. I love you ammi and I am proud to be your daughter. Glimpses of my late father in my memory, though are very few, always give me great inspiration.

All colleagues and staff members, especially the reception staff at the School of Computing Science have been very friendly. Special thanks to Jennie, Giovanna and Doug, who have been my closest friends at Newcastle, for making my stay pleasant. Many thanks to Muhammad Alsaeed for making it easier for me to use his WAN Emulator. I would also like to thank Mr. Waseem Asrar for guiding me while applying for the PhD and supporting me through tough times, Mr. Abul Kalam for facilitating me to make this journey possible and Dr. Shahid Hafeez Mirza for nominating me for sponsorship.

Special thanks to my sponsors: the Commonwealth Scholarship Commission for three years of PhD funding, Professor Isi Mitrani and the school Research Committee (ResCom) for additional financial support and Professor Santosh Shrivastava for funding my DSN 2007 trip.

Last but not least, my heartfelt thanks go to my husband Tariq, who made me feel that I could get through this tough time easily. Although we have been apart during my studies, his love and support has refreshed me everyday.

Abstract

Services offered by computing systems continue to play a crucial role in our every day lives. This thesis examines and solves a challenging problem in making these services *dependable* using means that can be assured not to compromise service *responsiveness*, particularly when no failure occurs. Causes of undependability are faults and faults of all known origins, including malicious attacks, are collectively referred to as *Byzantine* faults.

Service or *state machine replication* is the only known technique for tolerating Byzantine faults. It becomes more effective when replicas are spaced out over a wide area network (WAN) such as the Internet – adding tolerance to localised disasters. It requires that replicas process the randomly arriving user requests in an *identical order*. Achieving this requirement together with deterministic termination guarantees is impossible in a fail-prone environment. This impossibility prevails because of the inability to accurately estimate a bound on inter-replica communication delays over a WAN. Canonical protocols in the literature are designed to delay termination until the WAN preserves convergence between actual delays and the estimate used. They thus risk performance degradation of the replicated service. We eliminate this risk by using *Fail-Signal* processes to circumvent the impossibility.

A fail-signal (FS) process is made up of redundant, Byzantine-prone processes that continually check each other's performance. Consequently, it fails only by crashing and also signals its imminent failure. Using FS process constructs, a family of three order protocols has been developed: Protocol-0, Protocol-I and Protocol-II. Each protocol caters for a particular set of assumptions made in the FS process construction and the subsequent FS process behaviour. Protocol-I is extensively compared with a canonical protocol of Castro and Liskov which is widely acknowledged for its desirable performance. The study comprehensively establishes the cost and benefits of our approach in a variety of both real and emulated network settings, by varying number of replicas, system load and cryptographic techniques. The study shows that Protocol-I has superior performance when no failures occur.

Contents

List of Figures

List of Tables

List of Notations and Abbreviations

n	Number of nodes in the replicated server system
f	Fault-tolerance degree (maximum number of failures) of the replicated server system
N_i	i^{th} node of the replicated server system
s_i	i^{th} service process hosted on N_i (executing service application)
p_i	i^{th} order process hosted on N_i (executing order protocol)
m	Message used for communication between two processes
$D(m)$	Digest of m computed by using a cryptographic hash function D
PA	Paxos Algorithm [Lam01]
CTA	Chandra-Toueg Algorithm [CT91]
rnd#	Proposal number for PA/Round number for CTA (iteration number)
ϕ	Fault-tolerance degree (maximum number of failures) of fail-silent/fail-signal/fail-stop process
ISIQ	Identically Sequenced Input Queue
FS	Fail-Signal
CT	Simple Crash-Tolerant protocol derived from Protocol-I
BFT	Byzantine Fault-Tolerant order protocol of [CL99]
v	View number for BFT
cl_i	i^{th} client of the replicated server system
o	Order number assigned to a request
c	Coordinator process number (For Protocol-0, -I and –II, $c \geq 1$ normally and $c = -1$ during Install execution)
h and H	Low and high watermarks for BFT
p'_i	i^{th} shadow order process
P_i	i^{th} FS process implemented by p_i and p'_i
FS_i	Double-signed fail-signal message from FS process P_i
Π	Set of all FS processes
π	Set of all order processes co-located with service processes
π'	Set of all shadow processes
$max\_committed_i$	Largest o committed by p_i
A_i	Largest o acknowledged by p_i

$start\_o_c$	Starting o from which a new coordinator P_c starts its ordering regime
$Signalled_i$	Set of FS processes from which p_i has received fail-signal messages
Order_Pool	Set of all *ORDER* messages sent/received by a process
Ack_Pool	Set of all *ORDER* messages acknowledged by a process
eligible()	Number of the process eligible to be the next coordinator (sometimes used to refer to the process itself)
Σ_i	i^{th} system configuration with P_i as coordinator
$Acceptors_i$	Set of all acceptors in Σ_i
n_i	Number of acceptors in $\Sigma_i = \|Acceptors_i\|$
f_i	Maximum number of processes that can fail in Σ_i
VQ_i	Valid quorum size in Σ_i
Q_i	Smallest valid quorum size in Σ_i
$c+$	Number for the process executing Install part (Protocol-I and -II) as new coordinator (*eligible*() = $c+$)
$c-$	Number of a predecessor coordinator process
PC_{mx}	The largest o that P_{c+} knows to be *possibly* committed by some correct process.
CC_{mx}	The largest o that P_{c+} knows to be *certainly* committed by some correct process.
$m\_c_i(j)$	The largest o for which an $ORDER_j(o)$ is committed by p_i
$A_i(j)$	The largest o of $ORDER_j(o)$ for which an $ACK_i(o)$ is produced by p_i
commit_count	Vector containing largest committed request number for every client
Status_Pool(c)	Set of *STATUS*(c) messages sent/received
Start_Pool	Set of $START_{i,j}(c)$ messages received from every process p_j
Foundation_Pool(c)	Set of first arriving Q_c $STATUS_i(c)$ messages
Advanced_Pool(c)	Set of $STATUS_i(c)$ messages arriving after first Q_c ones
s	Source number
$Commit\_Board_i$	List of the largest $max\_commited_j$ value received by p_i in any $ACK_j(o)$ message from every process p_j (in Install-II)
LW_i	the largest element in $Commit\_Board_i$ which is less than or equal to at least Q_c elements.

Chapter 1

Introduction

Computers are increasingly pervading our everyday lives. We use them for a variety of services – from simple email service to financial dealings (e.g., cash withdrawal, holiday booking, Internet Auctions etc.) to entertainment (e.g., computer games). The underlying infrastructures providing these services are generally complex and consist of a number of computers connected by a network. They are often structured as per the well-known *Client-Server paradigm*. This paradigm identifies computers as *clients* and *servers*, where a client contacts a server requesting for some service provision. Server processes the request and provides the results back to the client.

The increasing trend of using computer systems also brings with it a high degree of reliance being placed on these systems. With this growth in dependency, the most crucial concern evidently becomes one of *reliability* and *availability*. That is, service requests should be processed correctly and service provision should proceed without interruption. This concern is central to system design due to the increasing susceptibility of such systems to hardware and software faults and malicious attacks. Hence, it is desirable, and more often essential, to have systems that can continue working correctly even in the presence of a reasonable number of faults.

The *state machine replication* [GS97, Sch93, DGG05] is a well-known approach to achieving fault-tolerance in distributed server systems and the only one for tolerating Byzantine faults. The approach is intuitively simple. The service that needs to be fault-tolerant is implemented as a deterministic state machine and then replicated over a number of *redundant* computers that are assumed to fail independent of each other. These computers execute the service process independently for each request to produce result. The results produced by them are used to mask out erroneous and/or absent responses from faulty replicas. Thus, these replicas collectively build a fault-tolerant service.

For replication to be effective, it is important that correct replicas generate identical responses for a given request. This leads to the following two requirements

that need to be met. First, the service in question must be built as a deterministic *state machine:* given that correct replicas are in identical states, processing a given request will lead to producing identical results and identical state changes at correct replicas. Building a service as a deterministic state machine need not be a trivial task and we refer the reader to [TS90] for the subtle issues that need to be addressed.

Secondly, correct replicas must process the requests in identical order. Specifically, requests received from various clients at various replicas should be (i) identical and (ii) executed in the same order. The replicas therefore need to coordinate with each other to ensure that the second requirement is met. To achieve this coordination, replicas are provided with a primitive called the *Atomic Broadcast* or *Total Order Broadcast*. The protocol that implements this primitive will be referred to here as the *Total Order* protocol.

Total order is a typical instance of the "agreement" problem whereby distributed replicas unanimously arrive at an identical decision through exchange of messages and without any external assistance. In total order, the decision is on the order in which a given request (or any other event) needs to be viewed/processed in relation to other requests (or events). Other instances of the agreement problem include non-blocking atomic commitment and group membership. The former is concerned with decision of whether a transaction should be committed whereas the latter deals with deciding whether a replica is considered to be part of the group.

Reaching consensus refers to solving the agreement problem. It has been shown that a solution to the most agreement problems can be devised by using a solution to reach consensus. In this chapter, we formally define consensus and discuss what makes reaching consensus a nontrivial task. Furthermore, the system context in which we intend to solve the problem is presented and the basic assumptions considered throughout this thesis are listed. The last two sections discuss thesis contributions and outline the thesis structure.

1.1 The Consensus Problem

This section presents the model of a networked system hosting a replicated service. It then formally defines the consensus problem. It also discusses the difficulties that make reaching consensus impossible in the given system context.

1.1.1 Replication Context

We consider a distributed system of n nodes (machines) hosting a replicated service. For simplicity, we will assume that each node hosts a *service process* executing the service application and an *order process* running some total order protocol. Let us represent the order processes by a set $\{p_1, p_2, ..., p_n\}$.

A *correct* node behaves according to its specification and the processes hosted by it are termed to be *correct*. A node or a process that is not correct is termed faulty. The most benign form of faults found in literature is *crash*. A node crashes when it halts execution of all processes it hosts. On the other extreme, a *Byzantine fault* causes the node to fail arbitrarily and no assumption can be made about the behaviour of a faulty process. We consider the *authenticated Byzantine* fault model in which the faulty behaviour is arbitrary but only within the assumed cryptographic constraints. That is, a faulty process cannot modify messages of a correct process undetectably and cannot forge a correct process's encrypted message. Note that any algorithm that tolerates Byzantine faults is also crash-tolerant. Similarly, if an algorithm cannot tolerate crash, it cannot certainly tolerate Byzantine faults.

The nodes are connected via an *asynchronous* network and communicate with each other only by message passing. Communication delays between, and the relative processing speeds of, any two nodes are assumed to be unknown. The only assumption this network model makes is that the bounds on communication delays and relative processing speeds are finite (but unknown). Due to its no-known-bounds aspect, the asynchronous network is the most general model of communication system and covers any wide area network such as the Internet; it also abstracts reliable implementations of 1-to-1 and 1-to-many communication schemes involving repeated transmission of messages until acknowledgements are received from destination(s).

1.1.2 Defining Consensus

The consensus problem can be informally defined in the following way. Each correct process p_i proposes a value v_i and decides on one (any one) of the proposed values v in such a way that the decision is identical with all correct processes. Thus, the processes are said to have reached consensus on the proposed values.

Formally, this decision process is required to satisfy the following properties.

(i) **Integrity** – If a correct process decides v then v was proposed by some process.

(ii) **Agreement** – No two correct processes decide differently.

(iii) **Termination** – Every correct node eventually decides.

Some consensus solutions satisfy a variant of the agreement property called *Uniform Agreement*. This is a weaker property and a consensus satisfying this property is called *uniform consensus*. Uniform agreement is defined as follows

Uniform Agreement – No two processes (correct or faulty) decide differently.

Note that, unlike agreement, the uniform agreement version of this property does require even the faulty processes to decide identically, which is even harder to achieve [CS00]. However, this only makes sense in fault models where a failure does not involve a process making erroneous state transitions. Since Byzantine processes make arbitrary state transitions, guaranteeing this property for a faulty process in this case is impossible. Hence, we will consider non-uniform consensus only.

First two properties ensure the service execution to be consistent at all correct processes. This is called *safety* property of a consensus algorithm. It results in identical sequence of inputs at all correct service processes. The third property ensures that the service processes continue execution and keep producing outputs for incoming client requests. This property of a consensus algorithm guarantees responsiveness and is called *liveness* property.

1.1.3 Why reaching consensus is hard?

Fischer, Lynch and Patterson [FLP85] establish that consensus cannot be solved with deterministic termination guarantees in an asynchronous network even with crash fault model. They state that, even with the strongest failure model, where processes only crash i.e., fail quiescently, it is impossible to find a deterministic consensus solution in an asynchronous network environment. This fundamental result is famously known as the *FLP impossibility*. The result has been proved for crash faults only but, it is also valid for less benign faults like Byzantine faults.

The core to this impossibility result is the argument that it is impossible to distinguish a crashed process from a slow one. The slowness of the latter can be due to a slow network connection or slow processing or both. This inability to safely distinguish crashed process from a slow one may result in choosing between one of the two situations.

(i) Say a correct process chooses to wait for a response from every other process. In this situation, it may end up waiting for a response from the crashed processes forever. This violates liveness property stated above.

(ii) Say correct processes are made to use timeouts to avoid the forever waiting situation. Two correct processes may timeout on different correct but slow processes. This will result in inconsistency in the state of correct processes and violates safety property.

Based on this core argument, [FLP85] formally proves that reaching consensus is impossible in this system context. However, there are several ways in which this impossibility can be circumvented and this investigation has been an active research area in the past two decades.

1.1.4 Total Order Broadcast: an Equivalent Problem

It has been shown that consensus and total order broadcast are equivalent problems. [DDS87] show that total order broadcast can be transformed into consensus whereas [CT96] show the other way round. Hence, it has been proved that the impossibility result holds for both problems. Moreover, if a solution exists for one, it can be transformed to solve the other. In this thesis, we focus on solving total order broadcast but the solutions proposed here are valid for both problems. We present the formal specification of total order broadcast below.

Formally, total order broadcast is described in terms of two primitives *TO-broadcast*(m) and *TO-deliver*(m) [DSU04]. When a process p invokes TO-broadcast(m) (respectively TO-deliver(m)), it is said that p TO-broadcasts (respectively TO-delivers) message m. These primitives satisfy the following three properties.

(i) **Termination** – If a correct process TO-broadcasts message m then it eventually TO-delivers m. Moreover, if a correct process TO-delivers m, then all correct processes eventually TO-deliver m.

(ii) **Integrity** – For any message m, every correct process TO-delivers m at most once and only if m was TO-broadcast by some process.

(iii) **Total Order**– If two correct processes p_1 and p_2 TO-deliver messages m_1 and m_2 then p_1 TO-delivers m_1 before m_2, iff p_2 TO-delivers m_1 before m_2.

Note that, as for consensus, we ignore the uniform variants of Termination, Integrity and Total Order properties here as they do not apply to Byzantine fault model.

We now discuss how a total order protocol is used to implement state machine replication. As mentioned earlier, every node of the replicated server system executes a service process and an order process. The order processes execute a protocol that ensures the above three total order broadcast properties. On receiving every new request from a client, order processes communicate with each other to assign a unique and identical order number. Hence, all correct order processes forward all clients' requests in identical order to the corresponding service processes for execution. This leads to identical result generation at various replicas (with deterministic service processes).

1.2 Thesis Background

Solving consensus is the main objective of this thesis. However, being crucial to distributed system applications, consensus problem has been studied extensively. Researchers have tackled the problem from different perspectives by making various assumptions and relaxing certain classical asynchrony restrictions in order to circumvent the FLP impossibility. A large number of approaches have been proposed to solve this problem based on different failure and communication models. However, we solve consensus by using a novel approach named *Fail-Signal* to circumvent the FLP impossibility. To appreciate the novelty of our approach and to show where this approach stands in relation to other works, we first present a brief overview of the two other common approaches below. A concise survey of these well-known and most commonly used approaches adopted to circumvent FLP impossibility is given in chapter 2. For clarity, we refer to them as type 1 and 2 in the following text.

1.2.1 Type 1 Protocols

Type 1 protocols, e.g. [CL99, YMV+03], are deterministic and their termination is guaranteed, even in failure-free runs, only when message delays over the network are perceived to remain stable for a suitably long duration. Such system-level requirements for termination are presented in [CT96] in terms of mistake-prone *failure detectors.* (They make mistakes when timeouts they employ to suspect process failures turn out to be too small.) Consensus performance is influenced by how often and how long the timeout values chosen for failure suspicion (*delay convergence*) remain too small compared to actual delays. Similarly, a partial synchronous model introduced in [DLS88] assumes that some bound on communication and processing delays either hold or will eventually hold.

Protocols in this category tend to be *coordinator-based*. That is, in these protocols a process is given the role of a coordinator and the ordering is enforced by this special process. [Lam98] and [CT91] are considered the pioneering works in this category for crash-tolerant environment. Moreover, some Byzantine fault-tolerant protocols also employ failure detectors and take a coordinator based approach. Among these, [CL99], also known as BFT, is the most famous for its high performance and low overhead cost.

Known works on performance evaluation of type-1 protocols are limited to fail-free and mistake-free environments e.g., [KS07, ADG+03, JMM07] and we know of no work that investigates on how to adaptively choose timeouts to minimise mistakes. (Establishing the efficacy of any such adaptation may not be easy as well.) Thus, using these protocols for providing consensus when communication delays (say, over the Internet) can fluctuate widely will be akin to letting TCP be used without any congestion control mechanism, such as [Jac88], that adapts the timeout values used for packet re-transmission.

1.2.2 Type 2 Protocols

Type 2 protocols are randomized in nature and require that the random choices made by processes independently from a finite set of values converge; this requirement is guaranteed in probabilistic terms to be a certainty with the passage of time [EMR01, MNC+06]. The more the number of processes seeking consensus and the more different are their initial values/intentions, the longer it may take for the convergence to occur (*choice convergence*). Moreover, to the best of our knowledge, no Byzantine fault-tolerant multi-valued randomized protocol has been proposed.

1.2.3 Our Approach

Our approach to circumvent FLP impossibility is to avoid having quiescent failures. We use fail-signal abstraction [MES03] that employs internal redundancy to detect and signal internal failures. A fail-signal process is composed of redundant processes and halts on detection of an internal fault. Hence, it converts Byzantine behaviour of the constituent processes to crash accompanied by fail-signalling. [MES03] has demonstrated on a preliminary level the use of FS process to increase the tolerance level of a group membership service [EMS95] from crash to Byzantine. We focus on developing coordinator-based Byzantine-tolerant order protocols.

1.3 Thesis Objectives

The thesis objectives are listed below.

- To device a conceptually new class of coordinator-based protocols that does not require any convergence like the ones mentioned above for termination and hence its performance is solely decided by the prevailing communication delays and having to cope with real failures.
- To develop a solution that is practical for building high-performance dependable system. We particularly aim for the protocols that offer fast and low-overhead performance when there are no failures in the system. The intended consensus solution should only induce reasonable latency so that the service responsiveness for the requesting clients is not much affected specially in fault-free situations.
- To measure the performance of the developed protocols and compare with some other well-known protocol to establish the differences. We intend to experimentally evaluate the costs and benefits of the adopted approach.
- Lastly, to achieve above targets while retaining the following as much as possible.
  (i) Keeping the redundancy level i.e., number of replica processes in the system to the established optimum value for the considered system context.
  (ii) Keeping assumptions regarding network and failure models to the weakest possible.

Thus the thesis statement can be succinctly put as follows.

1.3.1 Thesis Statement

This thesis designs and develops a family of order protocols using the fail-signal approach and demonstrates the practical viability of the approach through a comprehensive performance study projecting the cost and benefits both in absolute and relative terms.

Next section presents the system model for which this thesis aims to solve the consensus problem. We also list all basic assumptions made.

1.4 System Context and Assumptions

We consider a distributed system of n nodes. Each node N_i hosts a service process s_i and an order process p_i, as illustrated in figure 1.1. The service application is assumed to be a deterministic state machine.

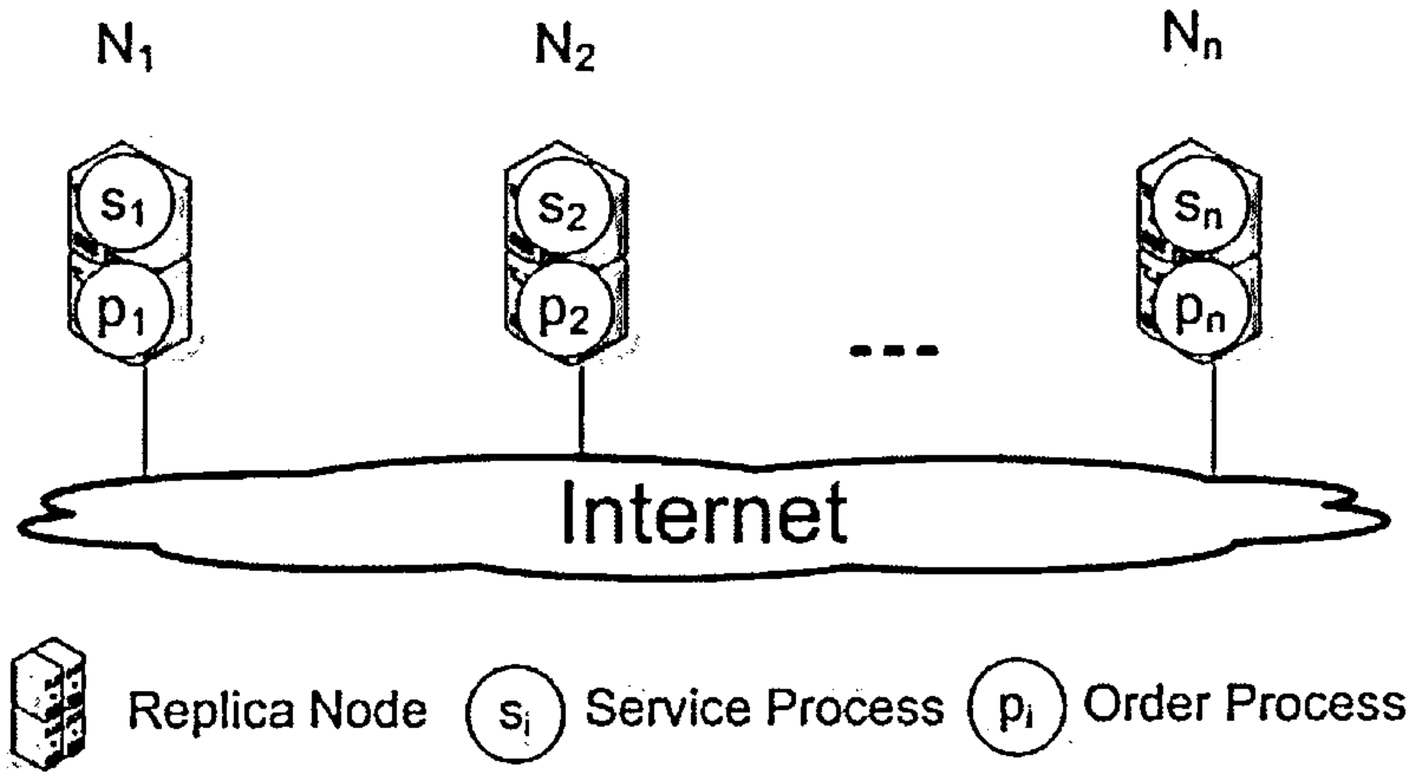


Figure 1.1. System Architecture of the Replicated Service

Replica nodes are connected via an asynchronous network like the Internet. As mentioned earlier, bounds on the communication delays over the network links and relative processing speeds of the nodes are not known. The only assumption made about the network is that these bounds are finite. Hence, it is assumed that if a correct process attempts to send a message infinitely large number of times, the message reaches its intended destinations eventually. Note that unlike other deterministic protocols [CL99, YMV+03], we do not make any system-wide assumption regarding eventual stabilization of communication delays even to ensure liveness. The network is assumed to be unreliable i.e., messages can get discarded, delayed, replicated and re-ordered. A strong adversary having control over the network can also alter the messages but restricted by the cryptographic assumptions stated below.

We assume a (cryptography-constrained) Byzantine fault model. Replicas may fail in Byzantine manner but the failures are to be independent of each other. No assumption can be made for the behaviour of such a node and hence the hosted process. Faulty nodes can crash, loose data, alter data, delay and/or block communication system, delay non-faulty nodes and send incorrect protocol messages to cause damage to the service. Moreover, multiple faulty nodes can collude cleverly in arbitrary ways to mislead the correct processes. Hence, the adversary can perform in any way to undermine the processing of correct processes but it is assumed to be computationally bound and can never subvert the following two cryptographic assumptions.

Every process uses cryptographic techniques for authentication during message communication [Tsu92]. Every message sent to other processes is signed and every message received is verified before consumption. We assume the following

(i) Cryptographic hash function D is used to compute message digests. The hash-functions used are assumed to be 1-way and collision-resistant. That is, if D is the hash function used and a message m produces digest $D(m)$ then it is not computationally feasible to compute m given $D(m)$. Also it should not be possible to find two distinct messages m and m' such that $D(m) = D(m')$.

(ii) All messages sent are signed by using digital signatures. It is assumed that known cryptographic techniques, such as public-key RSA/DSA signatures, are robust enough to prevent message spoofing and replays and to detect message corruption. Hence, it is assumed that a faulty replica can not forge signature of a correct process undetectably.

Like any other Byzantine fault-tolerant replicated system, we assume that the total number of faulty replicas does not exceed a given bound, say f, over the life time of the system. Hence, considering the optimum redundancy level for a replicated state machine, the total number of replicas is taken as $n = (2f+1)$ [Sch93]. Note however that the optimum number of replicas for achieving consensus for an f-Byzantine fault-tolerant system is $(3f+1)$ though [LSP82, DLS88]. We will retain this optimum number of replicas for the second of three protocols we develop and use one extra replica for the other two. Note that to keep the number of replicas in the system as close as the optimum, use of encryption becomes necessary. However, encryption is expensive in terms of performance and an efficient solution may be possible without using cryptography techniques but higher redundancy.

Furthermore, we assume that this (replicated) service has a finite set of authorised clients. Every client has a unique identifier i and is denoted by cl_i. Every client communicates with the replicated service by sending authenticated requests to all service processes over an asynchronous network. The requests are processed by service processes and results sent back to clients. Clients wait for $(f+1)$ identical replies corresponding to each request they send. All clients are assumed to be non-faulty. For ease in maintaining the record of the latest request of each client, it is assumed that every client sequentially numbers its request message. Hence, every request message sent by a client is tagged with the sequentially increasing *sequence number* before signing.

Finally, we assume that a *trusted dealer* initializes the system and all the nodes with cryptographic keys and hash functions.

1.5 Contributions

The first contribution of this thesis is to formally describe the semantics of fail-signal (FS for short) abstraction. We describe the behavioural characteristics of an FS process in detail using a state transition diagram involving three states: *Working*, *Signalled* and *Failing*. While the first two assure consistent behaviour from an FS process, the failing state admits an inconsistent behaviour whereby an FS process can send a correct response to some destinations and a fail-signal to others. This is akin to a crashing process omitting to send a multicast to a subset of destinations – benign form of two-facing behaviour.

Second contribution is a major one in the form of designing a family of three total order protocols that use FS process to achieve coordinator-directed ordering. The characteristic of an FS process is studied carefully that leads to a systematic design process.

The first protocol, named *Protocol-0*, is the most basic. Its design retains the classical assumptions made in the traditional construction of an FS process using two ordinary, Byzantine-prone processes. The assumptions are two fold: (i) the constituent processes, if correct, will not find each other untimely or erroneous, and (ii) one of them will not fail. The design also makes an additional, simplifying assumption that an FS process will not enter into the failing state. Despite this, the design of Protocol-0 is not straightforward. Even though the FS process itself does not exhibit 2-facing behaviour, the asynchronous network can make it appear to behave so, by selectively delaying or re-ordering messages/fail-signals generated by an FS process. The lessons learnt thereby are used to develop other protocols. Protocol-0 is argued to be correct and is an improvement over the work in [MES03] in terms of the number of replicas used. A primitive version of Protocol-0 was presented in the form of a consensus engine for large-scale self-organizing network applications in [IE07].

The other two protocols permit an FS process to enter the failing state but relax the assumptions involved in FS process construction differently. *Protocol-I* retains (i) above but allows both the constituent processes to fail but assumes that successive failures of the constituent processes will be apart at least by a pre-specified amount of time. We note here that the inter-failure threshold period is specified only qualitatively and not quantitatively. *Protocol-II*, on the other hand, retains assumption (ii) but permits correct constituent processes to find each other untimely occasionally.

All three protocols developed are structured in two parts. *Normal* part is executed in failure-free situations. The second part is called *Install* part and is executed when the coordinator signals a failure. Normal part is a relatively straightforward algorithm with simple communication steps as typical of many other total order protocols. Whereas, Install part is rather complex and is the part that deals with inconsistencies caused by Byzantine faults. Moreover, we study all intricacies and necessities of the latter part and propose an optimized version called *Install-II* together with correctness proofs.

The third contribution is also a significant one in the form of demonstrating the practicality of the fail-signal approach for Byzantine fault-tolerant ordering. We experimentally evaluate the performance of both parts of Protocol-I to observe the trade-offs, if any. An extensive experimentation is performed in various network configurations by varying parameters like the total number of replicas, cryptographic techniques and system load. Normal part is evaluated on a LAN cluster and on a couple of emulated WAN configurations. Install part is experimented on the emulated configurations. Moreover, the performance of both parts in terms of latency (and throughput for Normal part) is also compared with that of BFT [CL99]. BFT is famous for its practicality and is the closest in terms of design to the protocols in this thesis. Finally, a crash tolerant protocol, developed by simplifying Protocol-I is also implemented. Normal part of the two Byzantine protocols, namely Protocol-I and BFT, are also compared with the crash-tolerant version to see the effect on performance when tolerance level increases from the most benign to the least. The performance study of Normal part over LAN also appears in our seminal paper [IE06]. Over all results show that the proposed protocol is optimistic and outperforms BFT in normal runs. However, a performance trade-off is inevitable in failure situations where the proposed protocol yields high latency values.

1.6 Dissertation Structure

The thesis is structured as follows:

- Chapter 2 explores the common ways of eluding FLP impossibility result while designing a consensus/total order protocol. This chapter serves as a brief survey of the solutions available in literature and focuses on the description of the body of work having similar design techniques to this thesis. The approach undertaken in

this thesis is briefly introduced and its place in the taxonomy is also identified. This chapter is supported with a discussion on the working of canonical crash and Byzantine fault-tolerant order protocols like Paxos, Chandra-Toueg algorithm and the protocol of Castro and Liskov [CL99].

- Chapter 3 consists of two parts. The first part formally introduces the fail-signal abstraction. The construction of a fail-signal process and the underlying assumptions are described in detail. A three state behavioural model is presented for fail-signal process with one of the states shown to permit a two-facing behaviour. In the second part of this chapter, a simple Byzantine fault-tolerant coordinator-based protocol, named Protocol-0, is designed using fail-signal processes. The purpose of Protocol-0 is to demonstrate basic design strategies. Protocol-0 assumes absence of the failing state and uses the simplified fail-signal process. Finally, the last section discusses how inclusion of the failing state affects the working of protocol and proposes improvements for an advanced protocol design.
- Chapter 4 adopts the recommendations forwarded in chapter 3 and discusses the design of an advanced protocol named Protocol-I. It is devoted to describing Normal part executed in failure-free situations. Description is followed by exhaustive experimental evaluation of performance of Protocol-I and BFT. Two performance studies are presented, one measuring the performance in LAN setup and other in emulated WAN configuration. A number of parameters are varied to measure and compare latency figures of the two protocols.
- Chapter 5 presents Install part of Protocol-I. This is the most sophisticated part of the protocol and deals with arbitrary behaviour of faulty nodes. This chapter also presents results from experiments to compare performance of Protocol-I with BFT in emulated WAN settings. Moreover, an optimized version of Install is proposed at the end which is also accompanied with correctness proofs.
- Chapter 6 presents Protocol-II. The system model is similar to the one used in Protocol-0, while the protocol design is closer to Protocol-I. Hence, we only discuss the amendments over Protocol-I which need to be done in both fail-signal process construction and protocol design.
- Chapter 7 concludes the thesis by presenting summary and discussing possible future works.

Chapter 2

Related Work

This chapter presents a brief survey of the most common approaches taken to circumvent the FLP impossibility. Moreover, it introduces the approach used in this thesis as a different way of solving the agreement problem. Some well-known protocols that share common underlying concepts with the protocols presented in this thesis are also explained.

We first restate the impossibility result to highlight the factors that give rise to the impossibility result. Then we describe the two most commonly used approaches to circumvent the FLP impossibility and explain how these approaches get around this result by tweaking one of these factors. These approaches include introducing randomization and restricting asynchrony. The latter is further divided into partitionable and non-partitionable classes of protocols. The fail-signal approach is introduced as the third category and the earlier work that comes under this class is also visited. Thus, a brief taxonomy of all approaches is presented.

In the rest of this chapter, we focus on explaining some canonical protocols implementing non-partitionable systems. The discussed protocols are the closest in design strategies to our protocols and will serve to explain the basic concepts well. We first describe two well-known crash-tolerant protocols; Paxos [Lam98] and Chandra-Toueg's [CT96] algorithm to present the basic principles. Furthermore, a Byzantine fault-tolerant protocol commonly referred to as BFT [CL99] is explained in detail. This protocol is best known for its high performance and is considered a standard for comparison with the protocols designed in this thesis. Finally, the only work found in literature that uses fail-signal processes named FS-Newtop [MES03] is also explained briefly.

2.1 Circumventing the FLP impossibility: Brief survey

In this section we discuss two well-known approaches that have been commonly used to circumvent the FLP impossibility [FLP 85]. To recall, we state the impossibility result below with emphasis on the three major factors that contribute to this result.

"A *deterministic* solution to reaching consensus cannot be achieved if the network is *asynchronous* and if replicas can fail even merely by *crashing*."

We observe that the three factors which together make the impossibility result hold are:

(i) deterministic solution,

(ii) asynchronous network and

(iii) failures (even just crashes).

Relaxing/removing even one of these factors will allow us to get around the Impossibility. The two most common approaches used to find a solution have been targeted to relax the first two factors. Before describing these approaches in detail, we present the concept of using oracles which is core to most of the proposed solutions. An oracle is a component which is associated with each replica process. This component is queried by the process while making a choice to reach agreement. Depending on the type of choices an oracle assists to make, and hence the factor of impossibility result it helps to relax, an oracle can be one of the two types [DSU04, EMR01]: (i) Random-Oracle (R-Oracle for short) and (ii) Suspector-Oracle (S-Oracle for short). Further details about these types are given in the sections of the approaches they are used in.

2.1.1 Relaxing deterministic guarantees

The aim of this approach is to provide a probabilistic solution to consensus instead of a deterministic one. Hence, number of steps required to complete execution are not known. The idea is to seek a solution whose execution will terminate before some time t with certain probability. The probability of termination before t goes to one, as t approaches infinity. Hence, the liveness property is weakened. Algorithms that follow this approach are called randomized algorithms and use R-oracles. R-oracle is a component that generates random numbers from some probability distribution when queried by the associated process.

In general a typical crash-tolerant algorithm executes as follows. Each process p_i initially proposes a value o_i and multicasts to all processes. It maintains the initially proposed values received from all other processes in a vector $val_i[1 : n]$ [EMR01]. It also maintains a variable est_i as its estimate of the decision value and initializes it to o_i. Then the execution goes in asynchronous consecutive rounds, updating est_i each time. The protocol progresses in such a way that estimates at all processes eventually converge to a single decision value. This eventual termination guarantees liveness. On

the other hand, the protocol ensures that the final converged value is one of the initially proposed values in val_i and hence guarantees safety.

Each round consists of two phases. In the first phase, each process p_i multicasts its current estimate est_i to all processes. It waits for first phase messages from at least majority of processes. If these messages from a majority correspond to the same value o, then est_i is updated to o, otherwise est_i is set to a special value $\perp$ which represents an invalid estimate value. In the second phase, the updated est_i value is multicast. On receiving second phase messages from a majority, p_i takes one of the following actions.

- If the estimates received in a majority of messages correspond to same value o, p_i decides o. Else,
- If at least a single message contains o as estimate, est_i is updated to o. p_i moves to next round. Else,
- If all messages contain $\perp$, a value from val_i is randomly chosen by using R-oracle and est_i is updated to that value. p_i moves to next round.

A randomized protocol can be categorized as binary or multi-valued depending on the proposed value o_i to be a binary or a multi-valued number respectively. In case of a binary protocol, the first step of proposing o_i and maintaining val_i becomes unnecessary. Hence, R-oracle acts as a coin-flip mechanism and randomly chooses 0 or 1 in phase 2.

A large number of crash- [Ben83, EMR01, CMS89] and Byzantine fault-tolerant [Rab83, CD89, AH90, FM97, KS01] randomized algorithms have been proposed. Since, this thesis is concerned with deterministic solution to reach consensus, further details about randomized protocols are not discussed. Interested reader is referred to [Asp03] which presents an extensive survey of randomized protocols.

2.1.2 Relaxing asynchrony

The second, well-known approach to circumvent FLP impossibility is to relax the strict assumption about the asynchrony of the communication network. Hence, some timing restrictions are applied on the asynchronous model with a caveat that there may be periods of instability during which the applied restrictions may not hold and the system should be designed to behave in a predictably safe manner during such periods. The restrictions take the form of choosing *likely* bounds on message communication delays and processing speeds and it is assumed that there will be a time after which these

bounds will hold for sufficiently long amount of time. However, this assumption is only made to guarantee liveness. Hence, the system is assumed to go through periods of stability during which the protocol is guaranteed to terminate. The notion of stable periods [DLS88, CF99] is referred to as well-behaved executions in [KR01]. The latter also presents bounds on number of communication steps for crash-tolerant consensus protocols in both synchronous and asynchronous network models.

This class of protocols uses S-oracles to make progress. S-oracles are special modules that wait on timeouts for some "heartbeat"/"I am alive" messages to be received periodically from other processes. When the expected message from a given process is missing for a few, consecutive times, then that process is suspected. These timeouts are chosen based on the experience with the network performance so that (i) incorrect suspicions are minimal and, (ii) any two correct and connected processes that suspect each other, *eventually* begin to see each other correct and connected. With this assumption of *eventual* correctness of timeout bounds used, consensus can be solved in asynchronous environments.

We divide the protocols that use relaxed asynchrony assumptions into two categories based on the steps taken by the correct processes on receiving the list of suspected processes from their respective suspectors. In the first category, called *partitionable system*, the suspected processes are considered failed and are excluded by the unsuspected processes from the group of processes that need to reach agreement. Hence, suspicions define membership of the group of processes that can participate in protocol execution. Whereas, in the second category, called *non-partitionable system*, the suspected processes are allowed to participate but the role they play in protocol execution changes. The two systems are explained below.

2.1.2.1 Partitionable System

The protocol execution moves through a sequence of *views* at every process p_i, where a view is a set of processes which are deemed to be correct and connected to p_i. Every process queries an S-oracle to get the list of suspected processes. The core specification of a partitionable system is that a suspicion leads to the exclusion of the suspected from the current view only through agreement being reached with the unsuspected: the process p_i whose current view is V will construct the next view (V- S) only after every process in (V- S) has announced its suspicion on every member of S. However, to avoid inconsistencies, once a process has announced its suspicion to other members, it should

not unilaterally reverse its suspicion, even if it sees an overwhelming evidence to do so. By transforming suspicions into "agreed failures", a way (albeit imperfect) is found to get around the inherent inability to distinguish slow processes from crashed ones: a suspected process may indeed be functioning or communicating slowly, but it will be treated as a crashed one if, say, every one else in the group suspects it. With this imperfect way of 'distinguishing' crashed ones from slow ones, consensus can be solved not necessarily among *all* correct processes, but perhaps within subgroups of mutually unsuspecting processes.

Reaching agreement on the suspected is central to partitionable system. It also helps sort out inconsistent failure suspicions held by mutually unsuspecting members. When p_1 is seeking the exclusion of p_2 through agreement, if a third process p_3 (which p_1 does not suspect) does not suspect p_2 then p_3 can refute p_1's suspicion of p_2; p_1 can then either reverse its suspicion of p_2 or add p_3 to its suspect list. Further, an incorrect suspicion need not be symmetric: when p_1 suspects p_2, p_2 may not suspect p_1. It can also be intransitive: p_1 suspects p_2, p_2 suspects p_3, but p_1 does not suspect p_3. This asymmetry and intransitivity in the suspicion chain are sorted out when agreement on suspicions is waited for. But the agreement on suspicions cannot prevent the formation of concurrent, overlapping views as explained by the following example.

Say, a partitionable replicated system consists of processes p_1, p_2, p_3 and p_4 as illustrated in figure 2.1. At system start, all processes work in the same view consisting of all four processes. Assume that p_1 suspects p_4 and the suspicion is confirmed by p_2 and p_3, as a result, the view of p_1 is changed to exclude p_4 (p_2 and p_3 will also update their views to exclude p_4). On the other hand, p_4 has not (yet) suspected anyone, so its view remains unchanged. This is however a temporary situation, as p_4 will eventually suspect p_1, p_2 and p_3 (since these have stopped communicating with p_4) and its view will be updated to exclude them. It is proved in [BDG+95] that no non-blocking implementation can guarantee absence of overlapping concurrent views.

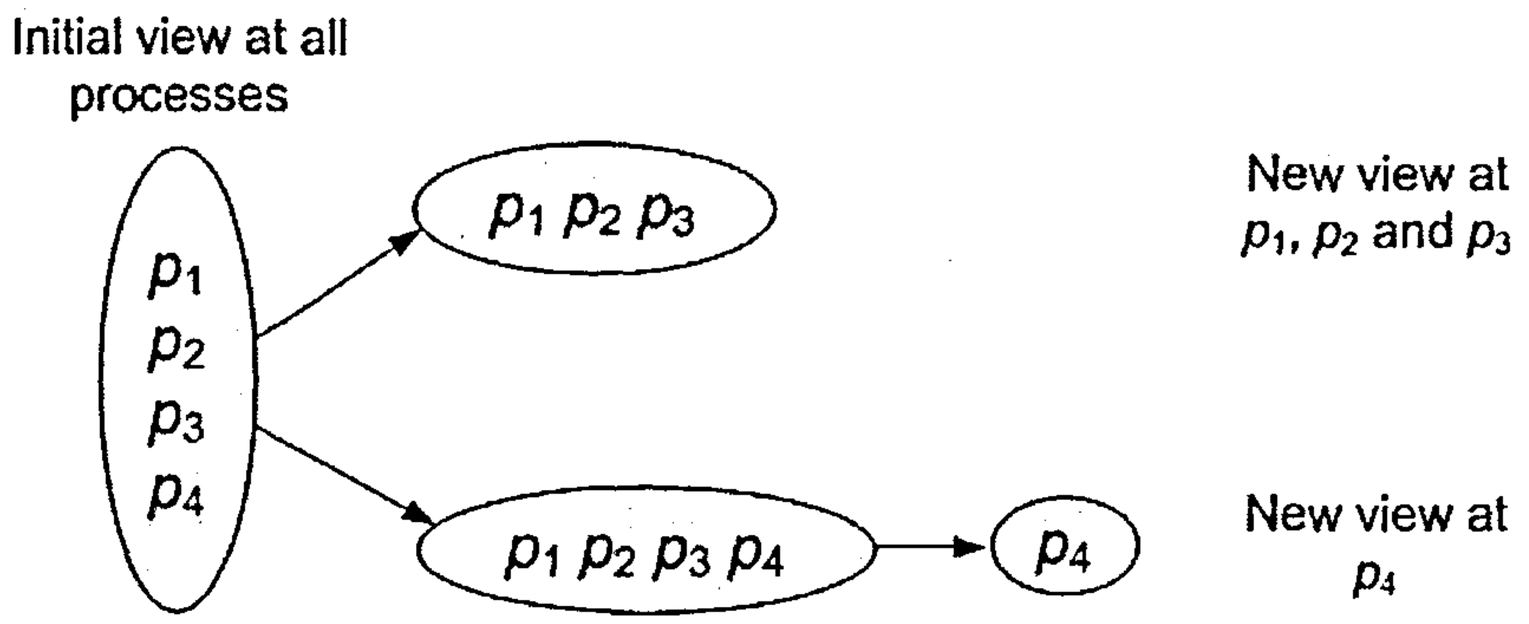


Figure 2.1. Concurrent Overlapping Group views

The most effective way to use a partitionable system is to use appropriate timeouts so that a group of correct and connected processes do not unnecessarily get divided into multiple subgroups also called *virtual* partitions with processes in one subgroup suspecting the ones in the other. In the worst situation, when the timeouts used turned out to be the most inadequate ones, the group gets split into singleton islands, whereby each correct process considers itself as the "sole" surviving process in the group. This comes about because a process has to reach agreement on the suspected only with the unsuspected ones; if a process suspects every other process then it is entitled to decide a view that contains only itself! In this situation, the system can only serve selective requests e.g., read only requests.

In summary, the partitionable system approach restricts asynchrony in the following way.

Timeouts are chosen to be so pessimistically large that false suspicions are indeed rare. This is often expressed as: correct processes do not suspect each other "capriciously" (see [RB91]).

When correct processes are rarely eliminated, provisions can be made to permit them to rejoin the departed group and participate in protocol execution. However, allowing processes to rejoin is itself a non-trivial task and needs divergence in the state to be reconciled. This is addressed in [BBD97, ANB+07].

Good amount of work has been done based on partitionable service [BDM95, BDM97, BBD97, DM96, ADK+92]. [EMS95] presents a crash-tolerant group communication protocol named *Newtop* in the context of multiple, overlapping groups. Newtop provides variety of services: symmetric total order, asymmetric total order, reliable multicast and partitionable group membership. [MES03] extends this work and

converts Newtop into a Byzantine fault-tolerant system, called *FS-Newtop*. FS-Newtop is discussed in a later section.

2.1.2.2 Non-partitionable System

In this approach the protocol execution moves through views each containing both correct and suspected processes. Hence the membership of the views is static and output from suspectors is not used to decide next view's membership. Instead, suspicions are used to decide the role a given process will play in protocol execution as described below.

Most of the proposed solutions in this class tend to be coordinator-based. The idea is to go through rounds or views of execution. Each round has a process pre-defined as the *coordinator*. Decision value is proposed by the coordinator. This value may be among the ones initially proposed by other processes or may even be generated by the coordinator itself. Of course, the coordinator may have failed or may fail while imposing its decision value. So, each process monitors the reliability of the coordinator by using S-oracle. If the coordinator is not suspected, a process accepts the value proposed by the former. Once the proposed value is known to be accepted by a majority of processes, it is considered as the decided value. Alternatively, suspecting processes move on to the next round with a new coordinator and repeat the process. The core specification of a Non-partitionable approach is that there should be no concurrent rounds and the i^{th} round executed by any correct process should be the same. However, implementing this is not trivial.

[CT91, CT96] introduced the concept of failure detectors for crash fault-model. Like, S-oracle, a *Failure Detector-oracle* (FD-oracle) is a component which the associated process can query to find out a list of failed processes. But recall that it is impossible to *correctly* identify a failed process [FLP85]. Hence, these detectors are unreliable and can make mistakes. [CT96] present a range of failure detectors classified with respect to two properties.

1. *Completeness* – It requires the failure detector to eventually suspect every crashed process.
2. *Accuracy* – It defines the degree to which a failure detector can make mistakes.

Eight classes of failure detectors have been introduced based on two completeness and four accuracy properties. Moreover, the minimal number of processes needed to reach consensus for these classes has been studied. Each class has been

mapped onto a set of system requirements. The requirement set becomes weaker for weaker failure detectors. Failure detectors also use timeouts to suspect failures, and, unlike failure suspectors, can unilaterally reverse their suspicions if they see evidence for that.

[CHT96] shows that the class of *Eventually Strong Failure Detectors* $\Diamond S$ is the weakest class needed to solve consensus. $\Diamond S$ is characterized by the following two properties.

Strong Completeness - Eventually every crashed process is permanently suspected by every correct process.

Eventual weak Accuracy – There is a time after which some correct process is never suspected.

Reaching consensus becomes possible because the eventual weak accuracy property ensures that eventually no correct process suspects at least a correct process which can play the role of the coordinator. On the other hand, strong completeness ensures that the coordinator chosen by all correct processes is not among the crashed ones.

Due to the two properties of $\Diamond S$ failure detectors, execution will eventually terminate and temporary periods of instability can only have the effects of delaying the termination of the protocol execution, and cannot make the execution terminate incorrectly nor make it non-terminatable.

A large body of work in literature has been dedicated to study various classes of FD-oracles. Issues like finding equivalence between FDs of different classes in terms of their respective power and capability [e.g., CGS00, FMR07], transforming FDs of one class into another and designing algorithms using FDs to solve agreement problems have been researched. Moreover, a new class of FD, denoted by $\Diamond S_k$ has been introduced by Mostefaoui et. al. [MR99b], which is essentially weaker than $\Diamond S$ class. $\Diamond S_k$ has limited scope accuracy and requires only at most k processes to not suspect a correct process after some time. Transformation of $\Diamond S_k$ and $\Diamond S$ has been shown in [AFM+04, MRR+06]. Thus, the concept of FD-oracles has been used to develop a large number of crash-tolerant consensus protocols [AT96, DFK+96, HN99]. A couple of the earliest protocols including the one proposed in [CT 91] are explained in detail in subsection 2.3.

Furthermore, adaptations of FD have been proposed to tackle with Byzantine failures. For example, Malkhi and Reiter [MR97] introduced $\Diamond S(bz)$ class and the notion of *quiet* process, Doudou and Schiper [DS97] proposed an advancement and defined $\Diamond M$ class and *Mute* process and Baldoni et. al. [BHR+99] further improved on the previous work and introduced notion of *reliable arbitrary behaviour detection module*.

2.2 Our Proposed Approach

We aim to circumvent FLP impossibility by avoiding the third factor that makes reaching consensus impossible i.e., quiescent failures. We recall that the core difficulty behind the impossibility result is the inability to distinguish a slow process from the one that has failed quiescently. However, if the failing process is somehow made to "announce" its imminent failure, this result will cease to hold. Furthermore, if failures are restricted to stopping only, then the decision gets even more simplified. This idea was first explored in [MES03]. This work introduces abstract processes that signal on failure. This abstraction is called *Signal-on-Fail* or *Fail-Signal*. The idea is to *construct* a process that behaves as a single process but consists of multiple redundant processes. This internal redundancy helps in circumventing the impossibility by detecting and signalling the failures. Although expensive in terms of increased redundancy requirement, this approach increases the tolerance level of the system by masking Byzantine failures. We show that adhering to the optimal redundancy needs careful protocol design.

The approach to constructing fail-signal processes is similar to that of well-known *fail-stop processes* [SS83] and *fail-silent processes* [BES+96]. A fail-stop process itself is made up of several fail-independent, fail-Byzantine, redundant processes which operate in parallel, check on each other's outputs and endorse each other's matching outputs. Only the endorsed outputs are treated as the outputs of the (abstract) fail-stop process, and the endorsement is indicated through digital signatures that are assumed to be non-forgeable. On detecting a failure of any of their counter-parts, the redundant processes stop all activities related to fail-stop computation and indicate this stopping by outputting verifiably-endorsed *fail-signal* messages. Thus, a fail-stop process, or a collection of processes implementing that abstraction, either outputs verifiably-endorsed messages of correct contents or stops functioning after signalling its stopping. That is, it exhibits only *crash* failure semantics and additionally

*fail-signal*s its own imminent crash. A fail-silent process also works in a similar fashion except that it halts without announcing its failure. Fail-stop process construction however requires more redundancy (2ϕ+1 processes if at most ϕ can fail simultaneously) because it also provides several other desirable properties. For example, the state of a fail-stop process at the time of failure is available in a stable storage and can be queried anytime after the failure. This is not the case for a fail-silent process which is constructed out of (ϕ+1) processes only.

A fail-signal process is a mid-way technique which uses signalling feature of fail-stop process combined with the lower redundancy of the fail-silent processes. Hence, it does not provide stable storage. We refer to the earlier work of [MES03] that involves porting fail-signal processes on a crash-tolerant group communication protocol to present the proof of concept. To demonstrate the power of this abstraction, we intend to explore the design of ordering protocols that are practical in performance and optimal in redundancy. Particularly, we focus on deterministic coordinator-based protocols. Therefore, to lay foundations and to explain basic concepts, the rest of this chapter is devoted to a few canonical protocols that use design strategies similar to ours.

Figure 2.2 summarizes the three major approaches taken to circumvent FLP impossibility; randomization, restricted asynchrony and fail-signal. Restricted asynchrony approach among the three is further divided into two categories; partitionable and non-partitionable. Canonical protocols in each of these categories are also noted each for crash and Byzantine fault models. We note that for the partitionable system class, no Byzantine fault-tolerant protocol other than the Total algorithm [MMA93] could be found in literature as surveyed by [DSU04]. It is important to point that no crash-tolerant protocol is shown in fail-signal category. This is due to the fact that fail-signal approach is inherently capable of tolerating arbitrary failures. Crash being the most benign class of faults is therefore automatically covered. In fact, designing a protocol using fail-signal processes, where faults are restricted to stopping-only becomes easier.

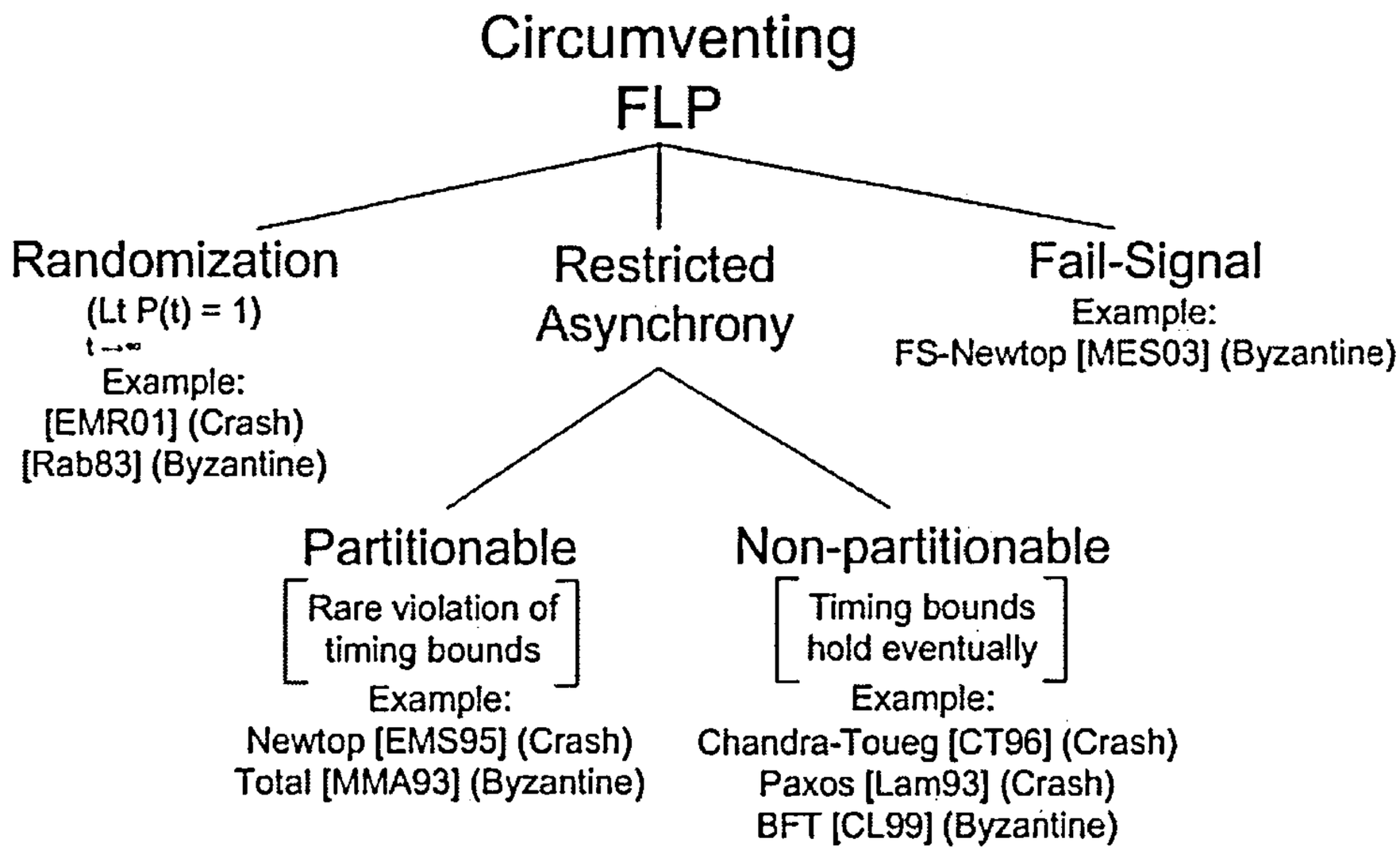


Figure 2.2. A Taxonomy of approaches to circumvent FLP impossibility

2.3 Non-Partitionable Crash-Tolerant Protocols

In this section we explain a couple of well-known crash-tolerant total order protocols that use non-partitionable system approach; Paxos[Lam98, Lam01] and Chandra-Toueg[CT91, CT96] algorithms. The original presentations of these algorithms demonstrate how the algorithms solve the consensus problem and then provide extensions for solving total order broadcast problem. We present the total order versions here. Being crash-tolerant, these algorithms assume that majority of the processes are correct requiring total number of processes to be $n = (2f+1)$. Moreover, we discuss the similarities and differences between the two protocols. Note that some of the notations used here are different from the ones found in original texts. This is done to establish relevance between these protocols. Moreover, it is assumed that each process p_i is uniquely identified by process number i.

2.3.1 Paxos: The Part-Time Parliament

Paxos is a coordinator- (also called leader) based protocol that uses FD-oracle Ω [CHT96]. Ω is associated with every process and determines whether the local process should act as leader. Ω selects a new process as leader when the current leader is suspected to have crashed. Paxos is designed to tolerate the situation where multiple processes consider themselves leaders at the same time. We note here that Ω has been

shown to be equivalent to $\Diamond S$ FD-oracle. That is, any one of these can be transformed into the other [CHT96, Chu98, MRR+05].

[Lam98] describes the protocol in terms of following three roles that each process can play:

(i) **Proposer (leader)** – The process which proposes a value to be decided.

(ii) **Acceptor** – The process which participates in the decision process.

(iii) **Learner** – The process which learns about the decided value.

The original algorithm can be optimistically broken down into two parts, as suggested in [Lam01]. First one is executed by every process in normal failure-free situation with one single leader. The second part is executed when leader fails before decision is reached and a new leader is selected by Ω. These parts are illustrated in figure 2.3, assuming p_1 as the leader process.

Part 1

(a) The leader proposes an order number *o* for a request and sends the proposal, called *accept* message, to all processes tagging it with a unique proposal number *rnd#*. The proposal number is to track the re-tries performed by the leader when the first proposal fails to reach decision. The proposal number is only assigned in ascending order and each leader has its own mutually exclusive pool of infinite proposal numbers.

(b) An acceptor *accepts* and acknowledges the proposal by sending *ACK* message if it has not acknowledged a higher proposal numbered *prepare* message of part 2 (see below) for the same request. Otherwise, a negative acknowledgement, *NACK* message is sent back. In case of a single leader, this step will always result in transmission of ACK.

(c) If the leader receives ACK messages from a majority of the acceptors, it sends a *success* message to all processes. On the other hand, if it receives even a single NACK, it re-starts the process by preparing another proposal. When a process receives success message, it decides the proposed value.

Part 2

(a) When the new leader gets selected by Ω, it needs to learn about all order numbers that have been decided by any process in the system. It therefore chooses a new proposal number *rnd#* from its pool and sends a *prepare* request to all processes for all missing order numbers. *rnd#* is chosen to be higher than the largest proposal

number known to be sent by any predecessor coordinator for all undecided orders. The prepare request carries all missing/undecided order numbers and is sent to seek the highest proposal number less than *rnd#* for each of these order numbers that has been accepted by any acceptor.

(b) An acceptor sends ACK message to the new leader if it has accepted a proposal with number less than *rnd#* for any of these order numbers during part 1 execution (with any leader). ACK message contains the highest numbered proposals accepted for these order numbers and the corresponding requests. With this reply, the acceptor restricts itself for not accepting any proposal with number less than *rnd#*, as mentioned in step (b) of Part 1. Alternatively, a NACK message is sent. This NACK message can represent two situations: (i) No proposal was accepted for any order number, (ii) No proposal less than *rnd#* was accepted for any order number. In the second case, the NACK message can be made to carry the proposal number higher than *rnd#* corresponding to each order number for which an accept message was sent. This will guide the new leader which can attempt again with a new large enough proposal number.

(c) After waiting for responses from a majority, the new leader starts execution of part 1 for all these order numbers. It prepares an accept message for every order number by selecting the request corresponding to the highest proposal number reported. In case no request is reported for an order number, the new leader fills holes by proposing a special "no-op" request for these orders.

An Example for Part 2

We quote an example from [Lam01] to elaborate Part 2 execution. Suppose that a leader has just failed and a new leader p_c has been selected by Ω. Say p_c knows about order assignments 1-134, 138 and 139. It will therefore execute part 2 for orders 135-137 and all orders above 139 to find out if these are accepted by any process. It can use a single sufficiently higher proposal number *rnd#_1* from its pool to construct a prepare message for all these missing orders as mentioned in (a) of Part 2 above. Suppose it received ACKs from at least a majority of processes containing requests with order numbers 135 and 140 only. Since at least a majority has not accepted orders 136, 137 and higher than 140, these orders could not have been decided. To fill in the gaps, p_c proposes 136 and 137 to be assigned to no-op requests. Hence p_c can start executing

part 1 of the protocol for every new request with 141 to be the first proposed order number.

Another possibility is that p_c receives a NACK containing a proposal number *rnd#'*, *rnd#'* > *rnd#_1*. This implies that the sender process has accepted at least one of these missing orders for *rnd#'*. Therefore, p_c chooses a proposal number *rnd#_2* from its pool such that *rnd#_2* > *rnd#'* and restarts execution of part 2.

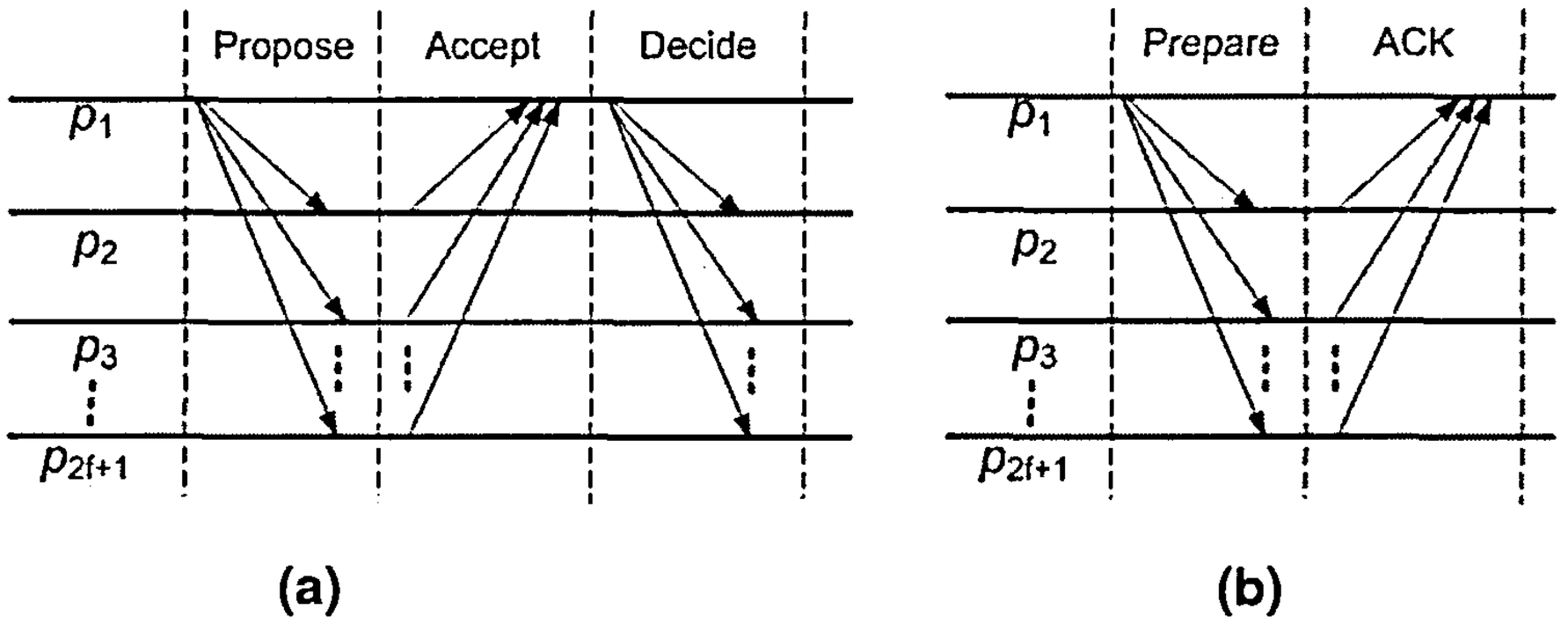


Figure 2.3. Paxos Algorithm steps (a) Part 1 (b) Part 2

Of course, there can be multiple processes considering themselves as leader at any given time. The protocol ensures safety by restricting all acceptors to accept only proposal numbers higher than the largest numbered prepare message acknowledged. Hence, inconsistencies will not arise between leaders as every leader gathers the latest accepted messages from a majority before proposing a value. However, these multiple leaders can keep on proposing higher numbered proposals (accept requests) too often resulting in none of the proposals arriving decision stage. Hence, progress can only be guaranteed by assuming some timing bounds on asynchrony, as mentioned in subsection 2.1.2. This will make the leaders wait before sending a new proposal for a timeout that is sufficient to get some earlier proposal decided.

More work has been done recently to improve the Paxos algorithm, usually called classic Paxos now, in terms of number of communication steps. A variant of classic Paxos, named Fast Paxos appears in [Lam06]. In short, Fast Paxos allows the decided value to be learnt in two communication steps when there is no collision, i.e., multiple values are not proposed concurrently. (The basic idea is same as [BGM+01].) However, it needs higher number of replicas and may lead to higher latency in case of collisions. [JMM07] and [DMS06] present performance study of the two versions for WANs. The latter also propose an algorithm that is claimed to perform better under

collisions. (Byzantine fault-tolerant version of Fast Paxos has also been proposed and can be found in [MA06].)

2.3.2 The Chandra and Toueg Algorithm

Chandra-Toueg algorithm is also a coordinator-based protocol [CT91] that uses FD-oracle $\Diamond S$ as described above. An optimized version has been presented in [Sch97].

For every request, each process p_i proposes a value o_i as the order number. p_i also maintains a variable est_i as its estimate of the decision value and initializes it to o_i. The execution goes through a series of sequentially numbered rounds, starting from round 1, until a decision is made. Each round *rnd#* has a pre-determined process p_c as coordinator. If n is the total number of processes, the relation is expressed as follows.

$$c \leftarrow (rnd\# \bmod n) + 1$$

This scheme is also known as *rotating coordinator paradigm*. Each round is divided into following four phases, as illustrated in figure 2.4. (Figure assumes p_1 as the coordinator process.)

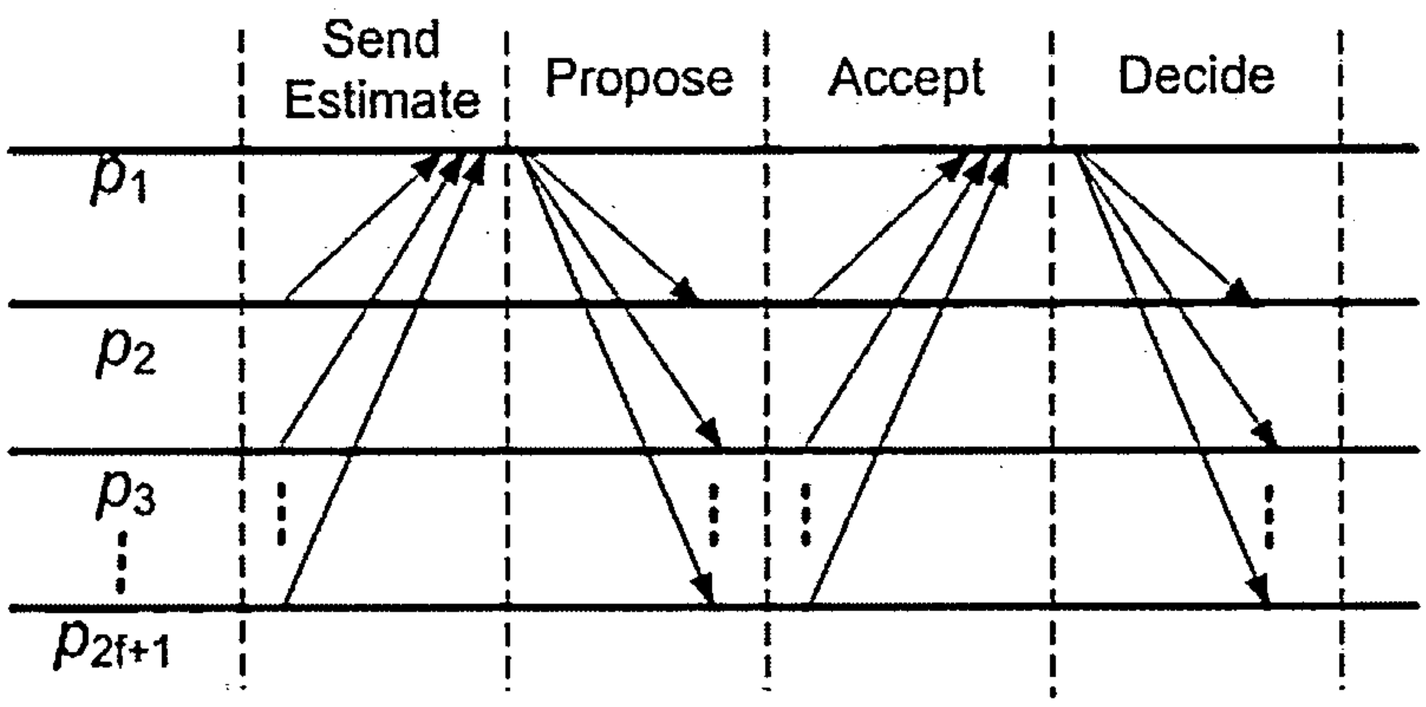


Figure 2.4. Chandra and Toueg's Algorithm steps

Phase 1 – Every process sends its current estimate of the decision value est_i to its current leader along with the latest round number in which the estimate was updated.

Phase 2 – The coordinator waits to receive estimates from a majority of processes. It then selects the latest updated value and proposes it as the new estimate. It sends this message to all.

Phase 3 – Every process waits until it receives either the new proposal from the current coordinator or an alert from its FD-oracle suspecting the coordinator. In the first case, it updates its estimate to the new proposal and sends an ACK message to the coordinator.

In the second case, it sends a NACK message to the coordinator indicating its suspicion and moves on to next round.

Phase 4 – The coordinator waits until it receives either an ACK message from a majority of processes or a single NACK message. In the first case, it prepares a decision message with the proposed estimate and reliably broadcasts the message to all. A process decides when it receives this message. While in the second case, the coordinator suspends current round execution and moves to the next round.

Since every process goes though sequentially numbered rounds responding to only one coordinator at a time and the coordinator chooses the latest updated estimate as the decision value, the protocol guarantees safety. The algorithm will terminate in first round, if the coordinator is considered alive by at least a majority of processes. In case of false suspicions, processes may keep moving to next rounds too often without reaching decision. Hence, like Paxos, liveness is only guaranteed when the communication and processing delays remain stable for sufficiently long duration.

2.3.3 Paxos vs. Chandra-Toueg Algorithm

In this subsection, we compare the two protocols and list main similarities and differences. Experimental comparison of the two protocols is presented by Urban et. al. [UHS+04]. They have proposed a number of optimizations for the two protocols and studied latency values in different classes of runs. The results show that although the two protocols have comparable performance in normal situations, the Paxos algorithm performs significantly better in cases with multiple correlated crashes or frequent false suspicions.

2.3.3.1 Similarities

1. Both algorithms go through iterative executions to decide a single value. Each iteration is associated with a coordinator process. Consecutive iterations are represented by proposal numbers in Paxos Algorithm (PA), while they form rounds in Chandra-Toueg's Algorithm (CTA).
2. Both algorithms facilitate recovery of processes from crashes, given that some state variables are retained in a stable storage.
3. Both algorithms need aid of FD-oracles, Ω in PA and $\Diamond S$ in CTA, and assume that the timing bounds used hold for sufficiently long duration to guarantee liveness.

2.3.3.2 Differences

1. **Proposal generation:**

   PA - Coordinator proposes the estimate. Hence, it proactively starts an iteration by proposing a value.

   CTA - Every process proposes its own estimate and the coordinator chooses one of these proposals. Hence, despite being the coordinator of an iteration (round), the process has to wait to receive estimates from at least a majority of processes, before proposing a value.

2. **Iterative executions**

   PA – A process does not necessarily need to go through consecutive iteration (proposal) numbers. It responds to the proposals send by various leaders, but only in increasing order.

   CTA – A process executes iterations only sequentially. It only moves to next iteration once it suspects the current coordinator.

3. **Current coordinator**

   PA – A process is not restricted to respond to a single coordinator at a time. It can acknowledge *accept* and *prepare* requests of two different coordinators subject to the restrictions mentioned in Part 1(b) and Part 2(b) of PA (see subsection 2.3.1).

   CTA – A process only responds to messages from the coordinator of the iteration it is executing.

4. **Communication steps**

   PA – Each iteration consists of three communication steps: Propose, Accept and Decide.

   CTA – Each iteration consists of four communication steps: Send estimate, Propose, Accept and Decide.

2.4 Non-Partitionable Byzantine Fault-Tolerant Protocols

The consensus problem has been given equal importance in the literature for Byzantine fault-tolerance. Many attempts have been made to circumvent the impossibility by using the above mentioned approaches. However, comparing crash-tolerant protocols with the Byzantine fault-tolerant ones, we observe that the foundational design techniques are the same. Indeed many algorithms can be seen as Byzantine fault-tolerant extensions of

Paxos [CL99, YMV+03, MA06]. This section mainly discusses the work available in the literature that is most closely related to the schemes presented in this thesis.

2.4.1 BFT

BFT is a Byzantine fault-tolerant state machine replication algorithm which can replicate a deterministic service [CL99]. BFT is an optimistic protocol and is best-known for its performance in failure-free situations. We describe in much detail how BFT totally orders clients' requests. BFT considers the same system context as taken in this thesis (presented in section 1.4). Any additional or different assumptions will be explicitly specified.

BFT provides safety property [Lyn96] given that the total number of faulty replicas f does not exceed $(n\text{-}1)/3$ over the life time of the system. Therefore it needs total number of replicas, $n \geq (3f+1)$. However it relies on network synchrony to provide liveness property. It assumes that $delay(t)$ does not grow faster than t where $delay(t)$ is the time between the moment t when a message is sent for the first time and the moment when it is received by its destination.

2.4.1.1 The Algorithm

The replicated service consists of a set of n replica processes. For simplicity it is assumed that $n = 3f+1$. Each process is identified by a unique integer i in $\{0, \ldots, n-1\}$ and is denoted as p_i. Each process executes service application; maintains service state and implements its operations. Clients send requests to execute a service operation to the processes. All correct processes execute the requested operations in identical order which is computed by executing order protocol. Results are delivered back to clients. Clients wait for $(f+1)$ identical replies from different processes.

BFT uses primary-backup scheme [AD76] and quorum replication techniques [Gif79] to enable processes execute client requests in identical order. Processes go through succession of configurations called *views*; which are numbered consecutively. In each view, one process is the primary (same as coordinator) and all others are backups. Process p_c is the primary of view v such that $c = v \bmod n$. Note that views of BFT should not be confused with membership views of partitionable system. Like Paxos algorithm, BFT can be divided into two parts. First one is called *Normal-Case* operation and is executed in failure-free situation. The second part is called the *View-Change* operation and is executed when primary is suspected to have failed. Note that

BFT implements a non-partitionable system. Each view that the protocols moves through, contains all processes in the system i.e., correct as well as suspected. Hence, process membership in all views is static.

In normal-case operation, the primary assigns a unique order number *o* to each request and sends the assignment to backups. Backups agree on the order if it is found valid. Validity checks involve verification that the number has not been assigned to any other request by the same primary, that the primary is not leaving gaps in order numbers, that the primary has not stopped assigning order numbers to new requests etc. Each backup accepts the proposal if validity checks are successful. Once the assignment is confirmed (confirmation process in the form of three communication phases is explained below), service operations are executed in the assigned order and results are sent back to clients. Alternatively, if a backup suspects the primary, it triggers view-change process to select a new primary.

BFT uses concepts of quorums and certificates to tolerate Byzantine faults. A quorum is a set of at least $(2f+1)$ processes with following properties

Intersection property - Any two quorums have at least one correct process in common.

Availability property - There is always a quorum available with no faulty process.

BFT uses certificates to keep a proof of each step of the protocol. Quorum certificate is a set of messages with one from each process in a quorum regarding a particular information/step of protocol. Weak certificates are sets with messages from at least $(f+1)$ different processes.

2.4.1.2 Communication with Client

Client cl_j requests execution of an operation *op* by sending a $\langle REQUEST, op, t, j\rangle_{\sigma j}$ message to the primary, as shown in figure 2.5 with p_0 acting as primary process. BFT uses timestamp *t* to totally order requests of each client i.e. request with greater *t* value will be assigned higher order number. It is also used to ensure exactly-once semantics for the execution of requests. Once a request is executed, the result is sent back by each process p_i to the client in the form of $\langle REPLY, v, t, j, i, rt\rangle_{\sigma i}$ message, where *v* is the current view number and *rt* is the result of execution.

2.4.1.3 Normal-Case Operation

The three phases of the protocol, which all correct processes execute during fault-free situation, are named *pre-prepare*, *prepare* and *commit*, as illustrated in figure 2.5(with *c*

= 0). The primary p_c enters pre-prepare phase by assigning a unique order number o to a request m from a client. The assignment is sent to all backups in the form of a $\langle\langle PRE\text{-}PREPARE, v, o, D(m)\rangle_{\sigma c}, m\rangle$ message, where v is the current view number. A backup accepts a pre-prepare message if

- It is in view v.
- It can verify the authenticity of the message.
- o is between a low water mark h and a high water mark H.
- It has not accepted a pre-prepare message with same v and o but different request m'.

Water marks are defined to prevent a faulty primary from exhausting the space of order numbers by selecting a very large one. Details on how values are chosen for h and H are given in a later subsection.

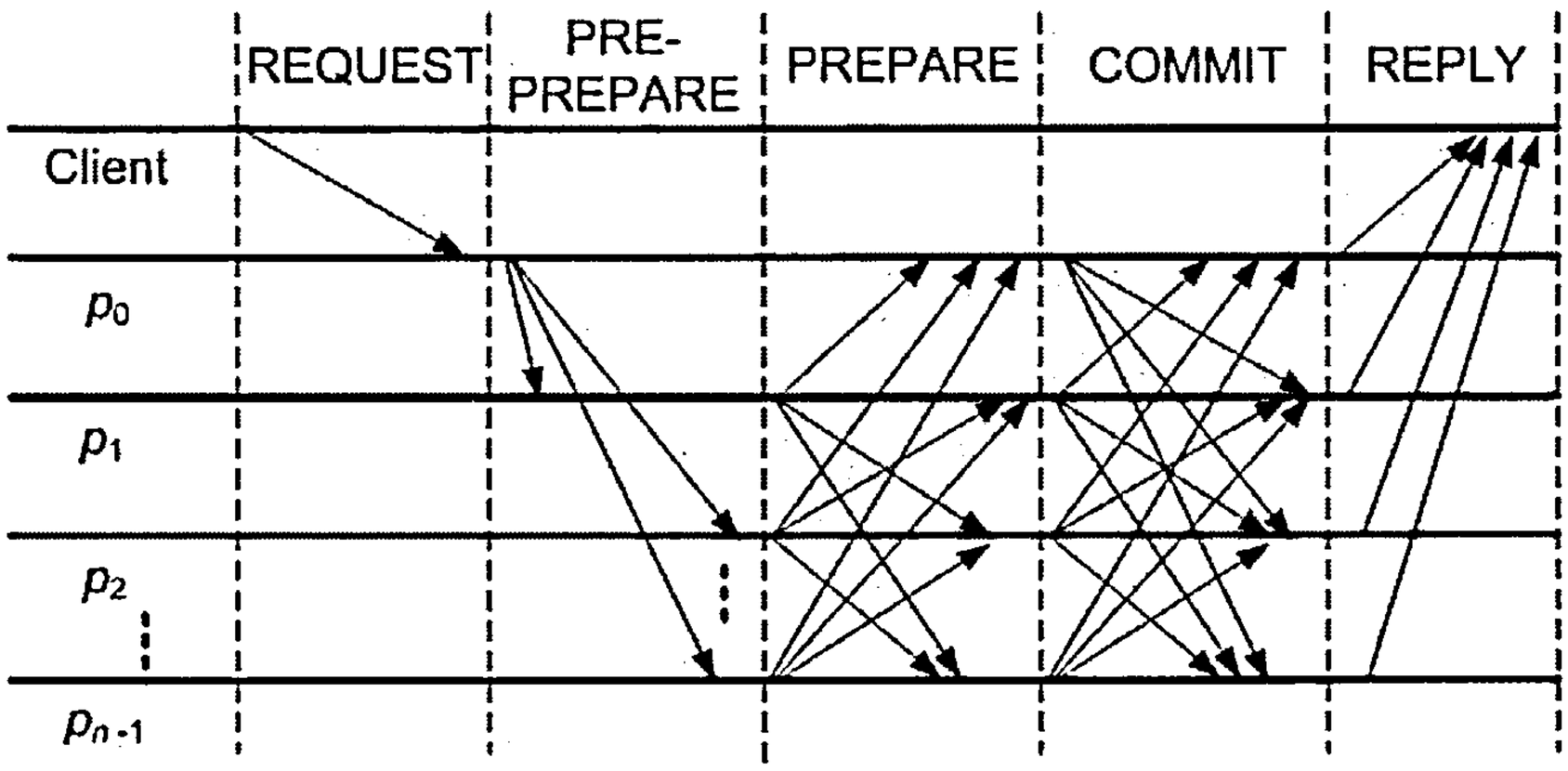


Figure 2.5. Normal Case Operation of BFT

On accepting a pre-prepare message, backup p_i enters prepare phase and multicasts $\langle PREPARE, v, o, D(m), i\rangle_{\sigma i}$ to all processes. It also stores both pre-prepare and prepare messages in its log. Once process p_i collects quorum certificate with a pre-prepare and $2f$ matching prepare messages containing same v, o and $D(m)$ from different processes, called *prepared certificate*, it enters commit phase and multicasts a $\langle COMMIT, v, o, D(m), i\rangle_{\sigma i}$ message to all processes. Receipt of a quorum certificate with $(2f+1)$ matching commit messages containing same v, o and $D(m)$ from different processes, called *commit certificate*, marks the end of commit phase. p_i is said to have request m committed when it has both the prepared and commit certificate for m. p_i then executes the operation requested in m after all lower order numbered requests are

executed. Result is then sent back to the requesting client in the form of REPLY message mentioned before.

2.4.1.4 Garbage Collection

Since each process stores prepared and commit certificates for each request, log size can increase enormously with time. These certificate messages cannot be discarded even after execution until it can be ensured and proved, when needed, that the corresponding operation has been executed by at least (f+1) correct processes. Moreover, a non-faulty process should also be able to provide the up to date service state with validity proof to another process that missed the messages which are discarded by all non-faulty processes.

Each process p_i stores service state, also called *checkpoint*, after executing every constant number of requests, defined in the beginning. It multicasts a $\langle CHECKPOINT, o, D(state), i\rangle_{\sigma i}$ message, where o is the order number of the last executed request and $D(state)$ is the digest of the state. Each process collects a certificate with (2f+1) checkpoint messages having same o and $D(state)$ from distinct processes. This certificate, called *stable certificate*, proves correctness of the checkpoint and renders it stable. All prepared and commit certificates belonging to requests with order numbers below o are then discarded.

The check point mechanism is used to define water marks mentioned earlier. Low-water mark h is equal to the order number of the last stable checkpoint and high water mark $H = h + k$, where k is big enough so that processes do not stall waiting for a check point to become stable.

2.4.1.5 View Changes

View changes are triggered by suspicions from backups. These suspicions are based on timeouts which provide liveness to the system in case primary is suspected not to be performing its job in a timely manner. Each process p_i maintains a timer to monitor the time a request remains unexecuted. It sets the timer when it receives a request which has not been executed yet. The timer is reset when the request gets executed but is set again if there exists another unexecuted request. In case of a time out, a request to move on to next view (v+1) is multicast in the form of a $\langle VIEW\text{-}CHANGE, v+1, o, CP, PC, i\rangle_{\sigma i}$ message, as shown in figure 2.6. Here o is the order number and CP is the stable

certificate of the latest stable checkpoint, and *PC* is a set containing prepared certificates of each request prepared at p_i with order number greater than *o*.

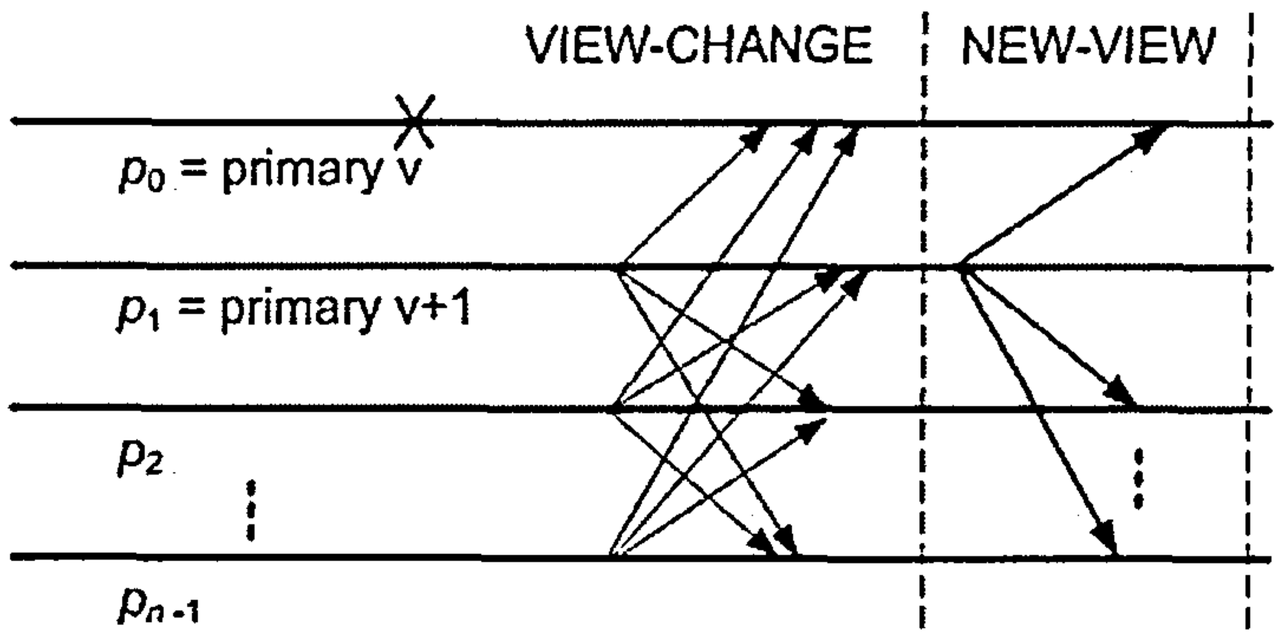


Figure 2.6. View-Change Protocol

New primary p_c for view (*v*+1) collects view-change messages for the new view from (2*f*+1) different processes also called *new-view certificate VC*. p_c analyses *VC* to determine the order number of the latest stable checkpoint *min-s* and the highest order number of a pre-prepare message *max-s*. It then prepares a set *O* containing pre-prepare messages for each number *o*, *min-s* < *o* ≤ *max-s* in the following way.

– In case p_c finds a pre-prepare message for request *m* with order number *o* in *VC*, it uses that to produce $\langle PRE\text{-}PREPARE, v+1, o, D(m)\rangle_{\sigma c}$ and adds it in *O*. Note that the latest view pre-prepare message will be chosen for every *o* if there are multiple available in *VC*.

– If no pre-prepare was found in *VC* for *o*, a new $\langle PRE\text{-}PREPARE, v+1, o, D(null)\rangle_{\sigma c}$ message is produced by p_c and added in *O*. Here *D*(*null*) is digest of a special *null* request which is used as a place holder during ordering process and is considered a no-op while execution of service operations (as in Paxos algorithm).

p_c prepares and multicasts a $\langle NEW\text{-}VIEW, v+1, VC, O\rangle_{\sigma c}$ message to all processes. p_c updates its log by adding all these pre-prepare messages. It also updates its stable checkpoint and/or current state to *min-s* if *min-s* has a greater order number than the other two. It enters view (*v*+1) and starts accepting orders in the new view.

Each process p_i checks the validity of the new-view message by computing *O* from *VC* in the same way and comparing it with the ones sent by p_c. If found valid, the information in new-view message is then added in the log and stable checkpoint and/or current state updated as mentioned for p_c. Each process then enters view (*v*+1) and starts executing normal protocol by multicasting a prepare message for each pre-prepare included in *O*.

Advanced version of BFT can be found in [CL00, CL02]. These extended versions incorporate proactive recovery to recover replicas periodically. This enables the system to tolerate any number of faults within the lifetime of the system (subject to some restrictions). Other optimizations include usage of symmetric cryptography instead of public key cryptography to increase efficiency and reduction of communication overhead using various techniques. Another interesting extension of BFT has been proposed by Rodrigues et. al. [RKB07] in which the authors have modified the design of BFT to suit to large-scale systems that are composed of many groups of replicas. The modification allows for arbitrary choice of n and f, $n > f$. However, the price to pay comes in the form of more restrictive liveness requirement.

2.4.2 Discussion

We observe that BFT has many features similar to the crash-tolerant protocols discussed in section 2.3 i.e. Paxos and Chandra-Toueg's Algorithms (PA and CTA resp.). For example, it also allows recovery of processes from failures and requires some used timeouts to hold for sufficiently long duration to guarantee liveness. Also, some of the design techniques resemble with one of these two algorithms as mentioned below. Recall the differences in the features of PA and CTA given in subsection 2.3.3.2.

- Proposal generation is by the coordinator like in PA.
- Iterations are executed sequentially as in CTA. That is, a process moves through consecutive views and only changes its view when it suspects the coordinator process.
- A process only responds to messages from the coordinator of its current iteration i.e., view, as in CTA.
- BFT is divided into two parts Normal and View-Change like PA's Part 1 and 2.

We intend to re-use many of these features but in combination with the fail-signal abstraction. Hence, this thesis develops and evaluates the performance of a Byzantine fault-tolerant total order protocol that uses fail-signal processes to achieve the aims mentioned in section 1.3.

2.4.3 FS-Newtop

FS-Newtop [MES03] is a Byzantine fault-tolerant version of the crash-tolerant group communication protocol Newtop [EMS95]. FS-Newtop uses fail-signal processes instead of normal processes to transform arbitrary failures into announced crashes.

Hence, system redundancy is increased in terms of number of processes in such a way that the original protocol design remains the same.

Newtop is based on partitionable system approach and provides total order multicast service. The protocol starts execution with a group comprising of all n processes in the system, $n \geq (2f+1)$ to tolerate f crash faults. Each process uses S-oracle to detect failures. Failure suspicions by S-oracle of a process triggers a mechanism to exclude suspected processes from the group. Hence, execution moves through views with changing membership. As mentioned in subsection 2.1.2.1, false suspicions can cause system to split in logical or virtual partitions even in case of no failures.

FS-Newtop introduces redundancy in each participant process. Each process is replaced by the fail-signal process. Since fail-signal process multicasts fail-signal to announce failure, S-oracle makes use of this feature to detect failures and does not suspect processes based on timeouts. This eliminates virtual partitioning. View membership is still dynamic but only fail-signalled, and therefore failed, processes are eliminated in consecutive views, preserving all correct processes in one group. Hence, use of FS process transforms a partionable protocol into a non-partitionable one.

FS-Newtop demonstrates the effectiveness of the fail-signal approach in circumventing FLP impossibility. It also shows that a total-order protocol that uses this approach no longer needs to rely on timeouts to hold eventually so that the failure suspicions are only correct. Hence, termination is not linked with any liveness requirement as is the case for other deterministic protocols. [MES03] also presents a performance study to measure cost of FS-Newtop over Newtop due to fail-signalling. However, we note that FS-Newtop uses $(4f+2)$ processes to tolerate f Byzantine faults which is much more than the optimal $(3f+1)$ needed by the standard Byzantine fault-tolerant protocols.

2.5 Summary

This chapter studied the two most common approaches found in literature to reach consensus. The first approach is randomization, whereby replica processes are assisted by Random-oracles in choosing decision values. The termination of such algorithms is guaranteed in probabilistic terms. The second approach assumes some restriction on asynchrony of the underlying network. Algorithms in this approach use Suspector-oracles based on timeouts to assist identification of the failed processes in the system.

They can be further divided into two classes. The first class called partitionable system assumes that suspectors rarely make mistakes and estimated timeouts on communication and processing delays are mostly correct. Hence, the suspicion list output by a suspector is used to exclude the suspected processes from the group of processes that execute the protocol. Whereas, the second category called non-partionable system permits frequent mistakes by the suspectors. However, it assumes that the suspicions will eventually be correct i.e., the used timeouts will eventually hold. Hence, suspected processes keep participating in decision process but are not given the most important role of the coordinator.

Furthermore, some canonical protocols belonging to the non-partitionable system class were discussed in detail. These include crash-tolerant protocols named Paxos and Chandra-Toueg's algorithm and Byzantine fault-tolerant protocols named BFT and FS-Newtop. We showed that the way of working of all the protocols share common basic principles. However, we noted that Byzantine fault-tolerant protocols need more redundancy. In particular, BFT comprises of more phases of communication as compare to its crash-tolerant counterparts.

Finally, we introduced fail-signal approach as the third approach to circumvent FLP impossibility. This approach includes provisions in the form of increased redundancy so that failures are announced whenever they occur. This eliminates the need of using timeouts to detect failures. We also referred to FS-Newtop which merges the fail-signal approach with a crash-tolerant partitionable group communication protocol named Newtop. We take the fail-signal concept to an advanced level and design Byzantine fault-tolerant total order protocols that are optimal and efficient.

Chapter 3

Basics on Implementing and Exploiting Fail-Signal Processes

This chapter describes the failure semantics of a fail-signal process and then outlines how such a process can be built through a careful management of redundant computation and output comparison. It is intended to serve two purposes: to revisit earlier work [BES+96] on constructing fail-signal abstraction in the presence of Byzantine faults; secondly and perhaps more importantly, to observe how a fail-signal process, despite internal redundancy, can still give rise to a benign form of two-facing behaviour. This factor needs to be taken into consideration while designing the fail-signal based order protocols.

An implementation of such a process is then presented in detail and we observe that a fail-signal process can be in one of three states: *Working*, *Signalled* and *Failing*. While the first two states encompass consistent behaviour of the fail-signal process, failing state is the one that covers possible arbitrariness in behaviour due to presence of faults. This problematic state is ignored temporarily and based on the reduced two-state assumption, a simple total order protocol is presented to demonstrate how the fail-signal concept can be used to enable distributed nodes to reach identical decisions. The protocol is then examined to highlight problems that arise due to the inclusion of failing state. Such analysis leads to some basic design choices for a more sophisticated protocol presented in Chapter 4. It also provides basis for construction of FS process with weaker assumption and includes a fourth state in the FS process behavioural model.

3.1 Behaviour of a Fail-Signal Process

A fail-signal process, *FS process* for short, executing a computational program P is an abstraction maintained by several processes (redundantly) executing P on distinct fail-independent platforms. These process replicas cooperate with each other so that the computational outputs they collectively present to the environment are guaranteed never

to be incorrect. If a correct output (correct - as per the specification of P) cannot be produced due to faults, then the collective response will be a special signal indicating that inability; this signal is called the *fail-signal*. The collective responses are thus either correct computational outputs or fail-signals and they constitute the responses produced by the FS process.

When the inability to produce a correct output is first detected, not only the fail-signal is transmitted but also steps are initiated to halt the execution of P and to transmit the fail-signal to wherever a computational output is due. If the execution of P is halted swiftly enough, no computational output can be transmitted once the fail-signal has been transmitted; that is, only the transmission of fail-signal can re-occur after its first occurrence.

Unfortunately, the Byzantine nature of the faults means that it is not possible to *guarantee* that the execution of P can be halted swiftly. This has two implications: A fail-signal that is being transmitted cannot be made to indicate the current state of the FS process (e.g., the last collective output that was produced correctly). Secondly, the transition from producing only correct outputs into producing only fail-signals may not be managed to be an instant switch-over but instead can be via a 'messy' intermediary state. Thus, the FS process can be in one of three states, as illustrated in Figure 3.1, and the transition from one state to another can occur at arbitrary instants of time (decided only by the adversary). The FS process behaviour in each of the states however is well-defined and is explained in detail below.

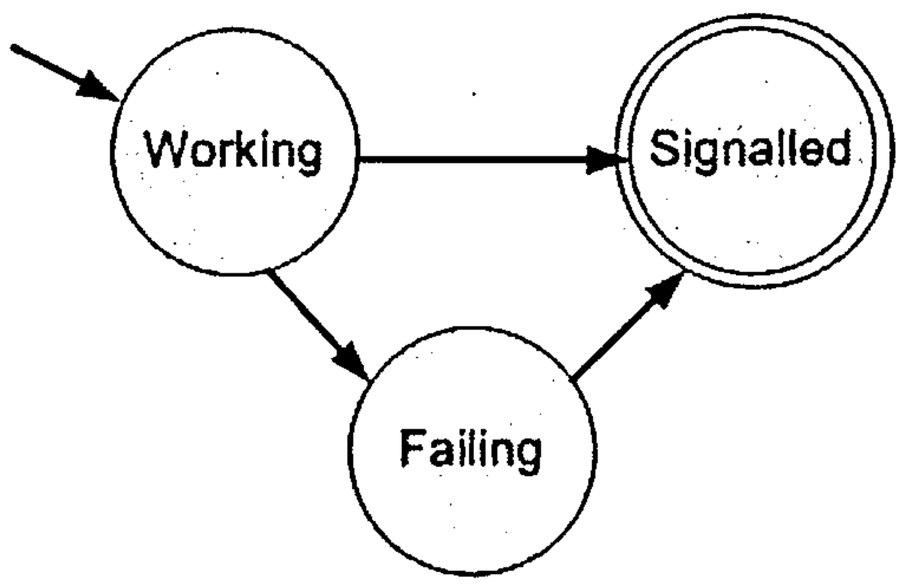


Figure 3.1. Three states of FS process

Working State:

When an FS process is free of internal faults, it is said to be in working state. The FS process operates as per the specification of the program P executed by its constituent process replicas: all outputs expected of it are produced and each output produced is

correct. Thus, an FS process in this state only produces correct outputs and for that reason it is also called an *operative* process. For simplicity, we assume that when an FS process is initialised, it is in the working state, i.e., all constituent process replicas are correct at the time of initialisation.

Signalled State:

When one or more correct constituent processes of an FS process detect that their counterpart have become faulty, the FS process is said to be in signalled state. As for the behaviour in this state, the FS process *halts* executing the program P and only sends a special message, called the *fail-signal*, to any destination to which a response is due; the fail-signal message is uniquely attributable to the source FS process and cannot be undetectably forged by any other process. This state is the terminal state for an FS process (unless explicitly mentioned otherwise – see subsection 3.6.2.3). An FS process in this state is referred to as the *signalled* or *halted* process.

Failing State:

Failing state is defined as the state in which failures of one or more of the constituent processes have *not* been detected by the correct constituent processes. This state can be entered only from the working state and can be left only to enter the signalled state. Unlike the other two states, it is characterised by an absence of consistency in the transmission of responses to destinations. While only correct outputs or only fail-signals are transmitted in the working or the signalled states respectively, these responses can be mixed randomly when the FS process in the failing state. Specifically, a *failing* FS process computes correct outputs (as if it is still in the working state) but behaves in one of the following ways when it comes to what it transmits to a given destination:

B1. An output is transmitted promptly or after some random delay;

B2. Fail-Signal is transmitted promptly or after some random delay; or,

B3. B1 and B2.

When the environment treats the fail-signal as a message to be acknowledged to the source, the failing state is guaranteed to be a transient one: the FS process that receives an acknowledgement for its own fail-signal enters the signalled state within some finite but unknown amount of time.

Remark: *The failing State and the Uncertainties*. Suppose that the specification requires outputs O_1, O_2 and O_3 to be transmitted in that order to a destination q_l. A failing FS process can instead transmit O_2 (promptly), *fail-signal* (instead of O_1 but after

transmitting O_2) and O_3 (thereafter). If q_1 receives the outputs in the transmitted order and decides, on receiving the fail-signal, that the FS process has halted, it will find the FS process behaviour not having halted when it receives O_3.

Suppose now that O_1, O_2 and O_3 are also to be transmitted (i.e., multicast) to another destination q_2. Say, the failing FS process chooses to transmit to q_2: O_1 (promptly), *fail-signal* (instead of O_2) and O_3. If q_2 receives the outputs in the transmitted order it will also find the FS process not having halted even after it received the fail-signal. However, the last output received prior to receiving fail-signal is different: O_1 for q_2 and O_2 for q_1. Thus, from the perspective of destinations q_1 and q_2, i.e., the environment of an FS process, the presence of the failing state introduces two uncertainties:

***Uncertainty I*:** A destination that receives a fail-signal cannot know whether the signalling FS process is in the failing state or in the signalled state; nor does it know how long it should wait for the FS process to make the transition to the signalled state. Therefore, it does not know when to stop accepting responses (such as O_3) that arrive after the fail-signal and start regarding the FS process as halted.

***Uncertainty II*:** If a destination that receives a fail-signal decides not to accept any more responses arriving from the FS process, then, it cannot know which of the multicasts it accepted (or it rejected respectively) are rejected (or accepted respectively) by other multicast destinations.

Summary: Referring to Figure 3.1, the possible states and state transitions of an FS process are summarised as follows. An FS process is initialised into the working state where it only outputs correct responses to all intended destinations. After having been in that state for some (unknown) amount of time, it chooses to transit either to the signalled state or to the failing state. In the signalled state, the fail-signal that is uniquely attributable to the FS process is transmitted to every destination to which an output is due. In the failing state, the FS process may transmit to a destination the correct response or the fail-signal or both in some order. A destination that receives the fail-signal does not know whether the FS process is in the signalled or in the failing state, and knows only that the transition to signalled state will occur eventually once it acknowledges the fail-signal received.

3.2 Requirements and Assumptions for Fail-Signal Processing

A fail-signal process should be implemented by (ϕ+1), $\phi \geq 1$, replica processes that are hosted on distinct, fail-independent nodes connected by a reliable network. This distributed system of (ϕ+1) nodes is referred to as a *Fail-Signal* (*FS*) *node*. All but one of these (ϕ+1) nodes can fail at any time and failures can be authenticated Byzantine which is any form of arbitrary behaviour restricted only by (the standard) cryptographic assumptions described in section 1.4. In particular, individual nodes can be COTS components and are *not* expected to exhibit any form of fail-signal behaviour when they fail. We recall here that a solution without encryption will need an FS node to be constructed by more that (ϕ+1) replicas and may be more efficient. However, for the solution proposed in this thesis, one of the prime objectives is to keep the redundancy to the optimum.

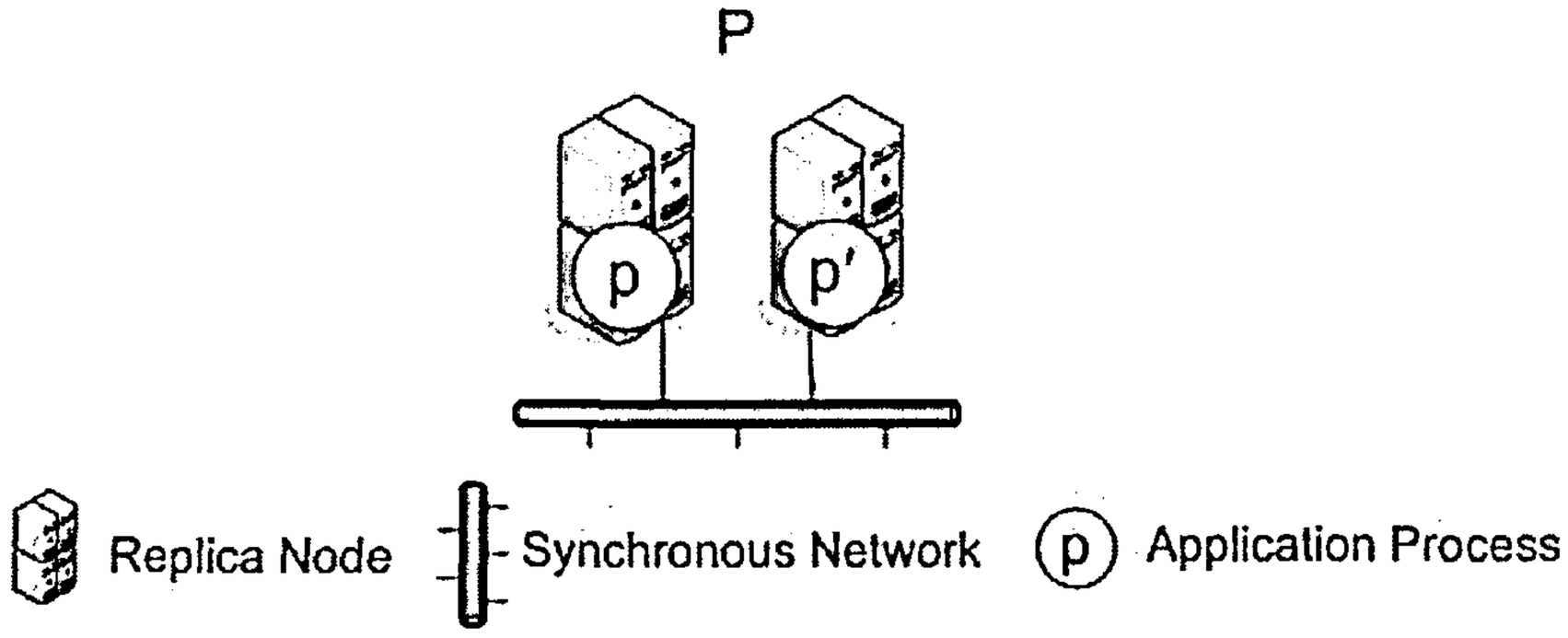


Figure 3.2. A Fail-Signal Node

Throughout this thesis, only FS nodes with $\phi = 1$ will be considered and Figure 3.2 depicts one such FS node. The two nodes are connected by a fast, reliable network to facilitate efficient communication between them. They host the redundant processes p and p' that execute program P. The FS process constructed out of p and p' is denoted simply as P and sometimes explicitly as (p, p').

The mechanisms for constructing an FS process out of spatially replicated processes have been detailed in [BES+96]. The central idea is to have the nodes independently compare the computational outcomes and the relative timeliness of redundant processes. If outcomes are found to be in agreement, then, and only then, they

are presented as 'the computational outputs' from the FS process; if, on the other hand, one node suspects the remote node's process not to be correct or timely, it emits the pre-prepared fail-signal – causing the FS process to transit out of the working state. Thus, the FS process construction basically addresses the following requirements.

1. Comparison must result in agreement when there is no fault in the FS node;
2. It must terminate if a correct node finds its counterpart producing no or untimely or incorrect result; and,
3. Any computational output of the FS process must contain verifiable and non-forgeable evidence indicating that both the nodes have been in agreement over its contents.

The FS process construction makes the following three assumptions to meet the first two requirements and relies on standard public-key cryptographic assumptions to meet the third. Recall that it has already been assumed that all nodes within an FS node are initially correct and that at least one does not fail.

3.2.1 Assumptions

Assumption 0:

The computational program P executed by an FS process is deterministic.

***Remarks*:**

By this assumption, the replicas p and p' must produce the same sequence of results when they process the same sequence of inputs. In this thesis work, p and p' implement an order protocol, i.e., P is a program for a total order protocol. Assumption 0 therefore requires that the order protocol be deterministic and not involve tasks or threads making random choices as in randomised consensus protocols [Ben83, EMR01, Rab83, KS01].

***Assumption 1*:**

The processing load on, and the message traffic between, the nodes within an FS node are carefully engineered to make it feasible to estimate accurate bounds on relative processing delays of these nodes and on (absolute) communication delays between them.

***Remarks*:**

Planned control of processing loads and network traffic and adequate provisioning of resources are common means pursued in real-time systems (e.g., MARS system [Kop97]) to ensure that relative processing delays and absolute communication delays are bounded by known amounts. Assumption 1 regards that similar measures are being

enforced *within* the FS node; it thereby makes it possible to accurately estimate a *differential delay bound* within which a replica process (say, p) that has produced an output can expect to receive the same output form its counter-part (p'), provided that both the nodes are non-faulty and an output is expected as per the specification of P. This accurate estimate allows each replica to correctly assess the timely behaviour of the other during mutual checking. Say, p finds p' not producing the expected output within the estimated delay bound. If p ends the mutual checking with the conclusion that p' has failed to produce a timely output, then, by Assumption 1, that conclusion is correct if the node of p is non-faulty. Neither p nor p' will reach such a conclusion if both nodes are non-faulty. Thus, Assumption 1 helps meet the requirements 1 and 2 mentioned above.

***Assumption 2*:**

At least one of the nodes within an FS node does not fail.

***Remark*:**

This is the reiteration of the foundation assumption made in the design of an FS node that all but one of the (ϕ+1) constituent nodes can fail at any time. Taking into account the malicious faults, this assumption essentially represents a system where either the strength of the adversary is limited or sufficient preventive measures are in place to prevent all nodes from being compromised at the same time.

3.3 Implementation and Operational Details

Figure 3.3 below depicts the scheme advocated in [BES+96] for building an FS process using p and p'. The *sequencer* processes of the FS node receive the messages addressed to p and p' and present them in an identical sequence. The processes receive input messages from only one source: the local *ISIQ* – the *identically sequenced input queue* (see Figure 3.3). Instructions such as '*receive*(m)' and '*receive any*(m)' thus naturally get interpreted for p and p' as: '*receive the first m in the local ISIQ*'. Since P is a deterministic program, p and p' must now generate identical outputs.

The *comparator* process in each node verifies the sameness and timeliness of the remotely computed outputs by comparing the latter with the locally computed ones. If the verification leads to no failure detection, the verified output is sent out as the output of the FS process P after attesting digital signatures to satisfy the third requirement mentioned earlier.

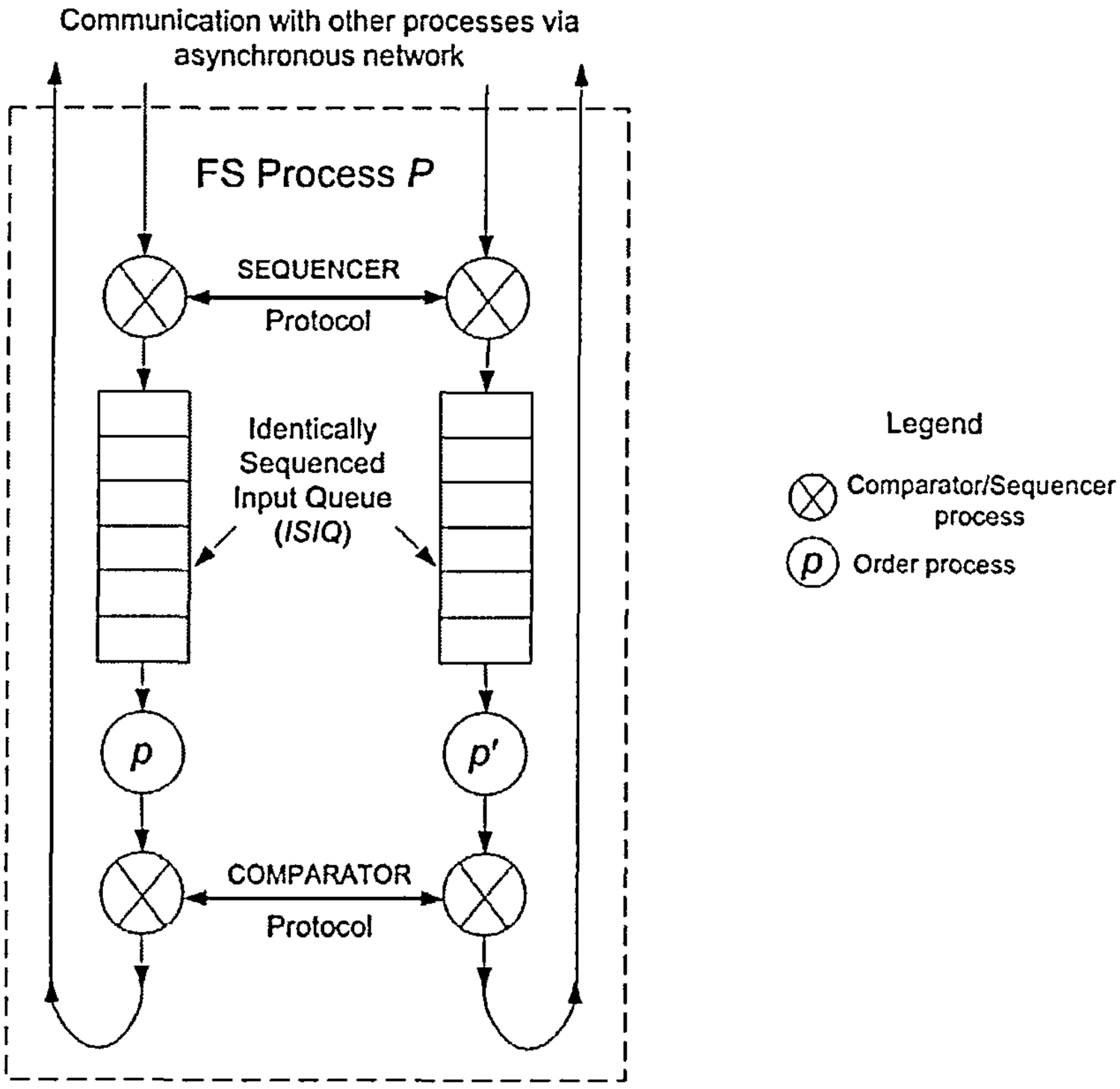


Figure 3.3. Operational Architecture of FS Process

In summary, the *sequencer* and the *comparator* process pairs build the FS process *P* out of *p* and *p'*. Their design must cope with the variety of 'fault' situations that may arise within an FS node and these situations can be categorised into three types.

No fault situation: By assumption, this is the initial situation within the FS node and the design of *sequencer* and *comparator* should ensure that the FS process remains in the *working* state so long as this situation prevails.

Faulty node within the FS node exhibits its failure directly to the correct node: The failure must be detected and the FS process taken to the *signalled* state. If failures of the faulty node are exhibited *only* to the correct node, then the transition to the *signalled* state will be directly from the *working* state; otherwise, it can be via the *failing* state.

Faulty node does not exhibit its failures directly to the correct node: Once the reports of failure are verified to be true, the FS process is taken to the *signalled* state (possibly from the *failing* state).

We will briefly describe the workings of the *sequencer* and the *comparator* processes by considering three cases: FS process (i) in the *working* state, (ii) transiting

to *signalled* state due to detection of directly exhibiting failures and (iii) transiting to *signalled* state from *failing* state.

3.3.1 Working State

The sequencer protocol advocated in [BES+96] is one of simple Leader-Follower scheme. Say, p is assigned the role of the leader and p' the follower. This means that the sequencer process in the host node of p (the leader sequencer) dictates the sequence in which the inputs need to be entered into *ISIQ* and thus processed. The sequencer process of p' (the follower sequencer) simply enforces the sequence dictated to it. In addition, it forwards to the leader sequencer any input destined for p' which has not been sequenced yet and observes if the leader sequences every forwarded input before some generously estimated timeout expires. The leader sequences every new input that it encounters and lets the follower know the sequence.

The *comparator* protocol also takes up the leader-follower approach and the steps are depicted in Fig 3.4. For every output produced by p, the local comparator process (the leader comparator) signs the output and forwards the signed message to its counterpart (the follower) in the host node of p' (shown as the dotted arrow in Fig 3.4). If the follower comparator finds that an identical output is locally produced, then it endorses the received 1-signed message by over-signing it and sending the doubly-signed result to the intended destination(s) and also to the leader comparator. When the latter receives an authentic, doubly-signed message containing its original 1-signed message, it forwards the received to the intended destination(s) and also to the follower comparator.

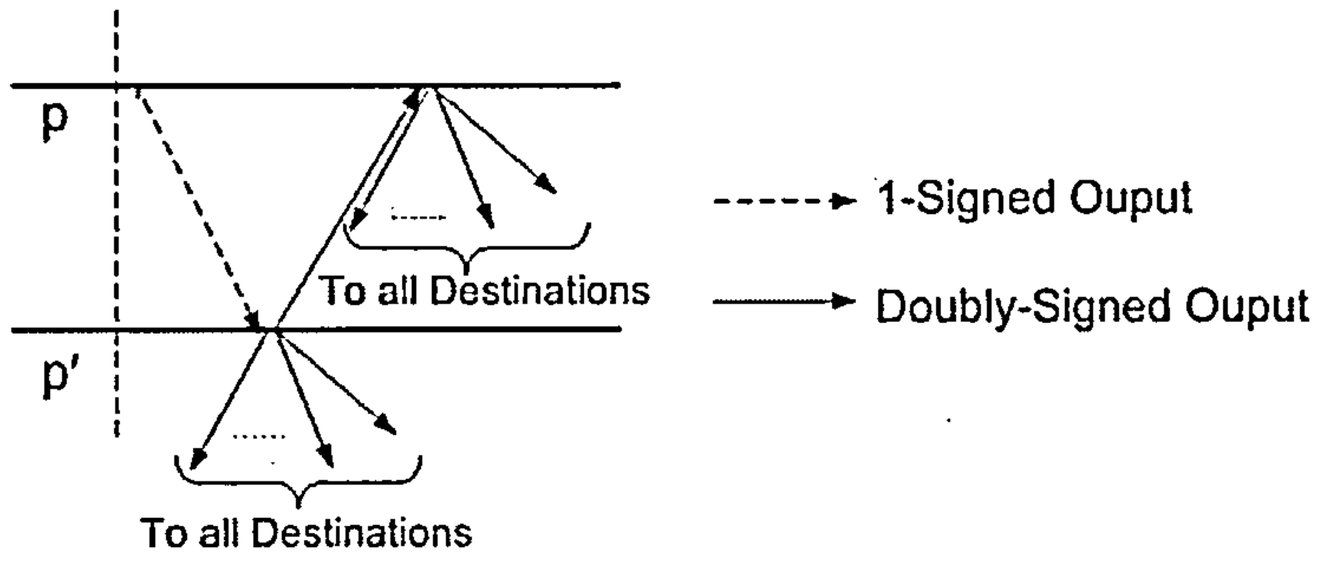


Figure 3.4. Output Checking and Endorsement

Note that the presence of two signatures in the output message of the FS node indicates that the message contents have been endorsed by both nodes; since both nodes cannot be faulty, the contents of a doubly-signed message must be correct. Note also

that each node independently transmits a double-signed output – leading to redundant output transmissions capable of tolerating transmission failure by one node.

3.3.1.1 Use of Comparator Timers

The comparator processes also monitor each other for timely behaviour using liberally estimated timeouts. Three types of timeouts are used: (i) for every output produced by p', the follower comparator expects to receive from the leader a 1-signed message containing the same output before the timer expires; (ii) for every 1-signed message received from the leader, there is a matching output produced by p' before the timer expires; and (iii), for every 1-signed output sent to the follower, the leader comparator expects to have received a doubly-signed equivalent from the follower before the timer expires. Since timeout durations are estimated accurately (Assumption 1), timeouts do not expire pre-maturely and their expiry cannot be a false-positive.

3.3.2 Direct failure detection

At the time of initialisation, each node in the FS node is supplied with a fail-signal message signed by the other node. Whenever the sequencer (or the comparator) process of a node first detects a failure of the other node, the local comparator (or the local sequencer) process is informed of this detection; the 1-signed fail-signal is over-signed with the local node's signature and the resulting double-signed fail-signal is used to indicate the transition from the working state. This transition is accomplished by executing the *halting procedure*: the sequencer process stops sequencing input messages (thus letting *ISIQ* become empty eventually) and sends the 2-signed fail-signal to the source of every input message received; the comparator process stops comparing and sends the 2-signed fail-signal to destination(s) for whom an output has been produced locally. Two important remarks are now in order.

It is possible that the comparator/sequencer of a correct node receives from its counter-part a doubly-signed fail-signal containing the local node's signature. This can occur when the comparator/sequencer of the faulty node detects a failure and initiates the halting procedure. When this happens, the halting procedure is initiated locally.

Note that each node possesses the ability to transmit the 2-signed fail-signal without the cooperation of its counter-part. Because of this ability, correct node can generate and transmit the 2-signed fail-signal even if its faulty counter-part refuses to cooperate (simply because of a crash). But possessing this ability has two implications:

1-signed fail-signal needs to have been formed at the time of system initialisation and cannot therefore contain any information about the process state at the time of fail-signalling (e.g., the first input for which the correct output could not be computed); so, the fail-signal only indicates that a failure has been observed within the FS node. Secondly, a maliciously faulty node can transmit a 2-signed fail-signal unbeknown to its correct counter part and the impact of such behaviour is discussed in Section 3.3.3.

3.3.2.1 Scenarios leading to failure detection

Following are the scenarios in which a sequencer process concludes a failure of its counterpart.

(i) Timeout occurs (at the follower sequencer) before the sequence for a forwarded input can be known;

(ii) A forwarded/sequenced input is found to be syntactically incorrect; and,

(iii) The fail-signal received is found to contain the local node's signature.

Following are the scenarios in which a comparator process concludes a failure.

(i) Timeout occurs at the follower while a 1-signed output that matches a locally computed output has not been received from the leader; timeout occurs at the follower while an output that matches a 1-signed output received from the leader has not been computed locally; timeout occurs at the leader while a 2-signed output expected from the follower has not been received (see also subsection 3.3.1.1);

(ii) What is received is found to be syntactically incorrect;

(iii) The fail-signal received is found to contain the local node's signature.

3.3.3 From Failing to Signalled State

With no loss of generality, let us suppose that the host node of p' (the follower node) is faulty. Consider the scenario that the follower node does not fail towards the leader, and fails only towards the environment by omitting to send some of the (double-signed) output messages or sending the output messages in some corrupted form. Since output messages are also emitted by the leader node as well (see Fig 3.4), the FS process appears to be in the working state if destinations choose to discard any corrupt output messages they possibly receive. However, the following malicious failure modes will cause the FS process to transit to the failing state.

3.3.3.1 Malicious Behaviours Leading to Failing State

Suppose that the faulty follower node deliberately avoids sending the 2-signed output message(s) to the leader. Before its timeout expires (see case 3 in subsection 3.3.1.1), the leader could have entrusted β 1-signed messages, $\beta \geq 1$, with the faulty follower. (In any comparator protocol, at least one comparator process must entrust the other with at least one 1-signed output message; otherwise, output comparison cannot make progress in the absence of faults; so, $\beta \geq 1$.) The faulty follower, being solely in possession of β double-signed, valid messages, can choose not to send some of them or send them only to some subsets of destinations possibly after some arbitrary delays. The destinations that get sent a double-signed message will experience the failing state behaviour of B3 (see Subsection 3.1) for that output, while those that do not get sent any double-signed message experience B2 by receiving fail-signal from the timed-out correct node.

Another form of malicious behaviour that leads to failing state is as follows. Suppose that the faulty follower prepares the 2-signed fail-signal without detecting any failure and transmits it selectively to some destinations, while maintaining correct behaviour to its counterpart. The net effect is that all destinations get sent (by the correct node) double-signed, output messages, while some (chosen by the faulty node) additionally get sent fail-signal messages and thereby experience the failing state behaviour of B3. When a destination echoes or acknowledges the fail-signal to both the nodes of the FS process, the correct leader node will realise that the fail-signal has already been sent out and initiate the halting procedure. Note that when the correct node executes the halting procedure, the faulty follower may be in sole possession of β double-signed, valid messages, as explained earlier. So, it is only safe to state that after some unknown but finite amount of time, the FS process will output only fail-signal and thus enter the signalled state.

3.4 Protocol-0: The Basic Protocol

This section develops an order protocol by making a few simplifying assumptions on the nature and the number of FS processes employed. Specifically, we assume that no FS process ever enters the failing state. That is, when a constituent process fails, it is directly detected by the counterpart (failure always enables direct failure detection as explained in subsection 3.3.2). Moreover, it is assumed that at least one FS process

never leaves the working state. The protocol is therefore aptly referred to as the *basic protocol* or simply *Protocol-0* in the rest of the thesis. The correctness arguments and the subsequent observations lay the foundations for developing more sophisticated protocols whose performance will be studied in detail. Protocol-0 is developed in the following system context.

3.4.1 System Context

Protocol-0 retains the basic replicated service system structure sketched in Chapter 1: a set of $(2f+1)$, $f \geq 1$, service processes running on asynchronously distributed nodes. To assist replicas in executing client requests in identical order, each replica node N_i is additionally installed with an order process, p_i, executing Protocol-0. To build FS processes, $(f+1)$ of these nodes will be paired with shadow nodes, as shown in Figure 3.5. Each shadow node N'_i, $1 \leq i \leq f+1$, hosts the shadow order process, p'_i, which also executes Protocol-0. The process pair p_i and p'_i implement the FS process P_i with the former acting as the leader and the latter as the shadow.

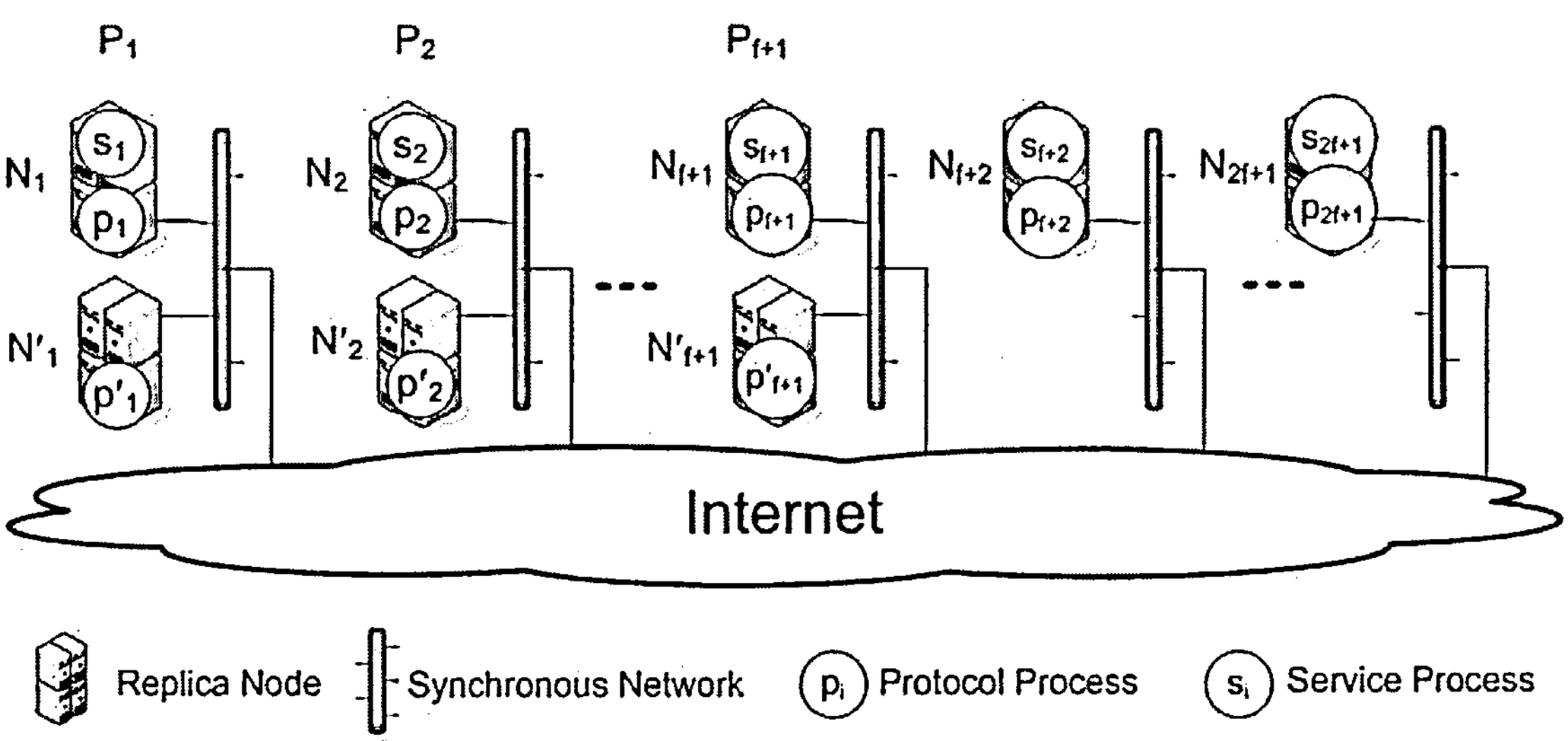


Figure 3.5. System Architecture

We list below the notations for various sets of order processes which are of interest to protocol description. The notations will follow this convention: Greek letters are used for denoting process sets, upper case Roman letters for items related to fail-signal processes (e.g., P_1), lower case Roman letters for those related to normal processes (e.g., p_1), and "′" for distinguishing those related to shadows (e.g., p'_1).

Π = Set of all FS processes = $\{P_1, P_2, \ldots, P_{f+1}\}$;

FS_i = 2-signed fail-signal from P_i;

π = Set of all order processes co-located with service processes = $\{p_1, p_2, \ldots, p_{2f+1}\}$;

π' = Set of all shadow order processes (not co-located with service processes)

= $\{p'_1, p'_2, \ldots, p'_{f+1}\}$.

The set ($\pi \cup \pi'$) of $3f+2$ order processes form the middleware for supporting Byzantine fault-tolerant service replication. Client requests are first received by this middleware layer i.e. requests are sent to all $3f+2$ order processes. These distributed order processes coordinate with each other to irreversibly assign a unique sequence number, called the *order number* and denoted as o, to each given request r. Following the terminology of BFT, we term this act of irreversibly assigning o to request r as *committing* o (to r).

The order processes in π forward the requests to local service processes for processing as per the order numbers committed. It is worth mentioning here that the total number of nodes used by Protocol-0 is $3f+2$ which is one more than that needed in BFT (i.e., $3f+1$).

3.4.2 Protocol Design

Since the protocol design is to exploit the aspects of FS processes, two design choices are readily made. Since the protocol has to be deterministic (see Assumption 0 in Subsection 3.2.1), we choose it to be coordinator based, which is the most common class of deterministic protocols. Secondly, since an FS process never generates an incorrect output (i.e., an incorrect order), it is best qualified to carry out the coordinator role: the order number o which it chooses for a given client request r can be simply trusted by every order process p in $\pi \cup \pi'$.

When the FS process acting as the coordinator transmits a fail-signal, one of the operative FS processes can take over the role; since there are ($f+1$) FS processes and only at most f nodes can fail, there will always be at least one operative FS process in the system. The protocol design thus basically needs to address two questions: which FS process should take over the coordinator role when the coordinator fail-signals and how the taking-over should be accomplished.

3.4.2.1 Who Becomes the New Coordinator?

This question is answered simply by publicly ranking all ($f+1$) FS processes and defining thereby the sequence in which they are to take up the coordinator role. Let this

ranking be P_1, P_2, …, P_{f+1} and be known to all $p \in \pi \cup \pi'$. Thus, P_1 becomes the first coordinator when the protocol is initialised.

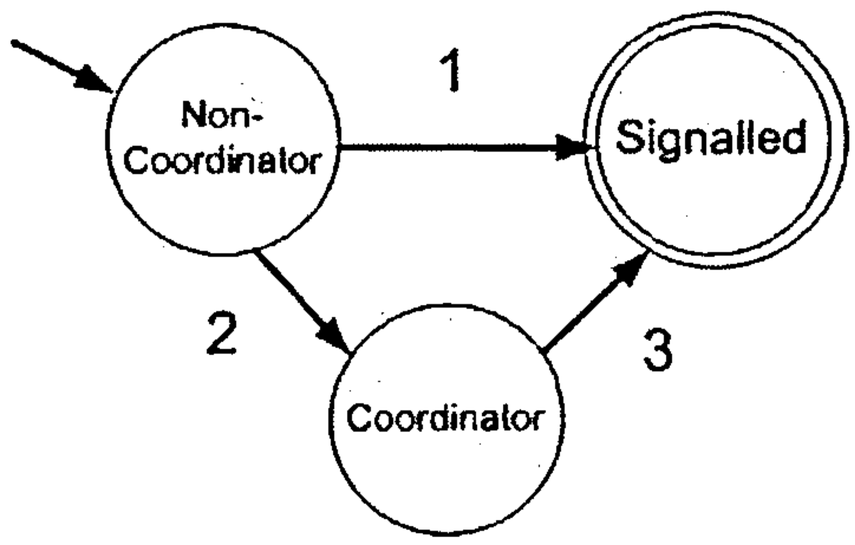


Figure 3.6. Possible role transitions during protocol execution

Figure 3.6 shows how the role of a given FS process can change during protocol execution. An operative FS process that is acting or not acting as the coordinator is in the coordinator or non-coordinator state, respectively. When it emits a fail-signal, it enters the terminal, signalled state. (The intermediary, failing state is never entered – see Figure 3.1.) All FS processes are assumed to be initially in the non-coordinator state at the start of the protocol and P_1 (the first ranked FS process) instantly makes the transition (shown as 2 in Figure 3.6) to coordinator state.

Note that at most one FS process can be acting as the coordinator at any given moment during protocol execution. Let P_c be the current coordinator. When P_c detects that it can no longer transmit correct ordering of outputs, it enters the terminal state of signalled as shown by transition 3 in Figure 3.6. When P_i, $i \neq c$, learns of fail-signalling by P_c, it decides to become the coordinator (decides to make transition 2) if $i = c+1$ or if it had received a fail-signal from each P_k, $c+1 \leq k \leq i-1$. If P_i, $i > c+1$ becomes the next coordinator after P_c has fail-signalled, then each P_k, $c+1 \leq k \leq i-1$ must have already made transition 1.

3.4.2.2 How to Become the New Coordinator?

The taking-over of the coordinator role by P_i from P_c is accomplished through P_i executing the *install* procedure and must satisfy the following condition: P_i must be seen by every correct process $p \in \pi \cup \pi'$ to be maintaining continuity in assigning order numbers to new client requests. More precisely, let $max\_committed_p$ be the largest order number committed by *any* correct process $p \in \pi \cup \pi'$ before p decides that P_c is no longer the coordinator. The starting order number *start_o* used by P_i for ordering new client requests must be:

$$start\_o = 1 + \text{maximum}\{max\_committed_p \mid \forall \text{ correct } p \in \pi \cup \pi'\}.$$

Obviously, P_i cannot compute *start_o* by directly contacting each p of $\pi \cup \pi'$ for $max\_committed_p$, since it cannot know which p is correct and which is not. So, the protocol imposes the following two rules for committing order numbers to requests.

- A correct p commits o only after it knows that every operative FS process $P \in \Pi$ has acknowledged the coordinator's proposal of o for r.
- Each operative non-coordinator FS process P_j multicasts its acknowledgement to the proposal of o for r, so long as (i) it knows that the proposing coordinator has not fail-signalled and (ii) it has acknowledged the proposal for (o -1).

The *install* procedure now becomes like a 2-phase commit protocol:

- P_i transmits an *INVITE* message to all FS processes; the message contains (i) the fail-signal of each P_k, $1 \leq k \leq i$-1, as evidence for P_i's eligibility to install itself as the coordinator and (ii) A_i which is the largest order number P_i has acknowledged so far.
- An operative FS process P_j, on receiving *INVITE* from P_i, replies with its own A_j - the largest order number P_j has acknowledged so far. The reply message is called the *STATUS* message which also contains the list of acknowledged proposals for all o in $[A_i + 1, A_j]$ if $A_j > A_i$.
- P_i waits to receive the *STATUS* message from each operative FS process. This waiting must terminate since any FS process which P_i regards to be operative must transmit either the *STATUS* message or a fail-signal. P_i computes *start_o*:

  $start\_o = 1 +$ maximum of $\{A_j \mid \forall P_j \in \Pi - Signalled_i\}$, where

  $Signalled_i$ is the set of FS processes from which P_i has received a fail-signal.

Remark:

It may appear that *start_o* can be computed as 1 + minimum of $\{A_j \mid \forall P_j \in \Pi - Signalled_i\}$: there is at least one FS process, say P_{mn}, which sent *STATUS* message to P_i with $A_{mn} = \text{minimum}\{A_j \mid \forall P_j \in \Pi - Signalled_i\}$; P_{mn} has not so far acknowledged any proposal with order number larger than A_{mn} and will not acknowledge any proposal from P_c which is now known to have fail-signalled; so, it would appear that no correct $p \in \pi \cup \pi'$ could have committed an order number larger than A_{mn} while P_i computes *start_o*. Such a conclusion can be incorrect as the following scenario explains.

Let P_{mn} be the only FS process in (Π - $Signalled_i$) to have sent A_{mn}, i.e., all other P_j had sent $A_j > A_{mn}$. Suppose also that P_{mn}, soon after sending its *STATUS* to P_i,

enters the signalled state and sends fail-signal to all processes in ($\pi \cup \pi'$). Let the fail-signal from P_{mn} take some arbitrarily long time to reach all processes except for one correct p which receives the fail-signal in zero time. Finally, assume that the fail-signal from P_c and any message that indicates fail-signalling of P_c (e.g., the *INVITE* from P_i) take some arbitrarily long time to reach p. All this means is that for a period of time only p knows the fail-signalling of P_{mn} and it also remains unaware of the coordinator change from P_c to P_i; during this period, p can commit proposals without awaiting acknowledgements from P_{mn} and commit up to minimum of $\{A_j \mid \forall P_j \in \Pi - Signalled_i - \{P_{mn}\}\}$ which is larger than A_{mn}. That is, $o\_max_p > A_{mn}$. On the other hand, maximum$\{A_j \mid \forall P_j \in \Pi - Signalled_i\} \geq$ maximum$\{A_j \mid \forall P_j \in \Pi - Signalled_i - \{P_k\}\}$ for any $P_k \in (\Pi - Signalled_i)$. That is, *start_o* computed as presented above is guaranteed to be larger than o_max_p for any correct p.

The scenario constructed above comes about primarily due to two reasons which are explored in detail later (see subsection 3.6.1) and are briefly mentioned below for now. The nature of communication outside FS nodes is asynchronous and this means that a multicast message, even when transmitted correctly, may take unknown amount of time to reach different individual destinations; in the extreme, zero to some and some (finitely) large periods to others – e.g., the fail-signal from P_{mn} reaching p in zero time and others after some arbitrarily long time. Secondly, a fail-signal does not, and cannot be made to, indicate the process state prior to fail-signalling; for example, the fail-signal from P_{mn} cannot indicate that the transmission of *STATUS* to P_i has just preceded its transmission. Had it been the case, p would have become aware of the on-going coordinator change and ceased committing proposals from P_c and the scenario would not have been possible.

3.4.3 Data Structures

The types of messages used reflect the protocol structure: coordinator proposes an order number o for a request r, and a proposal needs to be acknowledged by all operative FS processes before it can be committed by any order process in ($\pi \cup \pi'$). Disseminating proposals, acknowledgements and commits are done using messages of types *ORDER*, *ACK* and *COMMIT* respectively. The protocol structure also implies that order processes that do not construct FS processes, *viz.* $p_{f+2}, p_{f+3}, \ldots, p_{2f+1}$, play no significant part in ordering requests but simply learn the committed order numbers.

3.4.3.1 Messages Used

***ORDER*(*o*):** This is the short-hand notation for an *Order* message by which coordinator P_c proposes order number o for request r. (Where needed, we would explicitly indicate r in the form *ORDER*(o, r)). The message has four fields and its structure can be expressed as: $\langle ORDER, c, o, D(r)\rangle$, where $D(r)$ is the digest of r generated (by P_c) using a one-way, collision-resistant hash function. (See assumptions in section 1.4)

***ACK*$_i$(*o*):** This is a short form for an *Ack* message by which the FS process P_i acknowledges *ORDER*(o) which contains o and $D(r)$ of *ORDER*(o). The full message structure is: $\langle ACK, o, D(r), i\rangle$.

***COMMIT*$_i$(*o*):** This is a short form for a *Commit* message that indicates that P_i has committed order number o to some request r. (Where needed, we would explicitly indicate r in the form *COMMIT*(o, r)). The message structure is: $\langle COMMIT, o, D(r), i\rangle$. Any process that receives $COMMIT_i(o) = \langle COMMIT, o, Dig, i\rangle$ learns that o has been irreversibly assigned universally to the request r that satisfies $D(r) = Dig$.

The protocol additionally deals with *fail-signal*, *INVITE* and *STATUS* messages. It restricts that all messages be generated only by an FS process; therefore, they are all double-signed and may be redundantly transmitted by the replicated processes implementing the corresponding FS process. We assume that the communication subsystem delivers only authentic, double-signed messages from FS processes as 'received' messages to protocol execution and also filters out duplicates.

3.4.3.2 Variables and Functions Used

Processes p_i and p'_i, $1 \leq i \leq (f+1)$, maintain the following variables.

c holds the rank of the co-ordinator (initially 1);

A is the largest order number for which an *ACK* has been sent (initially 0);

Signalled$_i$: Set of FS processes known by p_i to have fail-signalled (initially empty);

Order_Pool: set of *ORDER* messages sent/received (initially empty);

Ack_Pool: set of *ACK* messages sent/received (initially empty).

Processes p_i and p'_i, $1 \leq i \leq (f+1)$, also have following constant sets.

Π = Set of all FS processes = $\{P_1, P_2, \ldots, P_{f+1}\}$;

π = $\{p_1, p_2, \ldots, p_{2f+1}\}$;

π' = $\{p'_1, p'_2, \ldots, p'_{f+1}\}$.

The predicate *committable*(o) is true for a given $o > 0$ if (i) $ORDER(o, r) \in$ *Order_Pool* and (ii) $\langle ACK, o, D(r), j\rangle \in$ *Ack_Pool* for every $P_j \in (\Pi$ - *Signalled*$_i)$.

The function *eligible*() returns the smallest j such that $P_j \in (\Pi$ - *Signalled*$_i)$.

When $c \neq i$, P_i is a non-coordinator FS process; when $c \neq i$ and $i =$ *eligible*(), P_i becomes eligible to act as the coordinator and therefore p_i and p'_i execute the *install* procedure. The protocol is presented next.

3.4.4 Algorithm

The protocol is executed by all processes in $(\pi \cup \pi')$. The processes p_i and p'_i, $1 \leq i \leq (f+1)$, which implement FS process P_i play a very significant role compared to those that do not construct FS processes, *viz.* p_{f+2}, p_{f+3}, ..., p_{2f+1}. Therefore, protocol description will focus only on the activities of the former; the activities performed by the latter are specifically identified.

The protocol for p_i and p'_i, $1 \leq i \leq (f+1)$, consists of three parts. The first part is the main programme and is executed continuously irrespective of whether P_i is the coordinator ($c = i$) or not ($c \neq i$). The other two parts are executed only when c and *eligible*() satisfy certain conditions. The second part is executed when $c \neq i$ and $i =$ *eligible*() and is initiated from the first part. The execution completes the *install* procedure (i.e., completes the transition 2 in Fig 3.6) and uses the third part to make p_i and p'_i start acting as the coordinator FS process P_i.

In describing the protocol, when we say that p_i or p'_i 'receives' or 'transmits' a message, we mean that the message is being picked up from the head of the ISIQ maintained by the local sequencer process or being transmitted through the local comparator process, respectively (see figure 3.3). To simplify description, we would allow a process to send a message to itself, in particular while the message is a multicast. This style is in common with the description of consensus protocols in the literature. Note that when a message is transmitted to itself, it is treated like any other message – i.e., transmitted through the local comparator process and received through the local sequencer process.

The main part consists of four tasks each of which is activated whenever a message of distinct type is received. The message types are: *ORDER*, *ACK*, *Fail-Signal*, and *INVITE*. The activities of these concurrent tasks are illustrated in Fig 3.7 and are then described in detail.

```
main()
while (P_i ∉ Signalled_i) do →
{                                                    /Task 1
(ORDER(o) received from P_c and o > A) →
{
1.1    enter received ORDER(o) into Order_Pool;
1.2    while (ORDER(A+1)∈ Order_Pool) do →
       {
1.3          multicast ACK(A+1) to all P ∈ Π;
1.4          enter ACK(A+1)into Ack_Pool;
1.5          A = A+1;
       }
}
||                                                   /Task 2
(ACK(o) received) →
{
2.1    enter received ACK(o) into Ack_Pool;
2.2    while (∃ o : committable(o) = true) do →
       {
2.3          multicast COMMIT(o) to all p ∈ π∪π';
2.4          Ack_Pool = Ack_Pool\{ACK(o)};
       }
}
||                                                   /Task 3
(FS_j received and P_j ∉ Signalled_i) →
{
3.1    enter P_j into Signalled_i;
3.2    multicast the received FS_j to all P ∈ Π;
3.3    if (c = j) →
3.4    {      c = -1; Order_Pool = Order_Pool\{ORDER(o>A)};
3.5    if (i = eligible()) →
3.6          multicast <INVITE, A, Signalled_i> to all P∈Π;}
}
||                                                   /Task 4
(<INVITE, A_j, Signalled_j> received from P_j) →
{
4.1    Signalled_i = Signalled_i ∪ Signalled_j;
4.2    if (j=eligible()) →
4.3    {      unicast <STATUS, i, A, {ORDER(A_j+1),..., ORDER(A)}> to P_j;
4.4          if (i = j) → install() else c = j;
       }
}
} // end while (P_i ∉ Signalled_i)
while (true) do →
{                                        / Task 5, performed by all p ∈ π∪π'
(COMMIT(o,r) received) →
{
   record o for r;
   if (not shadow) →
       {Deliver r to local service process as per increasing o;}
}
} // end while (true)
} // end main()
```

Figure 3.7. Tasks executed by an operative FS process

3.4.4.1 Description – Main Part

Task 1 is executed whenever a new *ORDER*(*o*) is received from the current coordinator and a received *ORDER*(*o*) is regarded to be new if no *ACK*(*o*) has been transmitted. The task enters the received into the *Order_Pool* and ensures that each *Order_Pool* entry is acknowledged in-sequence, i.e., as per its order number *o*. Note that an *ACK*(*o*) is sent to all p_i and p'_i, $1 \leq i \leq (f+1)$ which includes the sending process as well.

Task 2 deals with *ACK* messages received: it enters them into the *Ack_Pool* and explores if an uncommitted *ORDER*(*o*) can now be committed due to the new entry. If *committable*(*o*) is true, then *COMMIT*(*o*) is multicast to all $p \in \pi \cup \pi'$ and all *ACK* messages related to *ORDER*(*o*) are purged from the *Ack_Pool*. The latter is indicated using the operator '\'.

Task 3 updates the set $Signalled_i$ in response to receiving fail-signals from FS processes not in that set; it also multicasts the received fail-signal to all FS processes. If the signalling process turns out to be the coordinator (line 3.3), three actions are carried out (in lines 3.4-3.6): *c* is set to -1 to ensure that Task 1 no longer accepts any *Order* message from the signalled coordinator; the *Order_pool* is purged off any unacknowledged *ORDER* messages; finally, an *INVITE* message is multicast if P_i is found eligible to become the next coordinator. Note that the *INVITE* message contains the set $Signalled_i$ – informing any recipient of the reason for coordinator change in case the fail-signal from the coordinator has not yet been received.

Task 4 deals with any *INVITE* message received. Recall that multicast messages may be arbitrarily delayed to different destinations. This has two implications for Task 4: (i) p_i or p'_i may not yet be aware of the fail-signalling by the current coordinator and (ii) the received *INVITE* message may be so delayed that its sender, say, P_j has fail-signalled while the *INVITE* is in transit and p_i or p'_i is already in possession of P_j's fail-signal. So, processing of *INVITE* message from P_j begins with verifying the eligibility of P_j based on local $Signalled_i$ set and also on $Signalled_j$ sent by P_j (lines 4.1 and 4.2). If P_j is found eligible, a *STATUS* message is sent containing all *ORDER* messages which the local process has acknowledged but the same has not been done by P_j. (The list would be empty if $A \leq A_j$.) Of course, the sender of *INVITE* message can be the local process itself; in that case, *install*() procedure is invoked; otherwise, *c* is set to *j* in expectation of P_j installing itself as the coordinator. Contrary to this expectation, if P_j fail-signals,

then Task 3 will re-set c to -1 until another eligible FS process comes forward. Thus, 'telescopic' failures during coordinator change are handled.

Task 5 executed by all order processes in the system including the unpaired ones. On receiving a *COMMIT*(o, r) message, each process records o as committed for r. Furthermore, an order process p_i that is co-located with service process s_i, delivers r to s_i once all requests with smaller order numbers are delivered. Here we assume that each request is delivered only once i.e., duplicate *COMMIT*(o, r) messages are discarded.

3.4.4.2 Description – Install Procedure

Processes p_i and p'_i, for some i, $1 \leq i \leq (f+1)$, which find P_i eligible to take the coordinator role execute this procedure after having multicast an *INVITE* message. The pseudo-code is presented in Figure 3.8 and the aim is to compute *start_o* (defined in subsection 3.4.2.2) and a list of *ORDER* messages that are not a common knowledge among the operative FS processes called *TransferHistory*. Therefore the new coordinator should re-issue these *ORDER* messages before starting to propose order numbers on its own right. This computation begins after *STATUS* message has been received from all $P_j \in (\Pi - Signalled_i)$, as shown in line I1 in Figure 3.8. Recall that the *STATUS* message from a given P_j is unicast in response to *INVITE* multicast by p_i and p'_i and contains A_j and a list containing all *ORDER*(o), $A_i+1 \leq o \leq A_j$.

```
install()
{
I1    wait until <STATUS, j, Aj, {*}> received from all Pj ∈ (Π -
      Signalled);
I2    Amn = minimum{Aj | ∀ Pj ∈ (Π - Signalled) };
I3    Amx = maximum{Aj | ∀ Pj ∈ (Π - Signalled) };
I4    TransferHistory = {ORDER(Amn+1),..., ORDER(Amx)};
I5    coordinator(TransferHistory, Amx+1);
}
```

Figure 3.8. Install Procedure

TransferHistory is computed in line I4 as the list of all *ORDER*(o) in the range $[A_{mn}+1, A_{mx}]$ where A_{mn} and A_{mx} are respectively the minimum and maximum values of A reported in the *STATUS* messages received (lines I2 and I3). It consists of an *ORDER* message that some $P_j \in (\Pi - Signalled_i)$ does not have in its *Order_Pool* and some other $P_{j'} \in (\Pi - Signalled_i)$ has multicast an *ACK* message for. The computed *TransferHistory* and $(A_{mx}+1)$ are passed as parameters to initialise a parallel task *coordinator*() which enables P_i to act as the coordinator.

We note that if two or more *STATUS* messages contain an *ORDER*(o) for a given o, then all these *ORDER*(o) will have the same $D(r)$ and possibly different values of c. (This is shown as Lemma 1 later.) When *ORDER*(o) messages for a given o have different values of c, any one of them can be chosen to construct the *TransferHistory*.

3.4.4.3 Description – Coordinator Task

This task, as its code in Figure 3.9 depicts, has two parts: dealing with *TransferHistory* and then proposing new order numbers for requests.

```
coordinator(List TransferHistory, int start_o)
{
C1      if (TransferHistory not empty) →
C2      {       for every <ORDER, *, o, D(r)> in TransferHistory do →
C3              { form <ORDER, i, o, D(r)> and multicast to all P ∈ Π;}
        }
C4      int o = start_o;
C5      while (c = i) do →
C6      {       receive(r); // pick-up unordered r;
C7              form <ORDER, i, o, D(r)> and multicast to all P ∈ Π;
C8              o = o + 1;
        }
}
```

Figure 3.9. Coordinator Task

3.5 Correctness Arguments

The protocol is correct if it satisfies liveness and safety requirements stated below.

Liveness: Every client request is eventually ordered.

More precisely, for every request r, there is a *COMMIT*(o, r) transmitted by a correct order process for some o.

Safety: A client request that is ordered is uniquely ordered: if *COMMIT*(o, r) is transmitted, then neither *COMMIT*(o, r'), $r \neq r'$, nor *COMMIT*(o', r), $o \neq o'$, will be transmitted.

Two assumptions are central to the correctness arguments. Since only a maximum f processes can fail and both constituent processes of an FS process cannot fail, there is at least one FS process which never fail-signals during any execution of the protocol. We term these FS processes as *perfect* FS processes. Secondly, no FS process ever enters the failing state when exiting the working state; any FS process that transmits a fail-signal is already in the terminal, halted state.

Lemma 1: The *STATUS* messages received by a correct p_i that is executing *install*() cannot contain *ORDER* messages that have the same o but different $D(r)$.

Proof: Assume to the contrary that p_i receives a *STATUS* message from P_j containing $\langle ORDER, c, o, D(r)\rangle$ and another from $P_{j'}$ containing $\langle ORDER, c', o, D(r')\rangle$, $r \neq r'$. Since FS processes do not generate incorrect outputs and $r \neq r'$, we have $j \neq j'$ and $c \neq c'$.

With no loss of generality, let us suppose that $c < c'$. When $P_{c'}$ installed itself as the coordinator, the *TransferHistory* it used could not have had $\langle ORDER, c, o, D(r)\rangle$; otherwise, $P_{c'}$ could not have re-used o for r' and proposed $\langle ORDER, c', o, D(r')\rangle$ (lines C2 and C3 in Fig 3.9).

Note that P_j must have acknowledged $\langle ORDER, c, o, D(r)\rangle$ before it receives fail-signal from P_c; otherwise, $\langle ORDER, c, o, D(r)\rangle$ would have been purged from the *Order_Pool* (see line 3.4 in Fig 3.7) and would not have been included in the *STATUS* message sent during the install attempt being made later by p_i, $i > c$. Therefore, for $P_{c'}$ to have ignored P_j during its install, it must have had P_j in its *Signalled*; that is, P_j has fail-signalled and entered the terminal, halted state by the time $P_{c'}$ became the coordinator.

The correct p_i executing *install*() is clearly a later event compared to $P_{c'}$ becoming the coordinator. This is indicated by the causal chain of events expressed using Lamport's "*happened before relation*" denoted simply as "→" [Lam78] as below.

P_j fail-signals (sends FS_j) → $P_{c'}$ receives FS_j → $P_{c'}$ becomes coordinator (ignoring P_j) → $P_{c'}$ sends $FS_{c'}$ → p_i receives $FS_{c'}$ → p_i sends *INVITE* (executes *install*()).

∴ P_j fail-signals → p_i sends *INVITE*.

Since there is no failing state for an FS process, the fail-signalled and halted P_j could not be sending a *STATUS* message containing $\langle ORDER, c, o, D(r)\rangle$ to p_i, as assumed. Lemma is thus proved by contradiction.

Lemma 2: Let P_i be a perfect FS process. Say, p_i or p'_i multicasts an $ACK(o)$ for $\langle ORDER, c, o, D(r)\rangle$. At the time of this multicast, if a correct p_k implementing FS process P_k, $k \neq i$, has $\langle ORDER, c', o, D(r')\rangle$, $r \neq r'$, in its *Order_Pool*, then P_k has fail-signalled.

Proof: Since p_i and p'_i are both correct, with no loss of generality, we will focus only on p_i whose output behaviour is similar to that of p'_i. Let $Signalled_i$ denote the set *Signalled* maintained by p_i. p_i acknowledging an order from P_c implies that $P_c \notin Signalled_i$.

Let us first note that the FS process P_c could not have sent $\langle ORDER, c, o, D(r)\rangle$ to p_i and $\langle ORDER, c, o, D(r')\rangle$ to p_k. Therefore $c' \neq c$. There are two possibilities: $\langle ORDER, c', o, D(r')\rangle$ in the *Order_Pool* of p_k must have arrived from a coordinator that either

(i) Precedes P_c, i.e., $c' < c$ or

(ii) Succeeds P_c, i.e., $c' > c$.

In the first case, since P_c is re-using o for $r \neq r'$, it would not have received $\langle ORDER, c', o, D(r')\rangle$ in any of the *STATUS* messages received from $P_j \in (\Pi - Signalled_c)$ when it was installing itself as the coordinator. Therefore if a correct p_k has $\langle ORDER, c', o, D(r')\rangle$, in its *Order_Pool*, then $P_k \in Signalled_c$ when it was installing itself as the coordinator i.e. P_k has fail-signalled before P_c multicast $\langle ORDER, c, o, D(r)\rangle$.

Let us now consider the second case of $c' > c$. That is, $P_{c'}$ is a successor coordinator to P_c. We would show that $P_{c'}$ could not have become coordinator while p_i is acknowledging $\langle ORDER, c, o, D(r)\rangle$. For $P_{c'}$ to have become coordinator, $P_{c'}$ would have received *STATUS* message from all $P_j \in (\Pi - Signalled_{c'})$. Since as per the hypothesis, P_i is a perfect process, it will never belong to $Signalled_{c'}$ or $P_i \in (\Pi - Signalled_{c'})$. P_i unicasts *STATUS* message only if it finds $P_{c'}$ *eligible*() (see line 4.2 and 4.3 in fig 3.7). But it cannot find $P_{c'}$ as *eligible*() when $P_c \in (\Pi - Signalled_i)$, $c < c'$. Therefore a correct p_k cannot have $\langle ORDER, c', o, D(r')\rangle$, $c < c'$, in its *Order_Pool* while $ACK(o)$ for $\langle ORDER, c, o, D(r)\rangle$ is being sent by p_i.

Lemma 3: Let P_i be a perfect FS process. Say, p_i or p'_i multicasts an $ACK(o)$ for $\langle ORDER, c, o, D(r)\rangle$. If a correct p_k or correct $p_{k'}$ implementing FS process P_k, $k \neq i$, does not have any $\langle ORDER, c', o, D(r')\rangle$, $r \neq r'$, in its *Order_Pool* at the time $ACK(o)$ is being sent, then it is never going to have one.

Proof: As in lemma 2, since P_i is a perfect FS process, we will use p_i to represent the behaviour of P_i. With no loss of generality, consider only p_k which is correct. Let T_1 be the real time (not observable by any process) when p_i sends $ACK(o)$ and T_2 be the instance when p_k emits a fail-signal, if at all. ($T_2 = \infty$ if P_k is a perfect FS process.) The lemma is obviously true if $T_2 \leq T_1$. So, we will suppose that $T_2 > T_1$.

Since FS processes do not generate incorrect messages, $c' \neq c$. Let us first consider the case of $c > c'$. P_c was installed as coordinator after $P_{c'}$ has fail-signalled which must be before T_1 and hence before T_2. So, P_k was operational when P_c was

installed. That is, the installation of P_c could not have been completed without P_k unicasting *STATUS* message. Therefore, p_k is aware of fail-signalling by $P_{c'}$ before T_1. So, p_k will not accept any order from $P_{c'}$ after T_1.

For the second case i.e. $c' > c$, let us assume to the contrary that $\langle ORDER, c', o, D(r')\rangle$ is multicast by $P_{c'}$ and exists in the *Order_Pool* of p_k. Since $P_{c'}$ is re-using o for $r' \neq r$, it would not have received $\langle ORDER, c, o, D(r)\rangle$ in any of *STATUS* messages from any $P_j \in (\Pi - Signalled_{c'})$. But P_i is a perfect process and therefore $P_i \in (\Pi - Signalled_{c'})$. This means that $P_{c'}$ must wait for *STATUS* message from P_i before it could install itself as coordinator. There are two possibilities regarding $A_{c'}$ at the time of installation of $P_{c'}$.

(i) $A_{c'} < o$. P_i will include $\langle ORDER, c, o, D(r)\rangle$ in its *STATUS* message. By lemma 1, no contradictory $\langle ORDER, c', o, D(r')\rangle$, $r \neq r'$ can exist in a *STATUS* set. $\langle ORDER, c, o, D(r)\rangle$ will be considered by $P_{c'}$ and hence o cannot be re-used for $r' \neq r$ as assumed.

(ii) $A_{c'} \geq o$. So, $P_{c'}$ has some $ORDER(o)$ in its *Order_Pool*. If this entry is $ORDER(o, r)$, then p_k could not receive $ORDER(o, r')$ from $P_{c'}$. Let us suppose that $P_{c'}$ has $\langle ORDER, s, o, D(r')\rangle$ in its *Order_Pool*. This means that $P_{c'}$ has received $ORDER(o)$ from P_s, $s < c'$, and acknowledged it.

Let $P_{s'}$ be the first coordinator to assign o to r' and multicasts $\langle ORDER, s', o, D(r')\rangle$. Obviously $s' \leq s$. By the first case discussed above ($c > c'$), it is not possible to have $s' < c$; also $s' \neq c$. So, $c < s' < c'$. Since $P_{s'}$ was the first coordinator to assign o to r', it must have had $A_{s'} < o$ when it was installed. By the sub-case (i) discussed above, $P_{s'}$ could not be multicasting $\langle ORDER, s', o, D(r')\rangle$. So, it is not possible to have $A_{c'} \geq o$ and $\langle ORDER, s, o, D(r')\rangle \in$ *Order_Pool* of $P_{c'}$.

Lemma 4: Let P_i be a perfect FS process. If p_i or p'_i multicasts an $ACK(o)$ for $\langle ORDER, c, o, D(r)\rangle$ then (i) p_i and p'_i will eventually multicast $COMMIT(o, r)$ and (ii) no correct p_k will ever multicast $COMMIT(o, r')$, $r \neq r'$.

Proof: With no loss of generality, we will consider only p_i since all arguments provided for p_i apply equally to p'_i as well. Let us start with the first part of the lemma. Consider the case where P_c does not fail-signal until the end of execution. So, P_c must send $\langle ORDER, c, o, D(r)\rangle$ to all $P_j \in \Pi$. It will eventually receive $ACK(o)$ from all $P_k \in (\Pi - Signalled_c)$ and will multicast $COMMIT(o, r)$.

On the other hand, let us assume that P_c fail-signals during the execution. No correct p_k can have *ORDER*(o, r') in its *Order_Pool*, $r \neq r'$ (Lemma 3). Therefore no later coordinator $P_{c'}$, $c' > c$, can ever transmit *ORDER*(o, r'), $r \neq r'$. In other words a successor coordinator either transmits *ORDER*(o, r) or fail-signals before transmitting an *ORDER*(o, r'), $r \neq r'$. In the later case, a time must come when P_i will finally become the coordinator and will transmit *ORDER*(o, r). Hence P_i will eventually receive *ACK*(o) from all $P_k \in (\Pi - Signalled_i)$ and will multicast *COMMIT*(o, r) in both cases.

The second part of this lemma can be proved using lemmas 2 and 3. We note that for an *ORDER*(o, r') to be committable at any correct process p_k, it needs to be in the *Order_Pool* of p_k (See definition of *committable*(o) in sub-section 3.4.3.2). But Lemma 2 and 3 state that if *ORDER*(o, r) is in the *Order_Pool* of P_i, *ORDER*(o, r') cannot exist in the *Order_Pool* of any correct p_k. Hence *COMMIT*(o, r') can never be multicast by any correct process p_k.

Theorem 1 (Safety) – If *ORDER*(o, r) is committed i.e. *COMMIT*(o, r) is multicast by some correct process p_j, then neither *COMMIT*(o, r'), $r \neq r'$, nor *COMMIT*(o', r), $o \neq o'$, will ever be transmitted.

Proof: Let P_i be a perfect FS process. Since both p_i and p'_i are correct and will behave identically, with no loss of generality, we will only focus on behaviour of p_i. p_j can only transmit *COMMIT*(o, r) when *committable*(o) is true (lines 2.2 and 2.3 in Fig 3.7). Since $P_i \in (\Pi - Signalled_j)$, *commitable*($o$) cannot be true at p_j without receiving *ACK*(o, r) from P_i (see definition of *commitable*(o) in subsection 3.4.3.2). This means that p_i has multicast *ACK*(o, r). Since a correct p_i produces *ACK* for an o only once (line 1.5 and the condition $o > A$ activating Task 1 in Fig 3.7), p_i will never produce *ACK*(o, r'), $r' \neq r$. Hence *commitable*(o) cannot be true for *ORDER*(o, r'), $r' \neq r$ and *COMMIT*(o, r') cannot be transmitted by any FS process.

Theorem 2 (Liveness) – For every request r sent by a client to all processes, a *COMMIT*(o, r) will be multicast by some correct order process p_j for some o.

Proof: We prove the theorem by contradiction. Let us assume to the contrary that request r was sent by a client and that a *COMMIT*(o, r) is never multicast by any correct order process. This implies that no perfect process produced *ACK*(o) (Lemma 4). This means that none of the coordinator FS processes generated *ORDER*(o, r), otherwise it

would have been *ACK*ed by at least a perfect process. Further on, for a coordinator P_c that found r not committed, there can be two lines of actions.

(i) P_c chooses not to produce an *ORDER*(o, r), or

(ii) P_c fail-signals before producing an *ORDER*(o, r),

Recall that there is at least one perfect FS process P_i that will eventually take up the coordinator role and will never fail-signal. Hence the above situation implies that there is at least one coordinator which never fail-signals but chooses not to order request r forever. This is not possible. Hence the assumption can not be true.

3.6 Critical Analysis

The protocol explained above is designed to solve the consensus problem using Fail-signal approach; but with the reduced two-state assumption. In this section, we consider the extended three-state FS process model containing failing state and comment on the problems this extension has to address. Furthermore, we lay the foundations for an advanced protocol by proposing design techniques that can resolve the upcoming issues. Finally, some optimizations are discussed to lead to a better design solution.

3.6.1 Discussion

We start our discussion by recalling the remark given in subsection 3.4.2.2. The scenario laid down there is a classical situation that highlights a subtle problem caused by asynchrony of the communication system. To re-cap, the asynchronous nature of the network can cause messages to arrive at various destinations in any order with unknown delays. With fail-signal being one of these messages, the processes may find out about signalled processes at relatively different times and more importantly, may have a different view about the set of messages that the signalled process is regarded to have been sent before fail-signalling. For example, if P_x multicast messages m_1 and m_2 and then transit to signalled state multicasting fail-signal, a process P_y may receive messages in the order m_1, fail-signal, m_2 while P_z may receive m_1, m_2, fail-signal. Noting this remark, we also recall that this is exactly how the behaviour of P_x can be described if it was in the failing state (see remark *failing state and the uncertainties* in section 3.1). That is, being in the failing state, P_x will transmit messages to P_y and P_z in different order to cause inconsistency. Assuming that network delivered messages in the sent order, P_y and P_z receiving messages in different order can only be attributed to P_x

deliberately sending multicasts in different order to different destinations. Hence, our first observation is as follows.

Observation 1: An asynchronous network can make a two-state FS process appear as if it is a three-state FS process.

Now let us consider the three-state FS process. Failing state is a transient state. That is, the failing FS process will eventually move to signalled state, given that the fail-signal is echoed back by the environment to the sender. Let T_{wf} be the instant when an FS process P_x moves from working to failing state and T_{fs} be the time when it moves from failing to signalled state. Let us assume the worst case of P_x exhibiting behaviour B3 (section 3.1) between these time instants and transmitting both computed outputs and fail-signals simultaneously. Finally, suppose that all fail-signals transmitted by P_x are delayed beyond T_{fs}. Hence, P_x may be sending some output messages to selected processes and fail-signals to others but the latter will be delayed. This scenario mimics what could happen in a system of crash processes: a crash process, say p_i, crashes while multicasting some messages in parallel and as a consequence some destinations receive these messages and others do not. For p_i, (T_{fs} - T_{wf}) is the duration in which it carries out a few parallel partial multicasts and causes inconsistency. After T_{fs}, it stops permanently. Whereas, P_x makes several partial multicasts at different instances during (T_{fs} - T_{wf}) and stops functioning permanently albeit after fail-signalling. However any attempt to resolve any inconsistency cause by partial multicasts in both situations will require same effort. Hence our second observation is:

Observation 2: An asynchronous network can make a three-state FS process appear as if it is a crash process making partial multicasts during some bounded but unknown amount of time interval.

3.6.2 Using 3-state FS process model to solve consensus

We know that solving consensus for crash processes in asynchronous network requires at least (2*f*+1) processes [DLS88]. Whereas Protocol-0 only uses (*f*+1) FS processes. These processes when working in 3-state model may behave as crash processes in special situations (as described for observation 2). Hence reaching consensus in such situations, for example, between T_{wf} and T_{fs}, with this redundancy *may* not be a straightforward task. However since an FS process multicasts fail-signal messages which are expected to arrive at their destinations eventually, an FS process cannot be called equivalent to a crash process beyond T_{fs}. Hence, we cannot say for sure if a

solution with (f+1) FS processes is impossible. We leave exploration of the minimal bound on redundancy for solving consensus using FS processes as a future work. However, a simple solution to the redundancy issue is to increase the number of FS processes to (2f+1) and apply a crash-tolerant solution as done by [MES03]. This increases the actual redundancy to 2×(2f+1) = (4f+2). But we know that minimal redundancy needed to solve consensus in asynchronous network for Byzantine faults is (3f+1). Hence we avoid to increase the total number of processes used in Protocol-0 from (3f+2) to (4f+2) but rather aim to reduce it to the optimal number (3f+1).

We propose a hybrid approach to design a consensus solution that combines the traditionally used concept of quorums with our fail-signal process abstraction. Quorums have also been used by BFT, which is the closest to our design. To achieve the targeted aim, we do not increase the number of processes to create quorums but make the following design choices:

(i) Include the unpaired processes, *viz.* $p_{f+2}, p_{f+3}, \ldots, p_{2f+1}$, also to execute the protocol like paired ones and play significant part in ordering requests.

(ii) Only allow an operative FS process to play the role of coordinator, as long as there is one in the system, due to its special, well-defined failure characteristics.

These choices lead to two open questions:

1. Given that (i) permits unpaired, Byzantine-prone, processes to take part in the protocol execution, how should the component processes of FS processes not acting as the coordinator behave: as single autonomous processes or still maintain the FS abstraction generating 2-signed outputs?
2. Does the last $(f+1)^{th}$ coordinator strictly have to be an FS process?

These questions are addressed below.

3.6.2.1 Reduced coupling within FS processes

Since unpaired processes are now allowed to participate as non-coordinators and will be sending single-signed output messages, we propose that the FS processes while working as non-coordinators may have their component processes loosened in coupling so that they can work independent of each other and produce single-signed messages. But since we want to retain the FS process abstraction to play the coordinator role, we adopt a mixed approach that keeps FS processes working in two modes of operation; *Active* and *Passive*. FS processes working in these modes are called active and passive FS processes respectively. Active mode maintains the tight coupling and an active FS

process works in the same way as an FS process has been originally defined to work. Whereas passive mode permits loose coupling between the paired processes requiring them not to generate double-signed outputs but use output verification only for failure detection. We now describe in detail the two proposed modes below.

A. Active mode

The FS processes used in Protocol-0 are now being named as Active FS processes. To summarise, while working in active mode, an FS process produces double signed responses; computed output or fail-signal or both as described by the three states (figure 3.1). Each process within the pair has sequencer and comparator modules to monitor the working of its counter part process using defined timeout values. Failures are announced by these modules in the form of fail-signal when faults are detected within the FS process. Once the correct process in the pair becomes aware of the fault, the FS process stops functioning and is said to move to Signalled state generating only fail-signals.

B. Passive mode

Passive mode is introduced to enable each process in the pair work independently and yet remain part of the FS process so that the mode can be switched to active at any time. Each of the constituent processes can now generate responses that reflect its own state along with those that represent the state of the whole passive FS process. The idea of generation of single-signed outputs then comes directly from the notion of working independently. However, the core concept of mutual checking needs to be retained keeping failure detection alive within the FS process to ensure safety property. Hence, the facility of generation of fail-signal is still required. Hence, two types of responses can be expected from a passive FS process; single-signed computed output and double-signed fail-signal. Since single-signed outputs are sent without waiting for the checking process to be completed, we say that the checking is done passively, hence the name passive mode. We also note here that outputs can no longer be guaranteed to be always correct due to being single-signed only.

The three-state model remains the same for a passive FS process as defined in section 3.1. However, the behaviour of a passive FS process in each of these states is as follows.

Working State:

Each constituent process produces only correct outputs (single-signed) as per the specifications of the computational program P (Recall section 3.1). This is assumed to be the initial state of all passive FS processes.

Signalled State:

A passive FS process transits to this state when a fault is detected by a correct constituent process. This detecting process thereon produces only fail-signals (double-signed). However, the faulty constituent process is capable of producing both single-signed output and double-signed fail-signal randomly as in failing state (see below).

Failing State:

This is a transient state just like it is for an active FS process. In this state, one of the constituent processes has become faulty but the fault has not been detected by the correct counterpart yet. Since, in passive mode, the two paired processes work autonomously, the correct process amongst the pair, keeps sending correct 1-signed outputs in failing state. Whereas a faulty constituent process can behave in a Byzantine manner.

Analysing the behaviour of a passive FS process in signalled and failing states, we find that a faulty constituent process can behave randomly in both the states. This behaviour combined with asynchrony may suppress all fail-signals with only incorrect outputs reaching the destinations. Hence the correct constituent process sends fail-signals in signalled state just to ensure that the destinations are informed about the failure eventually. Fail-signal in this case is to announce the fact that this passive FS process can no longer switch to active mode, if and when needed. This is obvious because active mode needs computation of double-signed outputs by the FS process which is no longer achievable. Note that the correct constituent process in the failing passive FS process can be made to continue producing outputs along with fail-signals, if needed. This leads us to the following design optimization.

3.6.2.2 The last coordinator

We first recall the reason behind using (f+1) FS processes in the system model for Protocol-0. Since the system allows at most f failures, it was necessary to have a spare FS process that can continue acting as the coordinator even after all failures occurred. When unpaired processes are also allowed to execute the protocol, the $(f+1)^{th}$ coordinator need not have to be an FS process. We recall that each fail-signal is

attributed to a fault within the sender FS process (Assumption 1 subsection 3.2.1). Hence after the occurrence of *all f* failures, it is safe to delegate the coordinator role to an unpaired process. Hence we propose a reduction in the number of FS processes to *f*. Note that the number of primary order processes collocated with service processes remain the same i.e., (2*f*+1). Therefore the total number of processes reduces to the minimal (3*f*+1).

3.6.2.3 Relaxing Assumption 2 (failure pattern assumption)

We know that once a fail-signal is received from an FS process, be active or passive, it can no longer be trusted to take up coordinator role. We propose that such an FS process should not only not be given the role of a coordinator but also be logically isolated and be forbidden from participation in ordering of requests. Hence we point to the following design implications:

a. There is only one active FS process and that is acting as the coordinator.
b. Once a fail-signal is received from any FS process (active or passive), the signalled FS process is considered logically isolated from the rest of the system at an appropriate instant.
c. Once a fail-signal is received from the coordinator FS process, coordinator change should be triggered eventually; this could also be the appropriate moment to isolate the component processes of other fail-signalling FS processes.
d. Once the failure of an FS process is made well-known within the system, both constituent processes of the signalled FS process may fail.

Last point indeed relaxes assumption 2 (subsection 3.2.1) by allowing second failure in an FS process but only after the failure has been made "well-known". This is the time needed for at least a majority of processes to learn of the fail-signal and exchange some information about their state at the time of this recognition. This is important to maintain safety. Let us assume P_x is the FS process within which a constituent process say p_x has failed and this failure has been detected by p'_x at time t_x. It appears that if D is the unknown (but finite) bound on message communication delays over an asynchronous network between two correct processes then the second failure in P_x can occur at least after $2D$ time has elapsed subsequent to t_x. More details on computation of this bound will be given in the next chapter. We state the relaxed assumption 2 as assumption 2A below.

Assumption 2A:

The processes p and p' within any given FS process do not fail 'simultaneously': if one of them, say, p fails then the other process p' does not fail at least until it observes p's failure and an interval of $2D$ time elapses subsequent to the observation, where D is the unknown (but finite) bound on the communication delays over the asynchronous network.

Assumption 2A leads to a new characteristic in the behaviour of FS process. Since both the processes in a pair can fail, a situation where the two malicious processes collude and generate doubly-signed undetectably invalid messages may arise. Hence an FS process does no longer maintain crash semantics but can behave in an arbitrary manner once the $2D$ time has elapsed after the first failure in the pair has been detected. We represent this malicious behaviour by adding a state called Byzantine state in the 3-state model (figure 3.10).

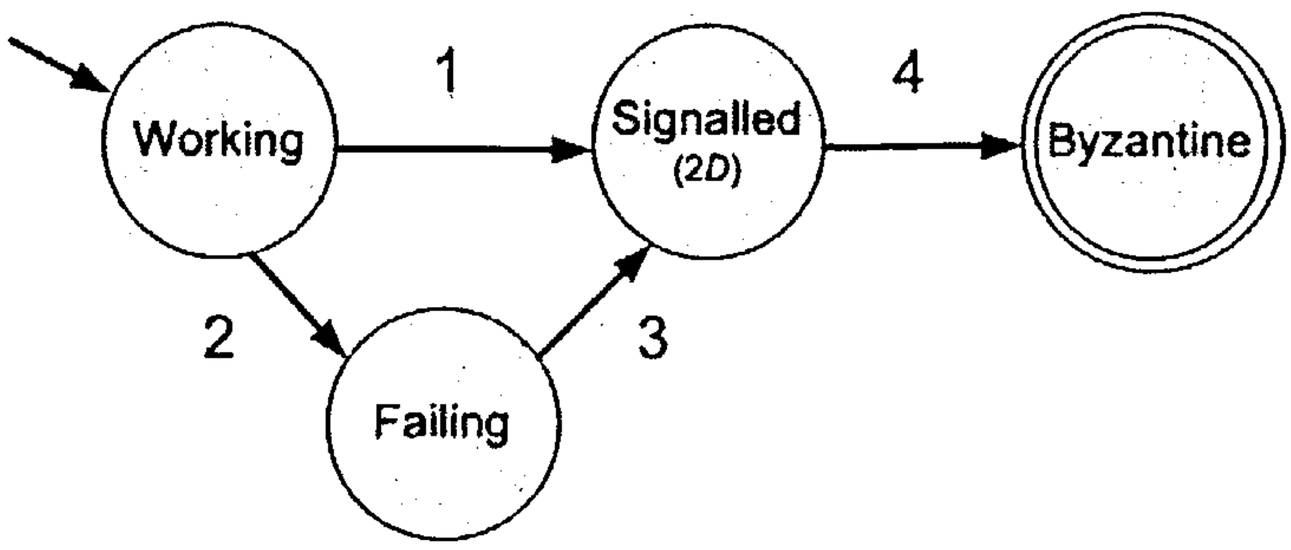


Figure 3.10. 4-State model of an FS Process with Assumption 2A

Byzantine state is the new terminal state of the 4-state FS process model. As mentioned earlier, transition 4 is allowed to occur only after the FS process has been in signalled state for at least $2D$ time. As the name suggests, behaviour of an FS process in Byzantine state is completely arbitrary and cannot be defined with certainty. Hence the protocols using this extended model need to take measures so that the double-signed output messages from an FS process in working state can be distinguished from the corrupted double-signed output messages sent by the same FS process in Byzantine state.

Assumption 2A is weaker than assumption 2. However, additional measures are needed to ensure that the $2D$ time limit holds between the two failures within the FS process. In other words, the second failure must be assured not to occur at least within $2D$ time of the first failure. This is similar to measures underlying the classical f-out-of-n assumption in which $(f+1)^{th}$ failure is assured not to occur within the system life time.

Hence, the known measures taken for failure independence and isolation to contain number of failures to f during the mission time would also be needed to realize assumption 2A.

We note here that guaranteeing failure isolation in the presence of malicious attacks is a challenging task. A recent paper [CL00] proposes proactive recovery mechanism to deal with this problem. This mechanism involves periodic and comprehensive rejuvenation of *every* node in the system: restarting the node with the fresh installation of OS and application software accessed from a secure, read-only memory. [RL04] additionally store a fingerprint of the software in a special register of the secure coprocessor which helps in detecting the corrupted code copy. A proactive recovery mechanism must deal with two related problems. Periodic rejuvenation must proceed without stopping the on-going order process. Hence, only a sub-set of nodes need to be rejuvenated at any given time and they cannot participate in the ordering activities until their rejuvenation is complete. Secondly, with no reliance on accurate failure detection, the nodes being rejuvenated at a given time can all be correct ones; therefore, the ordering process should be managed with reduced resilience capability during the time periods when rejuvenation is proceeding concurrently.

More recently, Sousa et. al. [SNV07] have analysed several significant works in asynchronous proactive recovery solutions including that of [CL00]. They argue that successful proactive recovery is not theoretically possible in classical asynchronous environments [SNL+06, SNV05]. The basis of their argument is that the periodic rejuvenation process must be *guaranteed* to have bounded completion time for it to be effective and this guarantee is not obtainable in asynchronous environments which are notionally time-free. The authors of [SLV07] then go on to advocate that some form of timing assumption is essential to guarantee successful proactive recovery and observe that the existing works on proactive recovery make such assumption but fail to formulate them explicitly. We believe that the advocacy in [SNV07] vindicates the presence of our $2D$ assumption since fail-signalling lays foundations for efficient proactive recovery in identifying only those nodes that warrant temporary isolation, concurrent rejuvenation and finally integration.

3.7 Summary

This chapter has studied conceptual and implementation details of fail-signal process. Benefit of using FS process to solve consensus problem is two-fold. Firstly it makes Byzantine fault appear as crash to the environment by deploying redundancy internally. Hence, a crash-tolerant protocol design would suffice. Secondly, it circumvents the FLP impossibility by signalling its failure; hence, the name fail-signal. However, its use has a couple of implications. Fail-signal sent by an FS process does not indicate the state of the process before signalling. Secondly, when the FS process fails, it may not stop producing computational outputs immediately and rather pass through a transient state with a benign two-facing behaviour. Therefore, a protocol that uses FS process to reach consensus needs to be capable of resolving any inconsistencies caused by these implications.

In this chapter we have taken a simple approach to avoid the problematic transient state i.e., failing state, temporarily and design a crash-tolerant protocol named Protocol-0 that uses FS processes to solve consensus for Byzantine faults. The design manages to keep the redundancy close to minimal for a Byzantine-tolerant protocol by using ($3f$+2) processes which includes (f+1) FS processes.

The design process and protocol's analysis brings out the intricacies involved in seeking consensus using FS processes. A further discussion on using 3-state FS process to achieve the same shows that the problem is similar to solving consensus for crash processes and can easily be solved by using ($2f$+1) FS processes. Aiming to use optimal redundancy, some ideas are floated for design of an advanced protocol. The proposal involves use of quorums with FS processes and aims to develop a hybrid system comprising of both paired and unpaired processes.

Chapter 4

Protocol I - Normal part

This chapter follows on from the design choices proposed in chapter 3 for consensus protocols using 4-state FS process. It presents algorithmic and implementation details of a coordinator-based Byzantine fault-tolerant protocol, named *Protocol-I*, that realizes these choices and caters for all implications. Furthermore, a comprehensive performance study is presented to evaluate the costs and benefits of using FS process abstraction, in its totality, to solve consensus.

Protocol-I comprises of two parts. First one is called *Normal Part*, which is executed by all processes to assign identical order numbers to each client request in failure-free situation. Second part is called *Install Part*. This part defines the sequence of steps to be executed in case a failure is signalled by the coordinator FS process. This chapter includes details of Normal part only whereas Install part is described in chapter 5.

The chapter starts by re-stating the assumption set with the relaxed version of assumption 2, presented as 2A in chapter 3. Then we present the new system architecture in which redundancy is reduced to the optimal ($3f+1$). Protocol-I is then introduced in section 4.3, starting with the newly added concepts of operative processes and quorum. Algorithm steps of Normal part are described in section 4.4 with details on the messages used for communication.

Finally, Protocol-I is compared both qualitatively and quantitatively with BFT, a Byzantine fault-tolerant total order protocol, well-known for its practicality. Two performance studies are presented with results from two sets of experiments. First study includes results from experimentation on a LAN cluster. Latencies of Protocol-I, BFT and a simple crash-tolerant protocol (derived from Protocol-I) are compared for various system sizes, workload and encryption techniques. Second study comprises of experiments performed in emulated WAN settings with similar parameters. Results show Protocol-I to be a feasible choice, specially in slower network settings.

4.1 Background

Protocol-I is a total order protocol that efficiently exploits fail-signal facility to tolerate Byzantine faults. Its design makes the following assumptions about the construction of FS process. Note that this assumption set combines assumption 2A proposed in subsection 3.6.2.3 with assumption 1 of subsection 3.2.1. For the sake of continuity, this new assumption set is restated below. Assumption 1 is renamed as 1A for uniformity.

1A. The delay estimates used for judging the timeliness of an order process are accurate and a non-faulty process never judges its non-faulty counter-part to be untimely.

2A. The processes p and p' within any given process pair do not fail 'simultaneously': if one of them, say, p fails then the other process p' does not fail at least until it observes p's failure and an interval of $2D$ time elapses subsequent to the observation, where D is the unknown (but finite) bound on the communication delays over the asynchronous network.

2A leads to a new characteristic in the behaviour of FS process, as explained in subsection 3.6.2.3. This is illustrated in figure 3.10 by introduction of Byzantine state to form the 4-state FS process model. In short, since 2A lets both the processes in a pair fail, it allows a situation where the two malicious processes collude and generate doubly-signed undetectably invalid messages. Hence an FS process no more has crash semantics but can behave in an arbitrary manner once the $2D$ time has elapsed after the first failure has been detected among the pair. Protocol-I uses this extended 4-state FS process to solve consensus optimally in the presence of Byzantine faults. The following section outlines the system model used in Protocol-I design.

4.2 System Model

The system architecture will be the same as Protocol-0 except that the number of FS processes is reduced to f and only one among them will be working in active mode at a time.

The system is considered to have at least $(2f+1)$, $f \geq 1$, service processes s_i, $1 \leq i \leq (2f+1)$, executing on distinct processors as before. The service processes are co-located with order processes p_i, $1 \leq i \leq (2f+1)$. Protocol-I uses f more nodes, each hosting a shadow process p'_i to form an FS process P_i (upper case P) with p_i, $1 \leq i \leq f$. Since number of shadow nodes used here is one less than that used in Protocol-0, the total

number of replicas is $3f+1$, which is the standard lower bound on a f-Byzantine tolerant deterministic solution to consensus problem in asynchronous environment. System architecture produced below as fig 4.1 reflects this change. Like Protocol-0, Protocol-I is also a deterministic coordinator based protocol. Therefore every order process is a deterministic state machine. The system can be represented as the following sets of order processes.

Π = Set of all FS Processes = $\{P_1, P_2, \ldots, P_f\}$

π = Set of all order processes co-located with service processes = $\{p_1, p_2, \ldots, p_{2f+1}\}$;

π' = Set of all shadow order processes (not co-located with service processes)

= $\{p'_1, p'_2, \ldots, p'_f\}$.

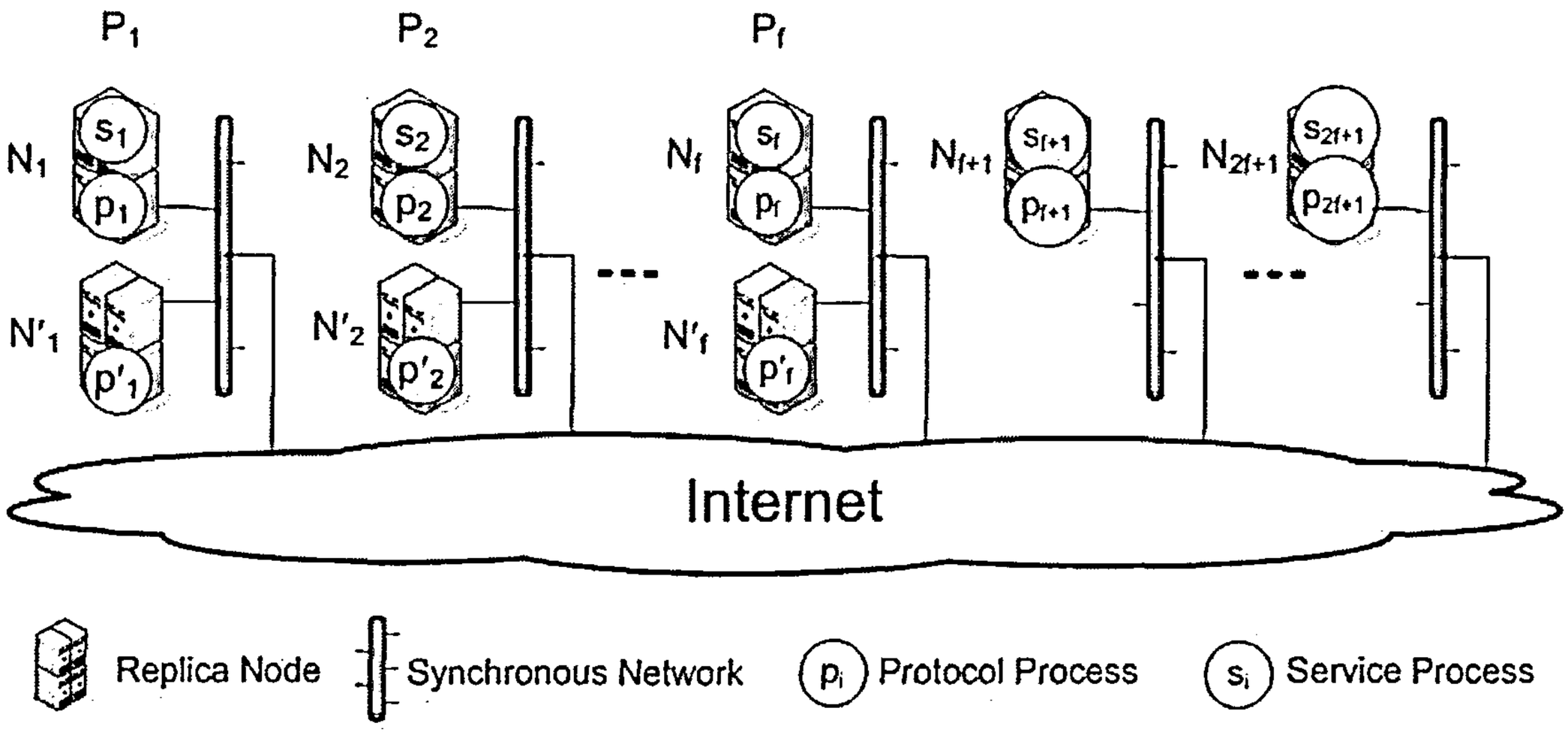


Figure 4.1. System Architecture used by Protocol-I

4.3 Protocol-I

The protocol sequentially ranks all f FS processes, denoted by $\{P_i: 1 \leq i \leq f\}$ and $Rank(P_i) = i$. This ranking is known to all processes in the system and defines the order in which coordinator role will be taken up by a pair with P_1 being the first coordinator. While the coordinator FS process works in active mode, all other FS processes work in passive mode. The protocol regards the system to be moving through a series of at most $(f+1)$ *configurations* which are distinguished by the process that is currently acting as the coordinator. For instance, Σ_1 is the initial system configuration with P_1 acting as coordinator. When P_1 fail-signals after working for a while, if *eligible*() = 2 (see sub-

section 3.4.3.2) then the system is regarded to move on to the second configuration Σ_2. These two configurations are represented below.

$\Sigma_1 = \{P_1, p_2, \ldots, p_{2f+1}, p'_2, p'_3, \ldots, p'_f\}$

$\Sigma_2 = \{p_1, P_2, \ldots, p_{2f+1}, p'_1, p'_3, \ldots, p'_f\}$

Note that in Σ_1, p_1 and p_1' are represented as P_1 while in Σ_2, P_2 represents p_2 and p_2' together. This change of coordinator goes on until all f FS Processes have fail-signalled, which is when any unpaired order process from the set $\{p_{f+1}, p_{f+2}, \ldots, p_{2f+1}\}$ can be randomly chosen to take over as the $(f+1)^{th}$ coordinator in Σ_{f+1}. (This unpaired order process is fixed to be p_{f+1}.) Since there can only be at most f failures in the system (see assumptions in section 1.4), the $(f+1)^{th}$ coordinator, which is the only unpaired coordinator ever to take over, is guaranteed to be non-faulty and will be the last coordinator. Thus the f^{th} and $(f+1)^{th}$ system configuration will be:

$\Sigma_f = \{p_1, p_2, \ldots, P_f, p_{f+1}, \ldots, p_{2f+1}, p'_1, p'_2, \ldots, p'_{f-1}\}$

$\Sigma_{f+1} = \{p_1, p_2, \ldots, p_{2f+1}, p'_1, p'_2, \ldots, p'_f\} = \pi \cup \pi'$

We note that in a given execution of the protocol, the system need not transit through every configuration defined here. For example, if in Σ_2, p_3 or p'_3 (implementing passive FS process P_3) fail-signals, then the system will not enter Σ_3 at all when P_2 fail-signals in Σ_2. This issue will be considered in detail while we discuss the part of protocol that deals with change of configuration in chapter 5.

4.3.1 Operative processes and quorum

Following the terminology of Paxos [Lam98], the term *acceptor* is used to denote an order process that is allowed to participate in an order assignment. A set of all acceptors in configuration Σ_i, $1 \leq i \leq (f+1)$, is denoted by $Acceptors_i$. For example, at system start

$Acceptors_1 = \pi \cup \pi' = \{P_1, p_2, \ldots, p_{2f+1}, p'_2, p'_3, \ldots, p'_f\}$.

Following the fail-signalling by P_1, if the system moves to Σ_2, then $Acceptors_2$ becomes:

$Acceptors_2 = \pi \cup \pi' \setminus \{P_1\} = \{P_2, p_3, \ldots, p_{2f+1}, p'_3, \ldots, p'_f\}$

Generalizing, $Acceptors_i$, $1 \leq i \leq f$, in Σ_i will be:

$Acceptors_i = \pi \cup \pi' \setminus \{P_1, P_2, \ldots, P_{i-1}\} = \{P_i, p_{i+1}, \ldots, p_{2f+1}, p'_{i+1}, \ldots, p'_f\}$;

After all f FS processes have fail-signalled, $Acceptors_{f+1}$ will contain all unpaired order processes with one of them behaving as coordinator i.e.

$Acceptors_{f+1} = \pi \cup \pi' \setminus \Pi = \{p_{f+1}, p_{f+2}, \ldots, p_{2f+1}\}$

Any process p_i or $p'_i \notin Acceptors_j$ is called a *learner* in Σ_j. A learner is not actually involved in order assignment but simply learns of each decision made by acceptors so that it can convey the decision to the locally hosted service process, if any.

A *quorum* is defined as any subset of *Acceptors* that satisfies the following two properties.

Intersection property - Any two quorums have at least one correct process in common.

Availability property - There is always a quorum containing no faulty process.

The protocol is designed to make progress only after a quorum agrees on a decision. To make sure processes in any two quorums decide consistently (hence correctly), intersection property is necessary. On the other hand, availability property is needed to ensure that liveness is achieved.

Size of a quorum in configuration Σ_i, $1 \leq i \leq f+1$, is denoted as VQ_i and called *Valid Quorum Size*. VQ_i can be any integer that satisfies both the intersection and availability properties.

Calculating the range of VQ_i:

If $n_i = |Acceptors_i| =$ Number of acceptors in $\Sigma_i = n - 2(i - 1)$.

$f_i = f - (i - 1) =$ Maximum number of processes that can fail in $Acceptors_i$.

1. To satisfy the Availability property,

$$VQ_i \leq (n_i - f_i) \tag{4.1}$$

2. To satisfy the Intersection property, at least one correct process is required to be common in any two quorums. In other words, the smallest number of correct processes that can be in a quorum should be more than half of the smallest number of correct processes in the system i.e. $(VQ_i - f_i) > (n_i - f_i)/2$ or

$$VQ_i > (n_i + f_i) / 2 \tag{4.2}$$

Intuitive Explanation: Consider two quorums A and B in Σ_i. When relation 4.2 is satisfied

$$|A| > (n_i + f_i) / 2, \qquad |B| > (n_i + f_i) / 2$$

Consider the worst case when all f_i failed processes are members of both A and B i.e. number of correct processes in A and B is $(|A| - f_i)$ and $(|B| - f_i)$ respectively. We have

$$(|A| - f_i) + (|B| - f_i) > (n_i - f_i)$$

Since there can only be at most $(n_i - f_i)$ correct processes in $Acceptors_i$, A and B must have one common correct process in them.

3. Analysing relations 4.1 and 4.2 above, we find that for a value of VQ_i to satisfy both the conditions, $(n_i + f_i) / 2$ is required to be less than $(n_i - f_i)$ for all values of i. A proof to show that this requirement is always satisfied for all values of i is given in figure 4.2.

To Prove: $(n_i + f_i) / 2 < (n_i - f_i)$, $1 \leq i \leq f+1$
Proof by induction:
Step 1. Base case - for $i = 1$, $n_i = n = 3f+1$ and $f_i = f$
$(n_i + f_i) / 2 < (n_i - f_i)$
$\Rightarrow (3f + 1 + f) / 2 < (3f + 1 - f)$
$\Rightarrow 2f + 0.5 < 2f + 1$
Hence the relation is proved for base case.
Step 2. Hypothesis - Let us assume that the given relation is true for $i = k$ i.e.
$(n_k + f_k) / 2 < (n_k - f_k)$
Step 3. To prove - The relation is also true for $i = k+1$ i.e.
$(n_{k+1} + f_{k+1}) / 2 < (n_{k+1} - f_{k+1})$
From the definition of n_i and f_i given above
$n_{i+1} = n_i - 2$ and
$f_{i+1} = f_i - 1$
The given relation for $i = k+1$ can be written as
$(n_k + f_k - 3) / 2 < (n_k - f_k - 1)$
$\Rightarrow (n_k + f_k) / 2 - 1.5 < (n_k - f_k) - 1$
$\Rightarrow (n_k + f_k) / 2 < (n_k - f_k) + 0.5$
Hence, by induction, the relation is proved to be true for all $i \geq 1$.

Figure 4.2. Proof for the bounds on Quorum size

Hence, a valid quorum size VQ_i for configuration Σ_i can be given as

$$(n_i + f_i) / 2 < VQ_i \leq (n_i - f_i) \qquad (4.3)$$

Let Q_i be the smallest value of VQ_i i.e.

$$Q_i = \lfloor (n_i + f_i) / 2 \rfloor + 1$$

Although any value in the range given by 4.3 can be chosen, keeping efficiency considerations in mind, we choose Q_i to be the size of quorum in Σ_i throughout the description of Protocol-I.

As mentioned earlier, when a fail-signal is received from the current coordinator P_c, i.e., when system moves from current configuration Σ_c to the next, say Σ_{c+1}, P_c is discarded from $Acceptors_{c+1}$. At the same time n_c, f_c and Q_c are also updated respectively to n_{c+1}, f_{c+1} and Q_{c+1}. Consider the special case where P_{c+1} has already fail-signalled while P_c fail-signals and $eligible() = c'$, $c' > c+1$ (see subsection 3.4.3.2 for definition of $eligible()$). So the system has to move from Σ_c to $\Sigma_{c'}$. In this situation, n_c and f_c are updated to $n_{c'} = |Acceptors_{c'}| = n - 2(c'-1)$ and $f_{c'} = f - (c' - 1)$ respectively. In

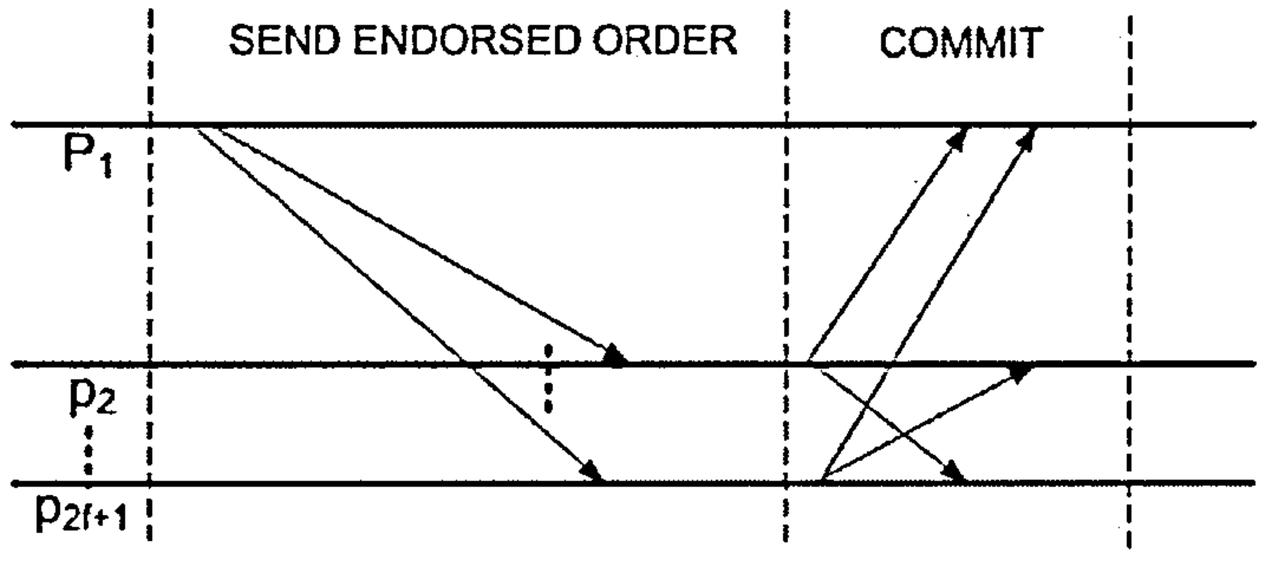
this way, the suffix used with *Acceptors*, Q and Σ remains same and identifies the coordinator number.

***Note*:** In the above scenario with $P_{c'}$ being *eligible*(), there may be a process $P_{k'}$, $k' > c'$, which has already fail-signalled i.e., $P_{k'} \in Signalled_{c'}$ (see subsection 3.4.2 for $Signalled_j$). However, despite this fail-signal, $P_{k'} \in Acceptors_{c'}$. Hence, update of n_c and f_c to $n_{c'}$ and $f_{c'}$ respectively only incorporates fail-signals sent by each FS process P_k, $c < k < c'$.

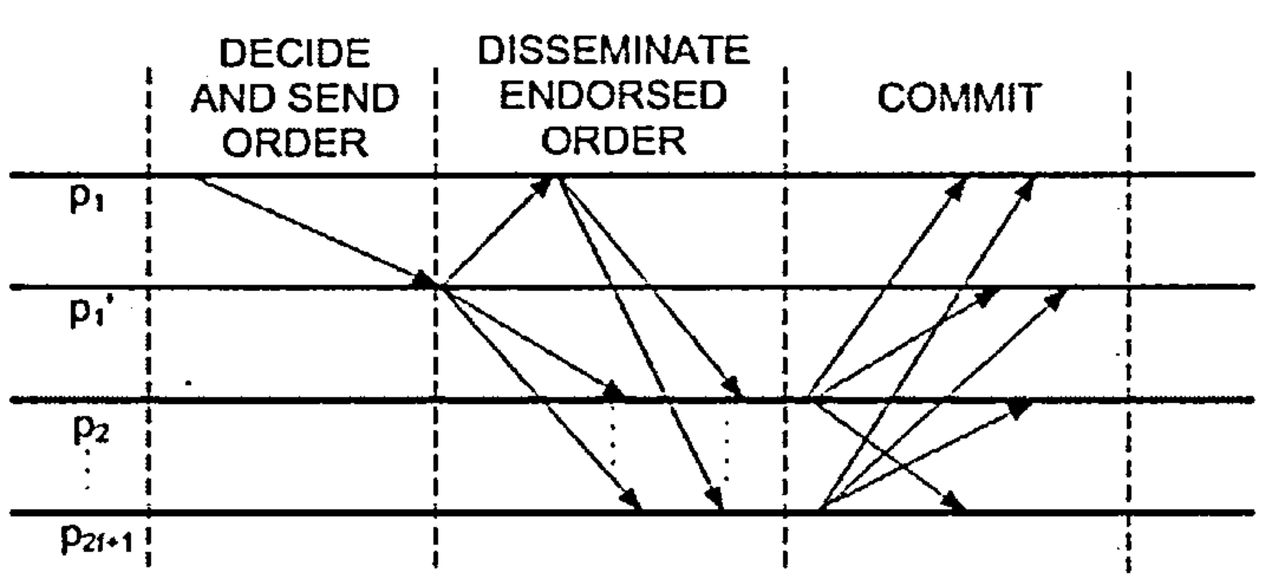
After introducing the concept of acceptor and configuration, we redefine the two parts of Protocol-I as follows. Normal Part is executed by all acceptor processes to assign identical order numbers to each client request when working within a configuration. Whereas, Install part defines the sequence of steps to be executed in case a failure is signalled by the coordinator FS process. This part covers the steps involved in a configuration change and also ensures that any order committed in one configuration is recognised in subsequent configurations. Description of Normal part is given in the following section.

4.4 Description of Normal Part

The normal part of the protocol is a two-step operation and is depicted in figure 4.3(a) with P_1 acting as the coordinator (i.e. configuration Σ_1). The first step, called "Send Endorsed Order", involves processing of a client request by coordinator P_c, which includes deciding on assignment of a unique sequential order number to the request and multicasting the doubly-signed order decision to all processes in the system. When an acceptor receives an order message, it sends acknowledgement to all processes including the coordinator. This constitutes the second step, called "Commit", which ends at a process when it receives acknowledgements from a quorum. This renders the order committed i.e. irreversibly assigned to the request. While collecting responses from a quorum, order message itself is considered as a sort of acknowledgement from the two processes forming the coordinator FS process.



(a) High-level illustration



(b) Detailed illustration

Figure 4.3. High-level and Detailed illustration of Normal Part Operation

First step of Normal part can be further broken down into two sub-steps each pertaining to a message communication between the two processes in the coordinator FS process, illustrated in figure 4.3(b). First sub-step is for the order process to assign a unique order number to the request and send it to its counter part. Second sub-step is for the shadow process to check and endorse the received order and multicast to all. Therefore, the three steps altogether make Protocol-I a three phase communication protocol, shown in fig 4.3(b).

4.4.1 Message Formats

All processes use the following message structures during protocol execution.

ORDER$_c(o)$**:** Order message is prepared by a coordinator P_c and is of the form $\langle ORDER, c, o, D(r)\rangle$, where o is the unique order number assigned to request r. Recall that the clients are correct and direct their requests to all nodes; hence, every process in $\pi \cup \pi'$ receives r and the *ORDER* for r does not have to contain r itself. Suffix is dropped from $ORDER_c(o)$ in this thesis where distinction of sender is not important.

ACK$_i(o)$**:** *ACK* message is prepared by any non-faulty process p_i in $Acceptors_c$ in response to receiving and accepting $ORDER_c(o)$ and is of the form $\langle ACK, c, o, D(ORDER_c(o)), i\rangle$. Note that *ACK* of Protocol-I contains digest of $ORDER_c(o)$ instead

of that of *r*, which was the case in Protocol-0. Also *ACK* here additionally contains coordinator number *c*.

4.4.2 Algorithm Steps

This subsection describes each step of Normal part in detail. It is assumed in the text below that all the messages received are correct and authentic. In other words, a message is considered a received message only after it has been delivered by the underlying communication system and it has passed all syntactic and semantic checks e.g. it has been authenticated, its structure is according to the specifications, its sender is in $Acceptors_c$ etc. Hence a message is only considered by any process p_i for further processing once it has passed these initial checks.

For a client request message *r*, the coordinator P_c assigns a unique sequential order-number *o* if all earlier messages from that source have been assigned a (lower) order number. It then sends an $ORDER_c(o)$ message to all order processes in $\pi \cup \pi'$. We describe the last phase of the normal part with the help of following four actions: *Accept*, *Acknowledge*, *Commit* and *Deliver*. These actions are depicted in figure 4.4 below indicating the sequence in which they are performed. The third action i.e., Commit marks the end of COMMIT phase and initiates the fourth action.

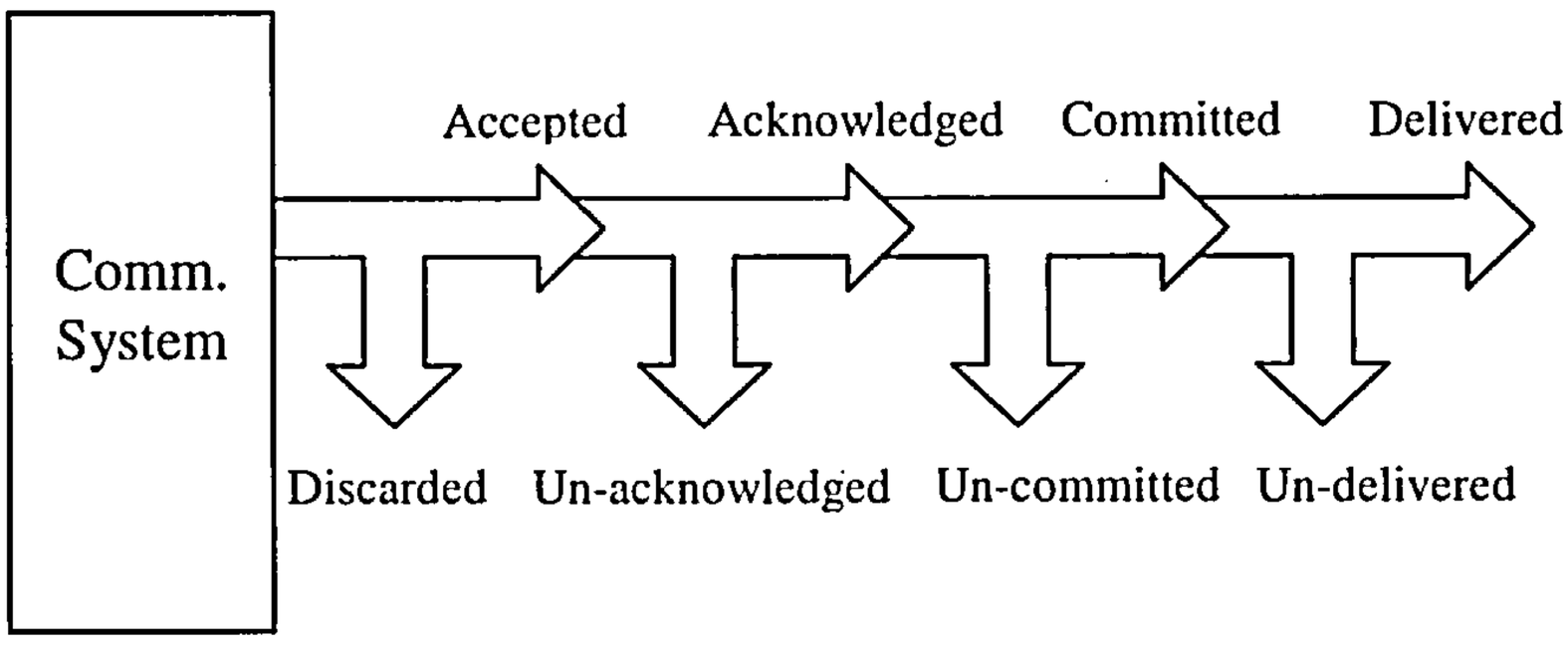


Figure 4.4. Sequence of actions performed in COMMIT Phase

1. Accept: An acceptor p_i *accepts* the $ORDER_c(o)$ only after one of the following conditions is satisfied

(i) If $P_c \notin Signalled_i$, $ORDER_c(o)$ is received by p_i, or

(ii) If $P_c \in Signalled_i$, $ORDER_c(o)$ has been received by p_i from at least (f_c+1) acceptors in Σ_c.

2. Acknowledge: Receipt of $ORDER_c(o)$ by any non-coordinator acceptor p_i, does not give any guarantee to p_i that all other order processes have received a copy of that

$ORDER_c(o)$. To ensure diffusion of this order decision, an accepted $ORDER_c(o)$ is *acknowledged* by p_i, by multicasting $ACK_i(o)$ to all order processes (including itself) and $ACK_i(o)$ essentially contains p_i's signature for $D(ORDER_c(o))$. Any acceptor p_i *acknowledges* the $ORDER_c(o)$, after the following 5 conditions are met: p_i has

(i) accepted $ORDER_c(o)$,

(ii) received r from the client whose digest is the same as $D(r)$ in $ORDER_c(o)$,

(iii) acknowledged an *ORDER* message for all earlier requests (if any) from the client of r,

(iv) acknowledged all earlier *ORDER* messages (with order number smaller than o), and

(v) not accepted any *ORDER* message with the same o but for different r.

Conditions (iii) and (v) are trivially satisfied so long as the coordinator has not fail-signalled – this exception is ignored for the time being. We denote the largest o *ACK*ed by p_i as A_i.

3. Commit: An $ORDER(o)$ is said to be committed i.e. the predicate *committable*(o) is true at *any* p_i if the following conditions are true

1. $ORDER(o)$ is acknowledged by p_i.

2. $ORDER(o')$, $o' \geq o$, is known to have been acknowledged by a quorum in Σ_c. This knowledge is deduced by receiving $ORDER(o')$ from coordinator process or authentic $ACK_j(o')$ from a non-coordinator acceptor p_j. This knowledge is necessary to ensure that no non-faulty process ever commits request r for a different $\hat{o}$ or commits a different request r' for the same o.

The set of messages that lead to committing $ORDER(o)$ will be referred to as the *proof_of_commitment* for o and will be retained until $ORDER(o')$, $o' > o$, is committed. The largest o' committed by p_i will be referred to as $max\_committed_i$.

Commitment of $ORDER(o)$ renders every $ORDER(o'')$, $o'' < o$ committed. This is because of the in-sequence acknowledgement process (condition (iii) above). Possession of *proof_of_commitment* for $ORDER(o)$ and condition (iii) guarantee that $ACK_k(o'')$ have already been sent by at least every non-faulty acceptor p_k in a quorum. Some of these $ACK_k(o'')$ may not have arrived at their destinations yet but according to the properties of asynchronous network considered, they will reach their destinations eventually.

4. Deliver: p_i *delivers r* for processing after it has *committed ORDER*(*o*) and, if $o > 1$, after it has delivered the request with order-number (*o*-1).

4.5 Qualitative comparison with Normal Part of BFT

To compare Normal part of Protocol-I with that of BFT side-by-side, we present the three phases which BFT executes to commit an *ORDER* in figure 4.5(b). (p_0 is acting as the BFT coordinator.) Note that the three phases of BFT involve: 1 to *n* (coordinator to all), *n* to *n*, and again *n* to *n* message transmission. The purpose of the *prepare* phase is to verify if the coordinator can be trusted in what it sent during the *pre-prepare* phase. When a process-pair (with *signal-on-fail* semantics) is acting as the coordinator, *n* to *n* transmissions of the *prepare* phase is obviously not needed and the 3-phase exchanges become: 1 to 1 (for endorsement), 2 to *n* (endorsed output to all), and *n* to *n*. Since *n* decreases for Protocol-I as more failures are signalled, the number of messages, being exchanged in a phase of *n* to *n* communication, and hence quorum size also decreases. Whereas *n* and hence quorum size remains constant for BFT. The last phase in both the protocols is to make sure that a quorum is in a position to commit a certain order. This helps smooth transition from one configuration to the next, in case of a failure (in Protocol-I) or a suspicion of failure (in BFT), without any loss of already committed orders. More details are presented in chapter 5 where this transition mechanism is actually revealed.

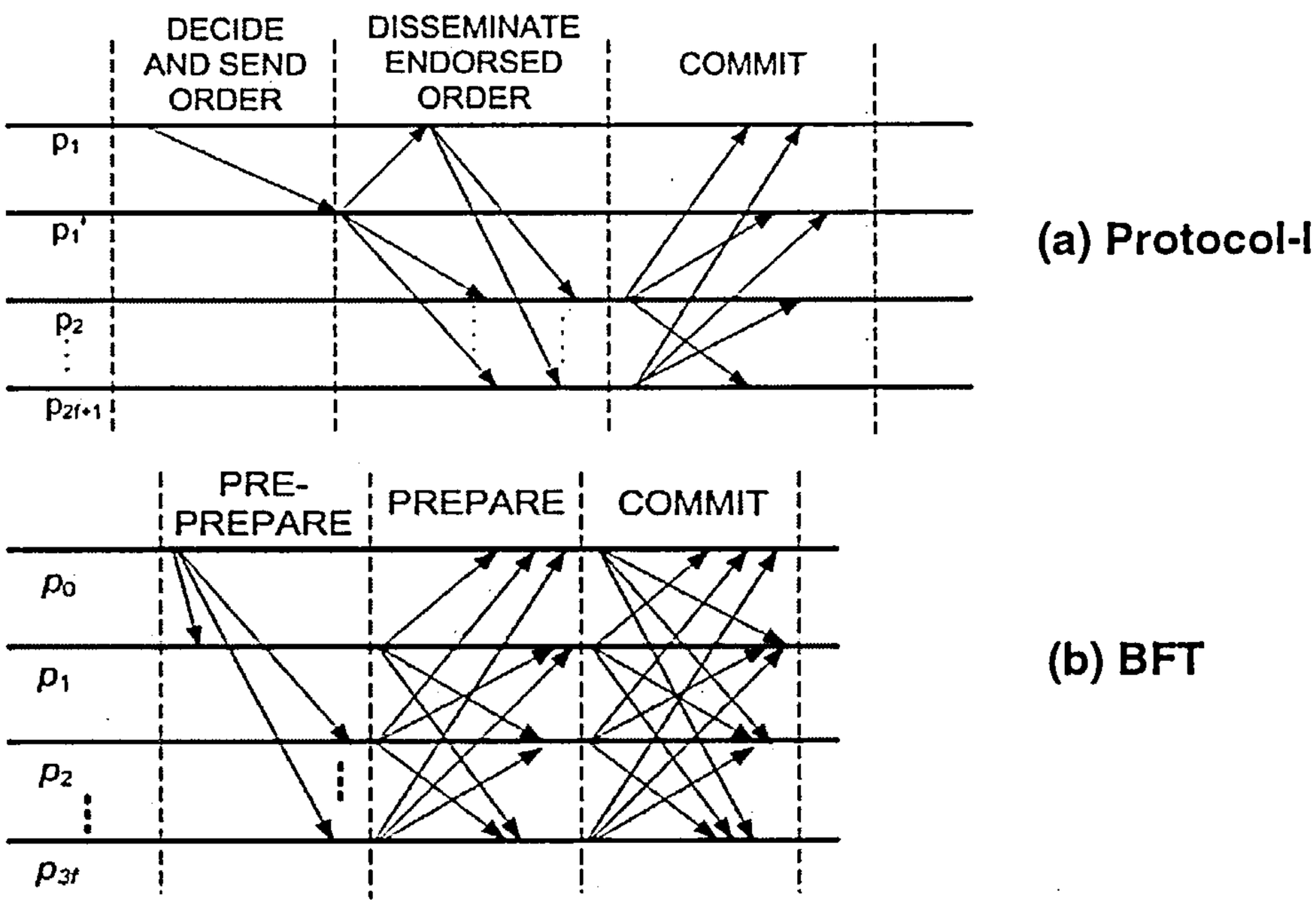


Figure 4.5. Fail-Free 3-phase Operation of the two protocols.

4.6 Implementation

This section explains the implementation details of the two protocols and compares the performance of the two protocols under various load conditions and network settings. There are two reasons of implementing BFT from scratch and not using the library which is made available by the authors of BFT. First is that this study focuses on performance comparison of the two protocols on the foundation level. Due to difference in basic way of working of the two in the three phases described in section 4.5, it was found important to compare the two in bare essentials. Hence any sophisticated optimizations need to be avoided and the concepts implemented in a simplistic way. Secondly a fair comparison is only possible when the two entities to be compared are built with same techniques and technology. Hence the two implementations should use same programming language (Java here) and tools to keep things on same level.

A third protocol was also implemented using same technology. This was a crash-tolerant version of Protocol-I (denoted as CT) and was included in the performance study so that the extent of slow-down in BFT and Protocol-I due to type of faults tolerated being Byzantine, instead of crash, can be evaluated. Protocol-I was amended to exclude shadow processes from the system, hence no pairing is used. Therefore system only has set of π processes ($\pi' = \varnothing$) with $n = 2f+1$, which is the known lower bound on solving consensus problem with crash tolerance in asynchronous network. Assuming that there is no adversary present in the system which can tamper with messages while on transmission line, cryptographic techniques are not needed for CT (all processes are crash-prone). Like protocol-I, CT is also coordinator-based and the coordinator process directly sends its order message to all other processes. Since CT does not take into consideration Byzantine faults, there is no reason for any non-coordinator process not to believe the coordinator on whatever order messages it transmits. Hence the third phase of Protocol-I becomes phase two for CT and an order message is committed after receiving (Q-1) acknowledgement messages. Diagrammatically the two phases of CT will be exactly same as the two phases of Protocol-I shown in figure 4.3(a), except that coordinator and the remaining paired processes of Protocol-I will be unpaired ones in CT. The two phases in CT therefore involve 1 to n (for sending order) and n to n transmissions (for commitment). Coordinator change is triggered by suspicions by acceptors. Like BFT, suspicions can be false. Hence configuration change does not change *Acceptors* and Q which are fixed

to π and (f+1) respectively. (In general for $n > 2f$, $Q = n\text{-}f$.) However, configuration change is not considered in this study.

4.7 Experimental Setup

Order latency and throughput of the three order protocols are measured and compared in fault-free scenarios in various network configurations. These are precisely defined below.

Latency: It is the time interval between the instance the request is batched by the coordinator and the instance a given process commits an order number (o) for that request. That is, latency does not include the time it takes to execute the 4^{th} action in figure 4.4.

Throughput: Throughput is measured as the number of messages committed by an order process per second.

The parameters that were varied to see performance dependency are described below.

1. **Batching interval:** For all three protocols, the coordinator process does not issue order for a request straight away but batches order messages over a period of time called the *batching_interval*. At the end of this interval, accumulated orders are transmitted together as a single message; the size of this message is called the *batch_size*. Similarly, all non-coordinator processes transmit their acknowledgements for a batch of order messages. Note that the larger this interval, the longer a given client request has to wait to be batched; so, the *batching_interval* should ideally be chosen as small as possible. Latency measured for this thesis does not include the time duration a received request spends waiting to be batched.
2. **Cryptographic techniques:** Three distinct combinations of message digest and signature schemes used are:
   - MD5 for computing message digests together with RSA scheme, key size 1024
   - MD5 with RSA scheme for key size 1536 and
   - SHA1 with DSA for key size of 1024.
3. **The fault-tolerance parameter (f)** takes the value of 1, 2 and 3.
4. **Network configuration:** Performance of the three protocols is measured on various network settings like LAN and WAN with replica processes assumed to be located at various distances apart. The three settings considered in this study are:

(a) Local Area Network (LAN)

(b) Inter-city Wide Area Network (Fast WAN)

(c) Inter-continental Wide Area Network (Slow WAN)

Experiments performed are divided into two categories. First set of experiments were run over an on-campus cluster of PCs i.e. a real LAN with parameters 1 to 3 being varied. Second set was conducted on the real LAN plus a fast WAN and slow WAN configuration emulated by a WAN Emulator with parameters 1 and 3 being varied. Both sets were run in configuration Σ_1 with P_1 acting as coordinator in Protocol-I. Similarly, p_1 plays the coordinator role in both BFT and CT. The two sets are described as two performance studies in sections 4.8 and 4.9 below. More details on these configurations are explained in the corresponding sections.

4.7.1 Strategies for varying workload of the system

System load can be increased in two ways. First is to have a fixed number of clients, each sending requests with increasing frequency. The other is to increase the number of clients, each sending requests at a constant rate. Both scenarios can be simulated by varying batching interval and batch size. For example, if batch size is fixed to a constant, decreasing batching interval can represent fast arriving requests. Alternatively, with a fixed batching interval, increasing batch size depicts the same situation. This thesis adopts the first method to measure performance i.e. decreasing batching interval to depict increasing system load. Although it is expected that both methods will have similar effect, second method may be used in future work to see the differences. One client is used in both sets of experiments to send requests to all replica processes.

4.7.2 Presentation of Results

Various network settings and traffic conditions may cause processes to take different amount of time to commit an order for a client request. Although this difference is not expected to be drastically big, it will only be fair to present statistics with corresponding process number. For presentation purposes, two processes are carefully chosen from all 4, 7 and 10 processes for $f = 1, 2$ and 3 respectively, such that their performance figures truly represent the whole set of processes. For instance in case of Protocol-I, one process from an FS process and one from a non-FS one is selected. Although all processes in BFT and CT are unpaired, latency values for the same processes are presented for these protocols as well to maintain consistency.

Note that the numbering schemes used in the two protocols to identify each process are different. For example, processes in Protocol-I are numbered as p_1, p'_1, p_2, p'_2, etc. while in BFT as p_1, p_2, p_3, etc. Hence, for the purpose of presentation of experiment results, we propose a general numbering scheme that represents equivalent processes of the two protocols with same numbers. This scheme is given in Table 4.0. Reader is referred to figure 4.11 for graphical illustration of the equivalent processes.

Protocol-I			BFT	General identification scheme
$f = 1$	$f = 2$	$f = 3$		
p_1	p_1	p_1	p_1	p_1
p'_1	p'_1	p'_1	p_2	p_2
p_2	p_2	p_2	p_3	p_3
p_3	p'_2	p'_2	p_4	p_4
	p_3	p_3	p_5	p_5
	p_4	p'_3	p_6	p_6
	p_5	p_4	p_7	p_7
		p_5	p_8	p_8
		p_6	p_9	p_9
		p_7	p_{10}	p_{10}

Table 4.0. General Representation Scheme for Processes in BFT and Protocol-I

Both performance study results show data for the following processes

- p_2 and p_4 (actually p'_1 and p_3 of Protocol-I) when $f = 1$,
- p_3 and p_7 (p_2 and p_5 of Protocol-I) when f=2 and
- p_4 and p_{10} (p'_2 and p_7 of Protocol-I) when f=3.

4.8 Performance Study I

The three protocols are implemented in Java using JDK 1.5 as a multi threaded application. These applications use TCP/IP sockets for any communication between processes. Protocol-I uses RMI instead of sockets for communication between paired processes only, to distinguish this communication from that between the unpaired ones.

The three protocol implementations were deployed on a cluster of 24 Linux machines (Fedora Core 4) connected by a 100 Mb Ethernet LAN. Each machine has a 2.80 GHz Pentium IV processor and 2GB RAM. 7 and 10 machines were used for $f = 2$

and 3 respectively plus one machine for client. The *batching_interval* is varied from 40 milliseconds (msec) to 500 msec, and the *batch_size* is fixed at 1 KB.

The results of this performance study are presented in the following manner. Figure 4.6 depicts order latency vs. batching interval for all three protocols with all three cryptographic techniques and f fixed at 2. Figure 4.7 depicts the same with f fixed at 3 but omitting MD5 with RSA key size 1536. Figure 4.8 shows throughput vs. batching interval for all three protocols with f fixed at 2 and the cryptographic techniques being DSA and RSA with key size 1024. Each point in a graph is an average over 100 experimental results.

4.8.1 Order Latency

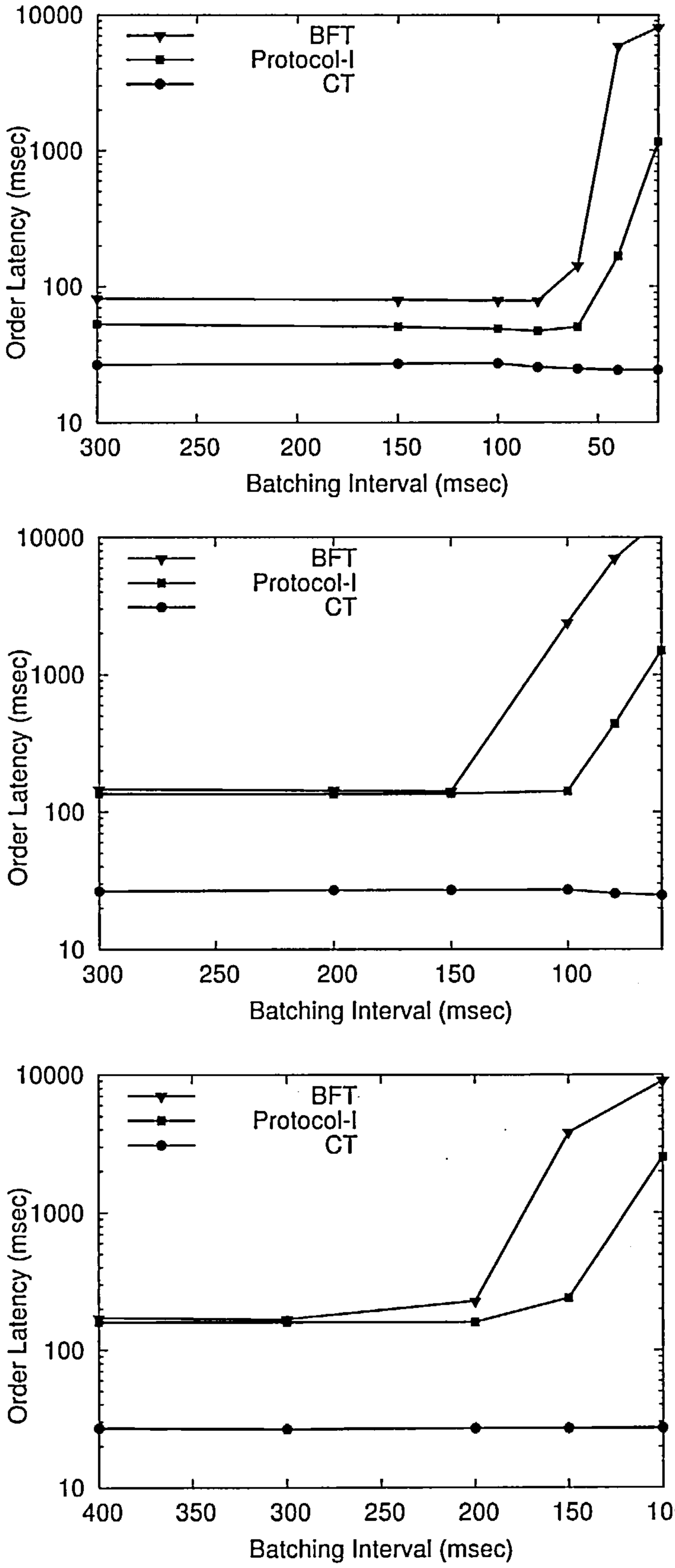
Figure 4.6 plots order latency observed at p_3 against batching interval for all three protocols. It can be seen that the order latencies stay nearly constant for large values of batching intervals, indicating that the system operates in steady-state (i.e., not in saturation region). They stay constant at 26 (and 23) msec for p_3 (and p_7) of CT, but increase drastically for BFT and Protocol-I when the batching interval decreases below a *threshold*, pushing the system operation into a 'saturation' region. (Note that latencies are represented in log scale along y-axis.) Further, the threshold for BFT is larger than that for Protocol-I. This indicates that BFT has a tendency to push the system into saturation earlier due to the large number of messages it places in the system and the cryptographic operations performed on each message. For the same reason, the steady-state latency for BFT is always more than that for Protocol-I.

As f (and hence n) is increased in figure 4.7, which shows order latency measured at p_4, it is observed that the saturation thresholds are encountered at larger *batching_intervals*, and the order latencies in the steady state increase. For example, comparing figure 4.6(c) with figure 4.7(b), it is observed that the steady-state latencies for Protocol-I shift from 160msec (168msec) for p_3 (p_7) and f=2 to 191msec (207msec) for p_4 (p_{10}) and f=3 and that for BFT increases from 171msec (174msec) to 236msec (243msec). It is interesting to see that the differences in steady-state latencies of Protocol-I and BFT increase considerably when crypto-technique is changed from RSA (Figures 4.7(a)) to DSA schemes (Figure 4.7(b)) and number of processes in the system is higher. For example, RSA with key size 1024 gives a difference of 19msec (19msec) between steady-state latencies of p_4 (p_{10}) for Protocol-I and BFT while the difference is

46msec (36msec) for p_4 (p_{10}) when DSA scheme is used for authentication. The explanation is as follows.

In both the schemes the time taken to sign a given message is similar; however, signature verification is much faster in the RSA scheme compared to DSA. Furthermore, in a typical n to n message exchange, each process signs one message while it needs to verify at least (n-f) messages. This suggests that DSA is generally not suited for Byzantine order protocols.

Similarly, the system starts saturating for Protocol-I when batching interval further decreases from the threshold value of 100msec for f=2 while the same reaches threshold point at 150msec for f=3. For BFT, the threshold value reaches even earlier at 300msec for f=3 as compared to 150msec for f=2. These observations can be attributed to the fact that as n increases, each individual process receives more messages which need to be authenticated and processed.

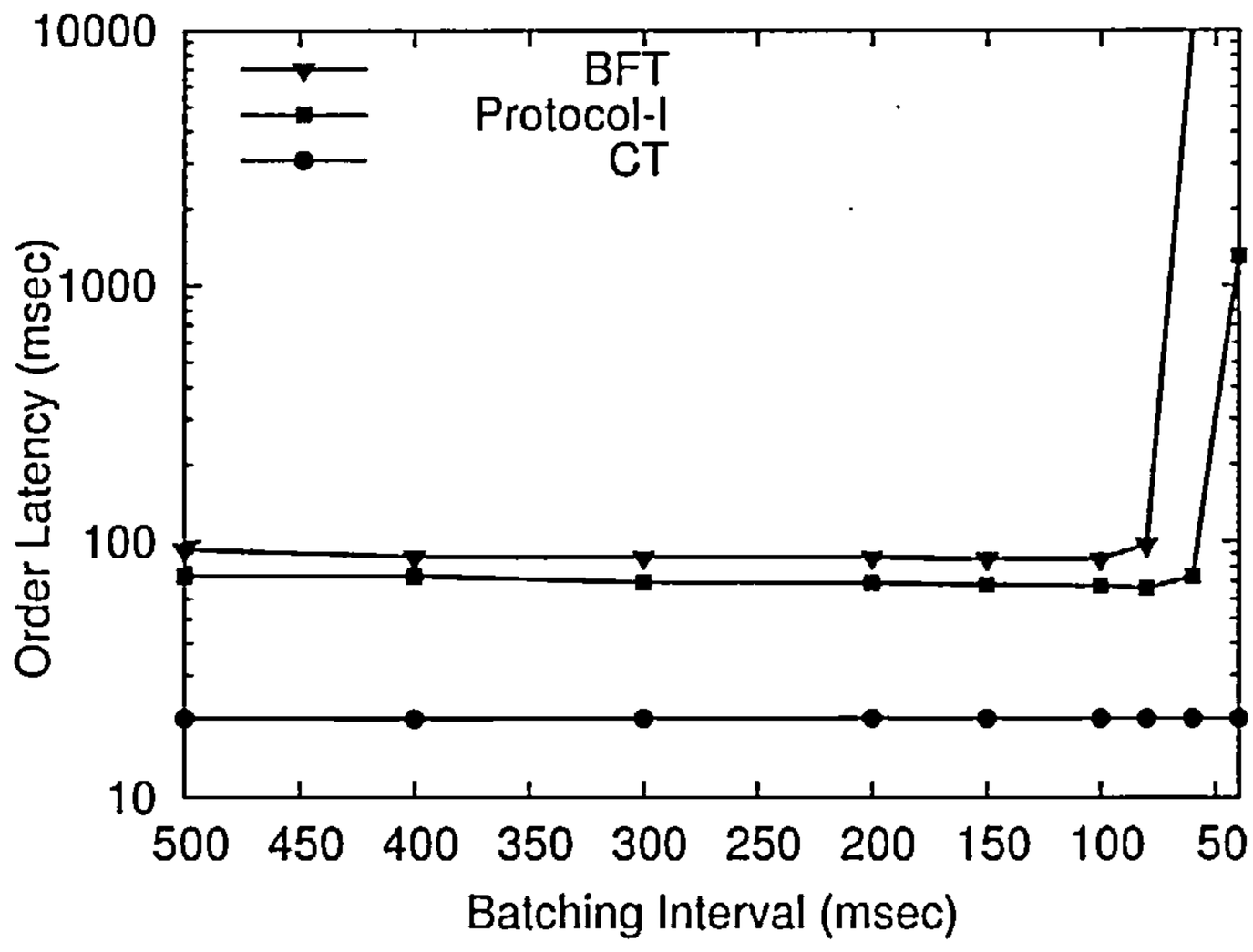


(a) MD5 with RSA key size 1024

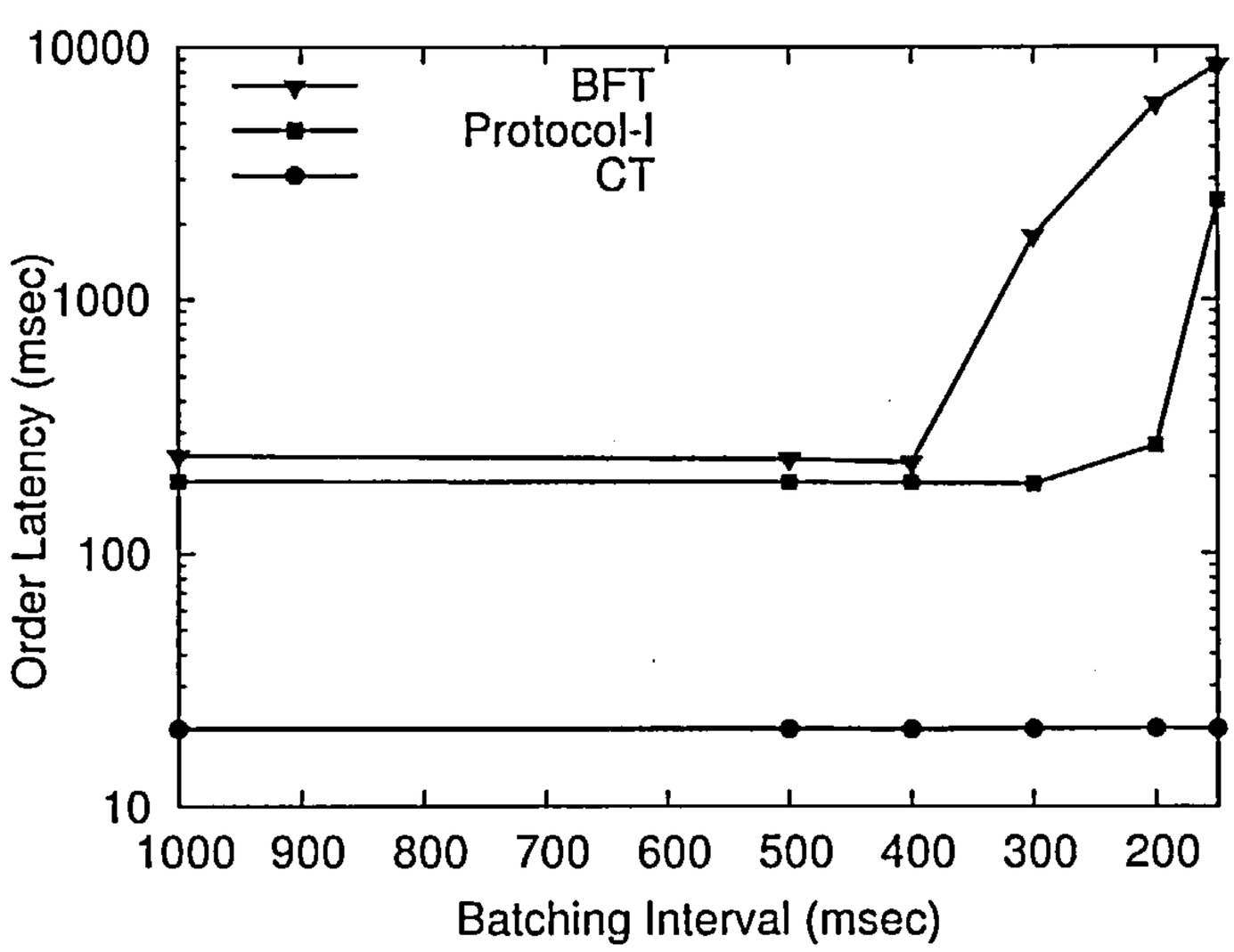
(b) MD5 with RSA key size 1536

(c) SHA1 with DSA key size 1024

Figure 4.6. Order latency at p_3 for $f = 2$ using various crypto techniques.



(a) MD5 with RSA key size 1024



(b) SHA1 with DSA key size 1024

Figure 4.7. Order latency at p_4 for $f = 3$ using various crypto techniques

Two important performance comparison parameters were found to be a) latency in steady state and b) threshold point from where system starts saturating. Hence to summarize the results and to present them in easily comparable format, order latencies of the selected processes in steady state region (average of all values acquired by running experiment with batching intervals in steady state region) and the value of the highest interval for which a protocol was found in saturation region is given below in tabular form (Table 4.1 and 4.2) for all different configurations.

f = 2	Steady State Order latency for p_3			
		Protocol-I	BFT	CT
	MD5RSA1024	52.1	80.7	25.9
	MD5RSA1536	137.6	144.5	25.9
	SHA1DSA1024	160.0	170.7	25.9
	Steady State Order latency for p_7			
		Protocol-I	BFT	CT
	MD5RSA1024	70.2	77.9	21.7
	MD5RSA1536	142.0	147.4	21.7
	SHA1DSA1024	168.4	174.4	21.7
	Threshold Batching Intervals for all processes			
		Protocol-I	BFT	CT
	MD5RSA1024	20	40	< 20
	MD5RSA1536	80	100	< 20
	SHA1DSA1024	100	150	< 20

Table 4.1. Steady-state order latency and Threshold Batching Intervals for f = 2.

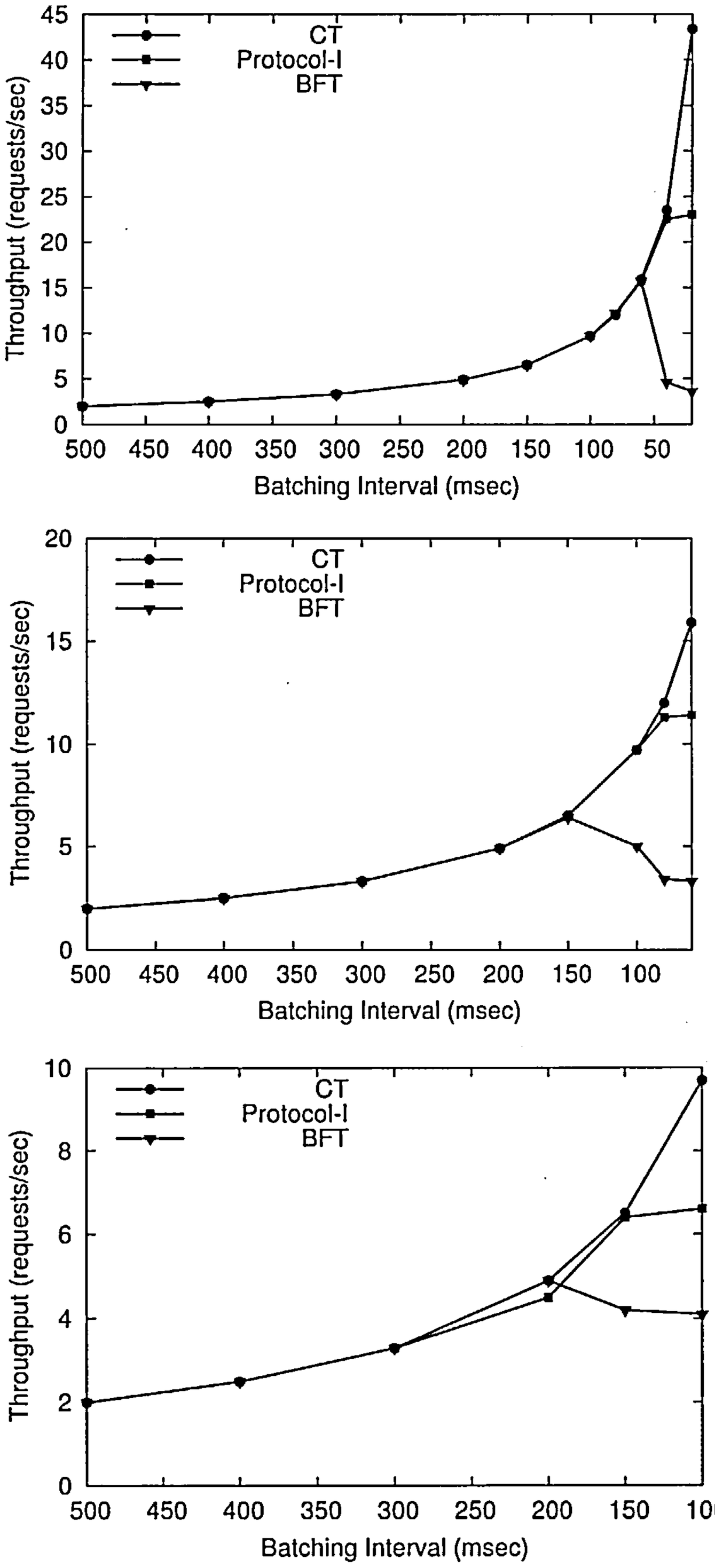
f = 3	Steady State Order latency for p_4			
		Protocol-I	BFT	CT
	MD5RSA1024	70.0	89.1	20.4
	SHA1DSA1024	190.6	236.0	20.4
	Steady State Order latency for p_{10}			
		Protocol-I	BFT	CT
	MD5RSA1024	79.1	98.2	31.0
	SHA1DSA1024	206.9	242.8	31.0
	Threshold Batching Intervals for all processes			
		Protocol-I	BFT	CT
	MD5RSA1024	40	60	< 40
	SHA1DSA1024	150	300	< 40

Table 4.2. Steady-state order latency and Threshold Batching Intervals for f = 3.

4.8.2 Throughput

Throughput was observed to be low for larger batching intervals for all the three crypto-techniques. Figure 4.8 shows that with decreasing batching intervals, throughput increases until the system reaches the saturation point after which it starts dropping down. This behaviour was observed for both Protocol-I and BFT whereas the drop could not be observed for CT for the range of batching intervals used.

Observing the behaviour of the three protocols for any of the crypto-techniques, the conclusion drawn about BFT above is confirmed that it causes system saturation earlier than the other two and throughput starts dropping immediately after entering saturation point which is found to stay stable for a while in case of Protocol-I. Moreover, getting to the saturation point earlier when DSA is used as compare to smaller thresholds for RSA depicts the expensive nature of DSA for these protocols. For example, for $f = 2$, RSA causes BFT to drop throughput when batching interval is reduced below 60msec and Protocol-I below 40msec whereas with DSA the same points are reached at 200msec for BFT and 150msec for Protocol-I.



(a) MD5 with RSA key size 1024

(b) MD5 with RSA key size 1536

(c) SHA1 with DSA key size 1024

Figure 4.8. Throughput at p_3 for $f = 2$ using various crypto techniques.

Following similar pattern, drop in throughput occurs early when number of processes is increased to 10 i.e. $f = 3$. Table 4.3 and 4.4 show varying trends of throughput for all three protocols in tabular form for p_3 when $f = 2$ and for p_4 when $f = 3$ respectively.

Batching Interval (msec)	Throughput (requests committed per second)								
	MD5withRSA1024			MD5withRSA1536			SHA1withDSA1024		
	Protocol-I	BFT	CT	Protocol-I	BFT	CT	Protocol-I	BFT	CT
500	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
400	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5
300	3.3	3.3	3.3	3.3	3.3	3.3	3.3	3.3	3.3
200	4.9	4.9	4.9	4.9	4.9	4.9	4.5	4.9	4.9
150	6.5	6.5	6.5	6.5	6.4	6.5	6.4	4.2	6.5
100	9.7	9.7	9.7	9.7	5.0	9.7	6.6	4.1	9.7
80	12.1	12.2	12.0	11.3	3.4	12.0			
60	15.7	15.7	15.9	11.4	3.3	15.9			
40	22.5	4.6	23.5						
20	23.0	3.6	43.4						

Table 4.3. Throughput for p_3 with various batching intervals when $f = 2$.

Batching Interval (msec)	Throughput (requests committed per second)					
	MD5withRSA1024			SHA1withDSA1024		
	Protocol-I	BFT	CT	Protocol-I	BFT	CT
500	2.0	2.0	2.0	2.0	2.0	2.0
400	2.5	2.5	2.5	2.5	2.5	2.5
300	3.3	3.3	3.3	3.3	2.4	3.3
200	4.9	4.9	4.9	4.8	3.2	4.9
150	6.5	6.5	6.5	4.2		
100	9.7	9.6	9.6	4.1		
80	12.1	12.1	12.1			
60	15.8	4.5	15.8			
40	7.8					

Table 4.4. Throughput for p_4 with various batching intervals when $f = 3$.

4.8.3 Summary of Observations for Study I

Following are the general observations made from all the results presented above.

- DSA algorithm is not suited to consensus protocols where messages are mostly multicast
- BFT tends to saturate the system faster than Protocol-I as load increases. CT is the slowest saturating protocol
- As number of processes in the system increase, performance degradation in BFT is more than that in Protocol-I

– Increasing tolerance level from Crash to Byzantine has a substantial cost in terms of both quantity and quality. Performance degradation was observed to be quite high from CT to BFT/Protocol-I even in LAN.

4.9 Performance Study II

After the extensive experimentation over LAN, taking into consideration the observations of Study I, RSA key size 1024 was found the most suitable and practical crypto algorithm for Byzantine fault-tolerant protocols and was chosen to be used for second set of experiments instead of all three techniques. Also, the consistent behaviour of CT was sufficient to observe the least effect it has of increasing *f* value; it gave an idea of how much a service can be affected if the tolerance level is increased from crash to Byzantine. Hence CT was dropped from the second set, which focuses only on comparing performances of the two Byzantine fault-tolerant protocols in a variety of network setups. Also fault-tolerance parameter (*f*) takes value of 1 and 2 for Performance Study II. The *batching_interval* is varied from 50 msec to 2500 msec, and the *batch_size* is fixed at 32 bytes.

Using the Internet to test performance of distributed applications on asynchronous network with various load and bandwidth settings is not a very feasible option. This is due to the very obvious reason of lack of control over the dynamics of the network. There are too many factors involved which could affect network performance, all of which are not under our control. Being able to maintain uniform environment is specially important when the tests are performed for comparison purposes. Moreover, the cost of setting up the hosting distributed environment in various configurations can be very high. Using an emulator is therefore considered a feasible option to avoid the above mentioned issues. The emulator is required to be able to emulate network traffic on a typical asynchronous network as precisely as possible, yet letting various parameters like network topology, bandwidth of each link, traffic load etc to be varied by the user to emulate certain setups.

The emulator built within the School, simply called WAN Emulator [AS07], was chosen to be used as the testing tool for Performance Study II. WAN Emulator was found best suited to the requirements. It requires no modifications to the underlying operating system, networking libraries etc. It provides a user-friendly GUI to set numerous parameters to desired values. This can even be done on-the-fly i.e. while the

emulation is being performed. Moreover WAN Emulator is capable of emulating network faults (loss, delay, corruption, reordering etc) and provides application-level software-implemented fault injection. This feature is highly relevant to our work and is planned to be used in future work. Also easy access of the source code and assistance from the locally available inventor are added advantages. The only hindrance in porting the implemented protocols directly to the WAN Emulator was that the latter is programmed for CORBA applications. Since CORBA communication model is more generic than RMI/sockets, it was considered worthy to amend the implementations to use CORBA for communication. Hence the implementation of the two protocols used in Study II replaces RMI and sockets with CORBA objects. Experiments in this study were conducted on the following settings

- Real LAN
- Emulated fast WAN
- Emulated slow WAN

For the sake of continuity, the following subsection briefly describes how the WAN Emulator works. Most of the text in this part is taken from [AS07, AS04]. Furthermore some of the parameters - only those relevant to this study, that the WAN Emulator allows to be changed, are listed with their corresponding values used for this experiment set.

4.9.1 WAN Emulator Service

The Wide Area Network Emulator for CORBA Applications is a system that attaches itself to an ORB: CORBA's Object Request Broker. All CORBA applications using the ORB's services can therefore be tested. The emulation is configurable and the emulator gives the ability to control every property of the network to be emulated.

A physical node of the emulated network is represented virtually by an instance of each of the following three together. (Shown in Fig 4.9.)

1. CORBA-based application to be tested,
2. WAN Emulator, and
3. Interceptor

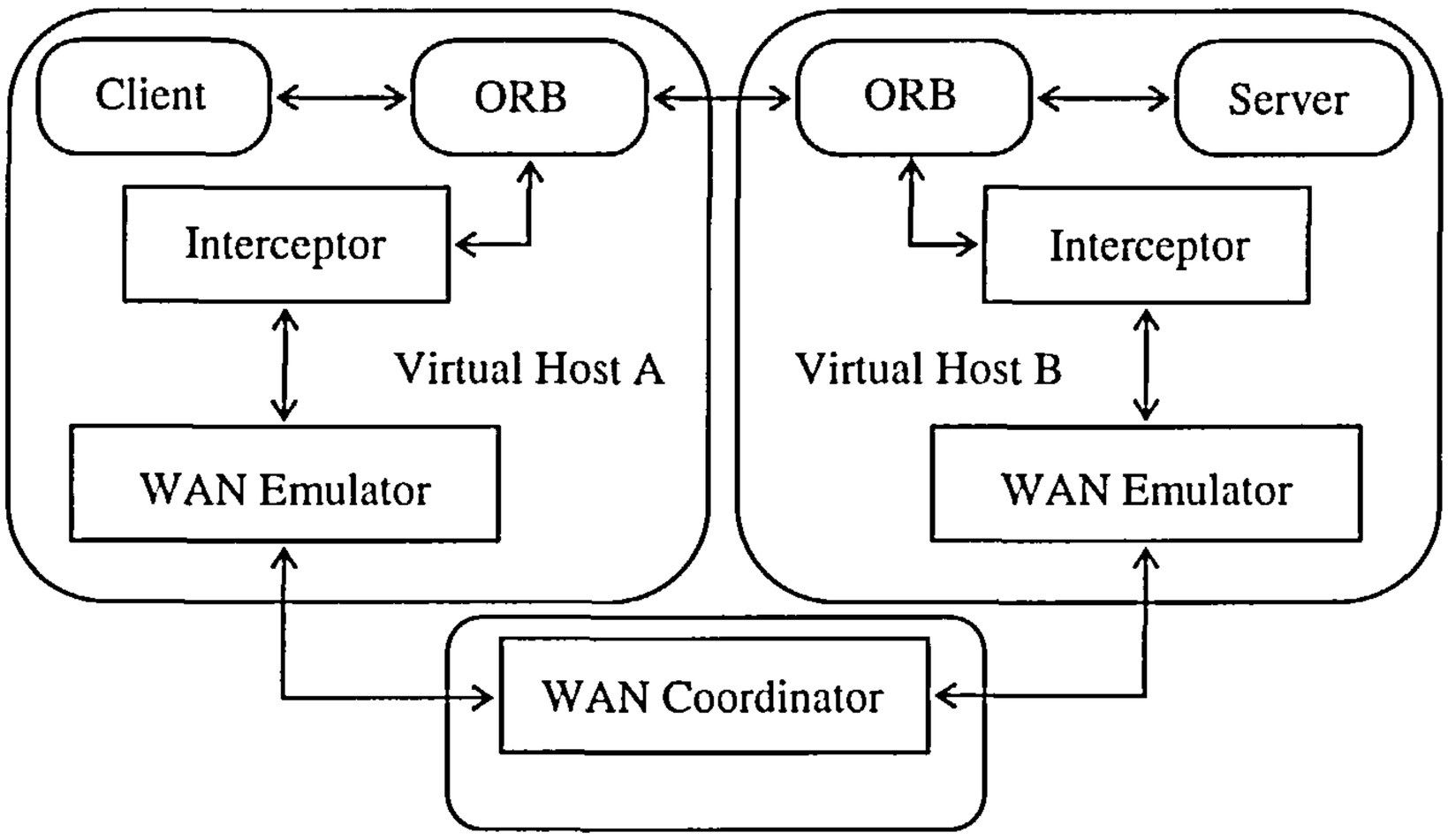


Figure 4.9: The WAN Emulator Architecture

Interceptor attaches itself to the application's ORB running on the host to capture CORBA traffic going through the ORB. These intercepted packets experience delays as if they are being transmitted over a WAN. This is done by the *WAN Emulator*. Specifically, each WAN Emulator generates the background traffic in the context of which the intercepted packets will be transmitted. Thereby, the later will be subjected to experiencing queuing and transmission delays and transmission losses even though the actual transmission will be over a LAN or even an inter-process communication (IPC) channel.

The three modules act together as a virtual host that emulate the node and its networking resources. Multiple virtual hosts can be run on the same physical machine or on machines distributed over a LAN.

A single system-wide *WAN Coordinator* facilitates interaction between WAN Emulators which are immediate neighbours in the network topology. The Coordinator is a CORBA service that can be run on any physical node in the system. It starts by loading the topology of the network to be emulated from a topology file and then waits for each WAN Emulator to register itself. It then acts as a controller of the emulation by issuing *Start* and *Stop* commands to the registered WAN Emulators. The Coordinator provides directory-like services to Emulators. More precisely, it provides the locations of CORBA objects and other Emulators.

A *WAN Emulator Service* is therefore composed of a WAN coordinator and a set of WAN Emulators plus corresponding interceptor components. The design of the emulator service assumes that the wide area networks to be emulated are large enough

that the overhead of the emulation is negligible compared to the actual communication delays.

4.9.2 Components of the WAN Emulator

A WAN emulator is composed of three components; emulation engine, trace file reader and traffic generator. Each emulator starts by registering with the Coordinator and then initializing a trace file reader, a traffic generator and the emulation engine (Fig 4.10), each described below. A WAN Emulator (emulation engine) provides emulation decisions to the interceptors and to other neighbouring emulation engines.

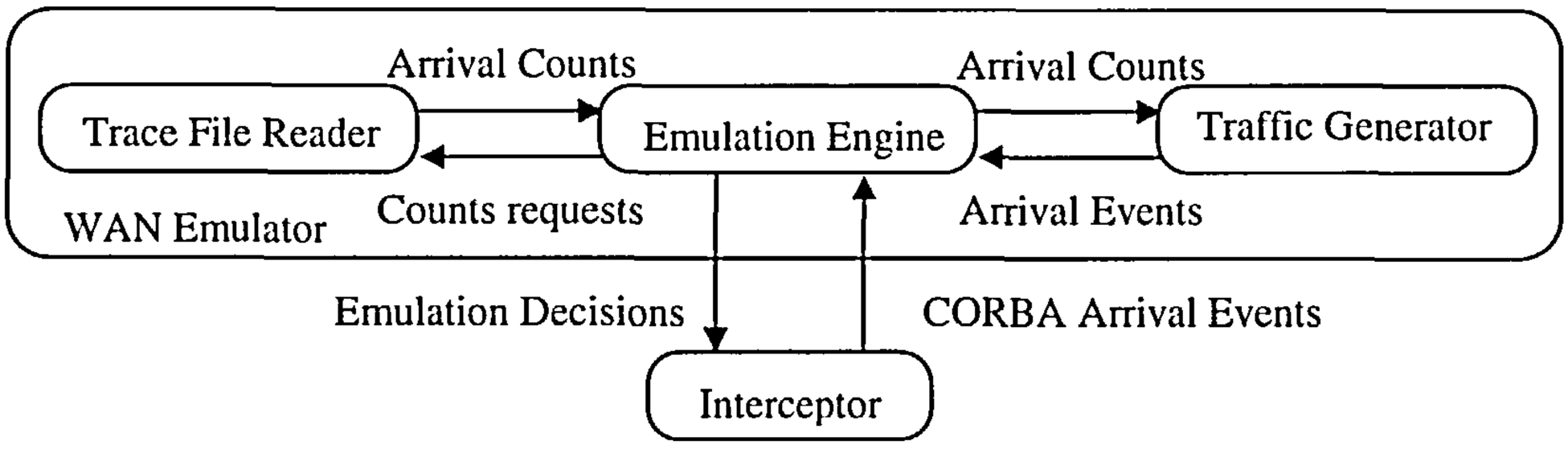


Figure 4.10. Three components of a WAN Emulator

Trace File Reader: It uses already captured traffic arrival count traces stored in the form of trace files. These traffic traces can be generated by using *Trace File Generator* facility provided by the emulator service, which supports both Poisson arrival and Self-Similar traffic patterns. Alternatively, traces can be captured from real networks or created manually. Trace file reader is probed after every (configurable) unit time by the emulation engine to provide arrival count for that unit time.

Traffic Generator: It is initialized by the emulation engine and generates network arrival events during a unit time based on the arrival count provided by the emulation engine for that unit time. Any arrival distribution policy can be used to distribute arrivals within a unit time and packet length distribution to generate packet arrival events of various sizes. For our work, self-similar arrival patterns were used with packet size traces captured from an IP backbone.

Emulation Engine: The emulation engine is at the core of the entire system. It co-ordinates the operations of the trace file reader and the traffic generator. It also maintains various buffers and counters to store various properties of all links connected to a node in the emulated network. These properties include error rate, packet drop rate, propagation delay, bandwidth, number of packets to be sent and their sizes etc. For each

unit time, it receives the arrival count from the trace file reader and invokes the traffic generator to get arrival pattern of packets and their sizes for that unit time and updates its buffers and counters based on the properties of each link. This counts for the background traffic generated to emulate WAN for a node. When the emulation engine receives a CORBA packet from Interceptor, it makes an emulation decision based on the emulated traffic load. This decision can also be to inject error in the packet or drop the packet. Since in this study we do not inject any errors and do not drop any packets, the decision only indicates the delay which this packet should suffer.

We describe below two parameters that are used in this study to create various emulation settings. These parameters along with many others are defined for every link between two nodes of the emulated network and can be set to required values in a topology file.

Propagation Delay - The propagation delay is the time (in milliseconds) that a bit takes to travel from one end of the network to the other end. This parameter can be set to values acquired straight from Ping or any other such tools.

Bandwidth – This refers to the maximum capacity of the link between any given two nodes of the emulated network, expressed as the number of bits the link can transmit per unit time. If there is a multi-link path between two ends, then the capacity of the bottleneck link should be considered.

Each Emulator (figure 4.10) communicates only with its immediate neighbours and the Coordinator to reach an emulation decision. An immediate neighbour is an emulator that emulates a node directly linked (via physical network link) to the node in the emulated network defined in topology file. The CORBA arrival events triggered by an interceptor could be handled immediately if the target object is hosted locally. If not, the emulation engine finds its location and following the route to the node hosting that object asks its neighbour to emulate the rest of the network for this request. Each emulator registers all CORBA objects it hosts with the Coordinator. In return, it receives configuration and commands from the Coordinator. It can also obtain the locations of CORBA objects that it does not host.

Role of Interceptor: Interceptor is the part of WAN emulator service that actually realizes the emulation decision made by corresponding Emulator. When a CORBA application sends a message, the ORB serving the application notifies the installed interceptor before transferring the message to its destination. The interceptor in turn contacts the emulator with which it is registered asking what to do with the packet. This

is shown as (a CORBA) *arrival event* in figure 4.10. After receiving emulation decision from the emulation engine it is responsible for delaying the messages.

4.9.3 Emulated Network Configurations

There are many ways to place the processes in a WAN and test each protocol's performance. When $f = 2$, for example, there are 7 processes in all, a process can be placed in a far away location and the other 6 processes in another location (connected via much faster links). This creates an imbalance in the delays and will affect the performance of the protocol. Similarly, 5 processes can be connected with fast links and the other two with much slower links, etc. To maintain continuity with Study I and to see what are the relative performance figures for similar asynchronous setups, this thesis only addresses homogeneous networks links. The two protocols are tested for two extreme cases: either all processes are linked via fast links or all processes are connected via slow links. Figure 4.11 shows the chosen configurations for both $f = 1$ and $f = 2$.

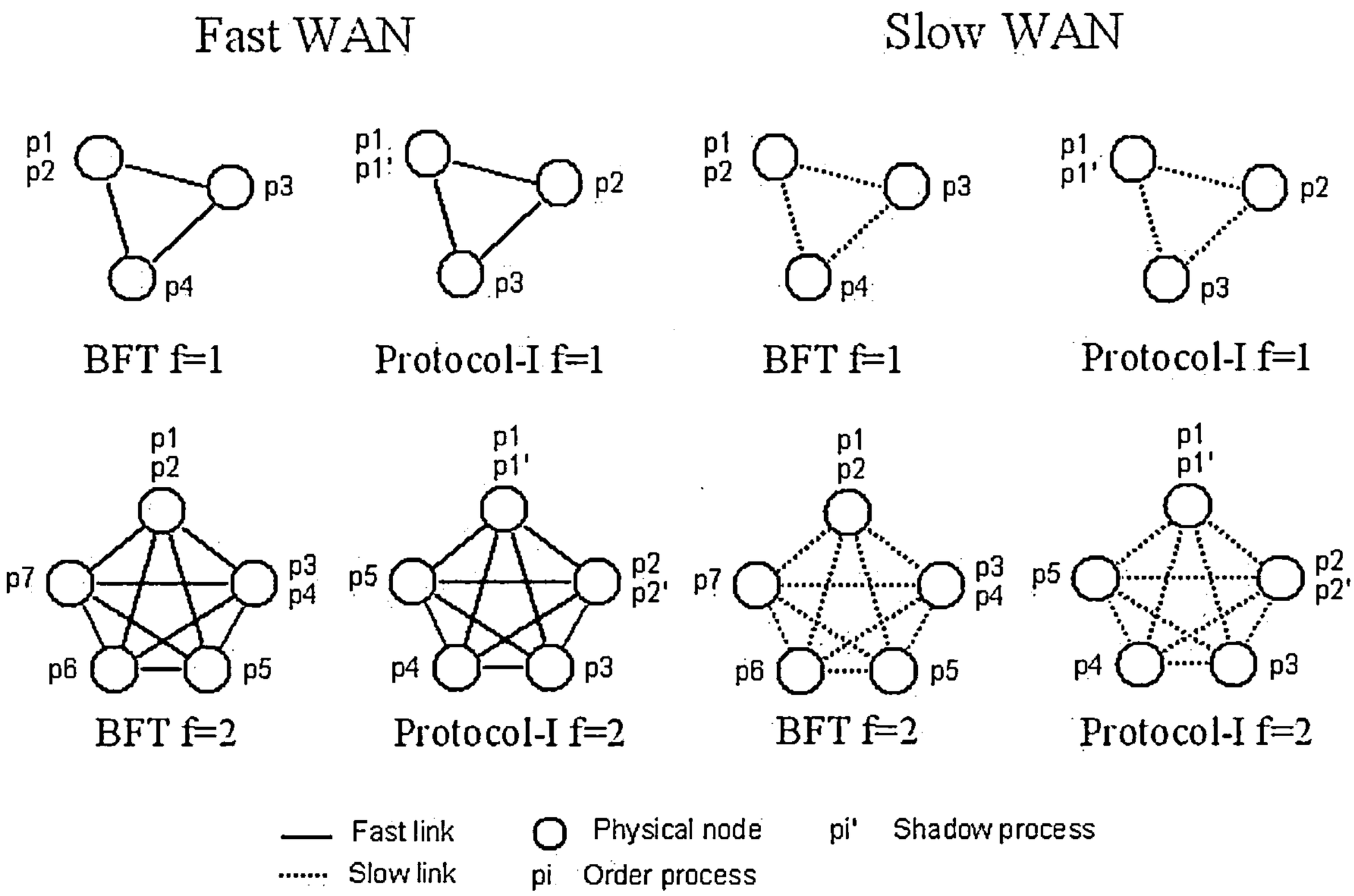


Figure 4.11. Emulated WAN Configurations

The purpose of this study is to analyze the role which the speed of asynchronous links plays when communication time between the paired processes is fixed to a small but constant value. Recall that the assumption made by Protocol-I that

paired processes are meant to be connected via fast Ethernet links. All paired processes were located on the same virtual host as if these were two processes running on one physical node in the emulated network. This has the same affect as being on a fast Ethernet as it is assumed that transmission over these links is almost instantaneous. While emulating these fast links, no delays were injected by the Emulator. Hence communication time between the paired processes is the time it takes for the emulator to intercept and inspect the packet. For fairness and uniformity, the respective processes in BFT were also hosted together even though they were not paired.

The circles in figure 4.11 represent the sites where the processes, p_i, (and their shadows, p'_i, in the case of Protocol-I) are located. The solid lines represent identical fast links whereas the dotted ones represent identical slow links. The same cluster, as used in Study I, was used to run this set of experiments except that each machine has Fedora Core 5 installed on it. For every circle in figure 4.11 a Linux machine in the cluster is used to host a WAN emulator service of section 4.9.1 and the process(es) assigned to that virtual host. WAN coordinator and client were run on separate machines.

For both of the network configurations following parameters were fixed to given values also mentioned below.

- Unit time was fixed to 1 second.
- Packet size distribution followed measurements taken from Sprint IP Backbone [FM03].
- Arrival events were evenly distributed over a unit time.
- Arrival count(s) was(were) generated by using self-similar traffic model as studies of network traffic show that it is self-similar in nature [CB97]. The mean packet rate was fixed to 30 packets/second on each link and self-similarity (or hurst) value to 0.8.

Following are details of the two network configurations that were emulated

Fast WAN Configuration.

This configuration represents a network with nodes spread across a country. The propagation delays are fixed to 2msec which is typical of inter-city links within the UK (e.g. between Newcastle and London). The bandwidth of each link is 1Mb/s. The average utilization of each link given this bandwidth and the traffic generated from the traces mentioned earlier ranges between 10% and 20%.

Slow WAN configuration.

This configuration represents the other extreme: all processes are located in far apart geographical locations and linked by slow links i.e. over an intercontinental network. The propagation delays are fixed to 50msec which is typical of far apart locations and links that span the oceans (e.g. between Newcastle, England and Riyadh, Saudi Arabia). The bandwidth of each link is 512Kb/s. The average utilization of each link given this bandwidth and the traffic generated from the traces mentioned earlier ranges between 20% and 40%.

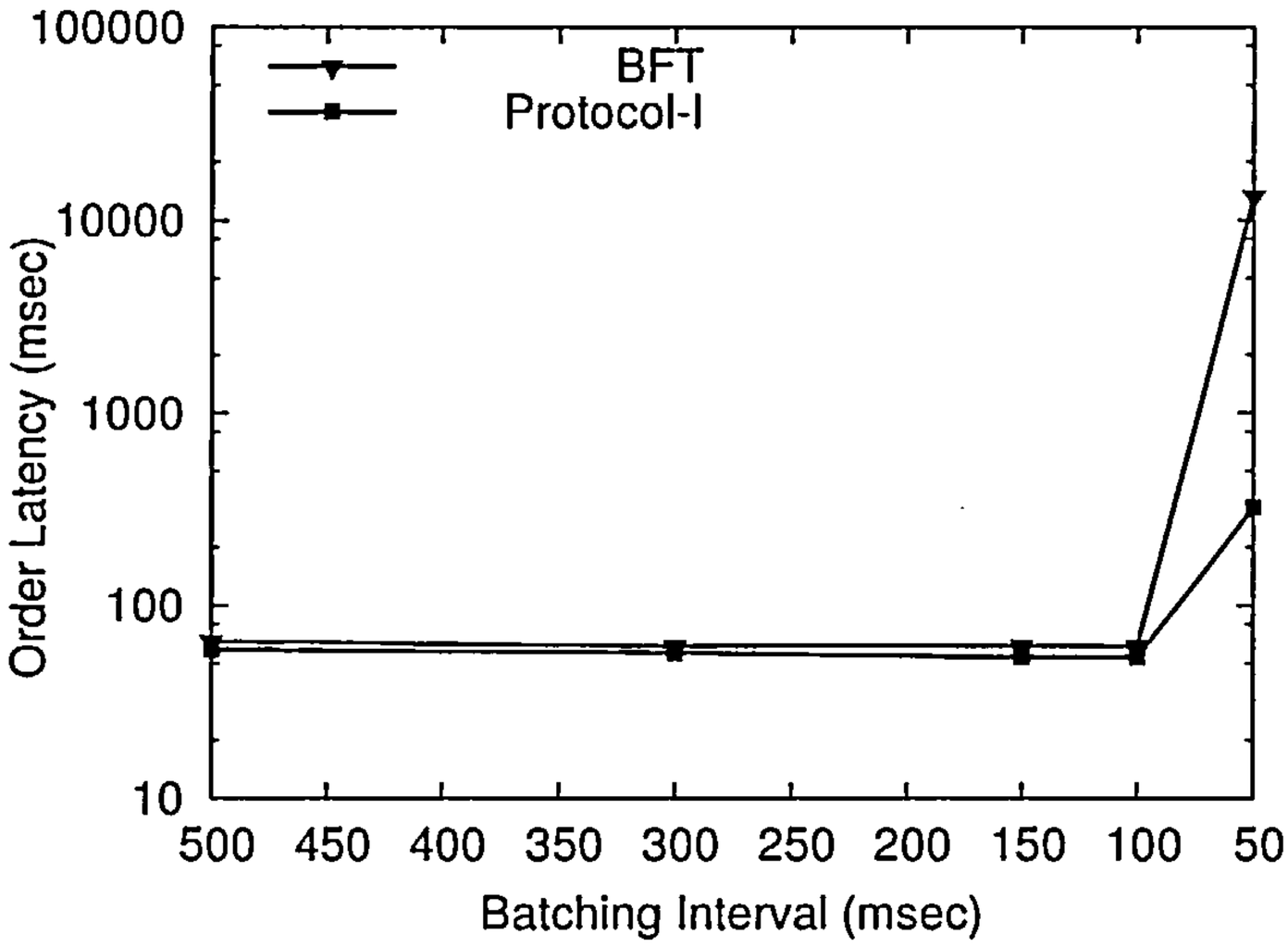
4.9.4 Experiment Results

This section presents the results from the experiments performed by running Protocol-I and BFT over the two emulated network settings described above. These will show the difference in performance of the two protocols when the network connections between some processes is faster than the others. For example, for Protocol-I, link between the constituent processes of an FS process is faster than that between two FS processes. In addition to comparative analysis of the two protocols over only one type of network, it was found interesting to be able to compare the performance of a protocol over different types of networks i.e. (real) LAN and (emulated) WAN. Although Study I captures the results from protocol runs over a real LAN but since the implementations here were modified to use CORBA communication model for Study II, it was important to redo some of the experiments with the new CORBA-compliant code. Hence LAN of the cluster described earlier was used without introducing other sources of traffic to measure the performance of the two protocols i.e. no emulation was used. For each case, $f = 1$ and $f = 2$, a single process is loaded on a machine utilizing 4 and 7 machines respectively. The client issuing the requests was loaded on another machine by itself. Each point in a graph is an average over 500 experimental results.

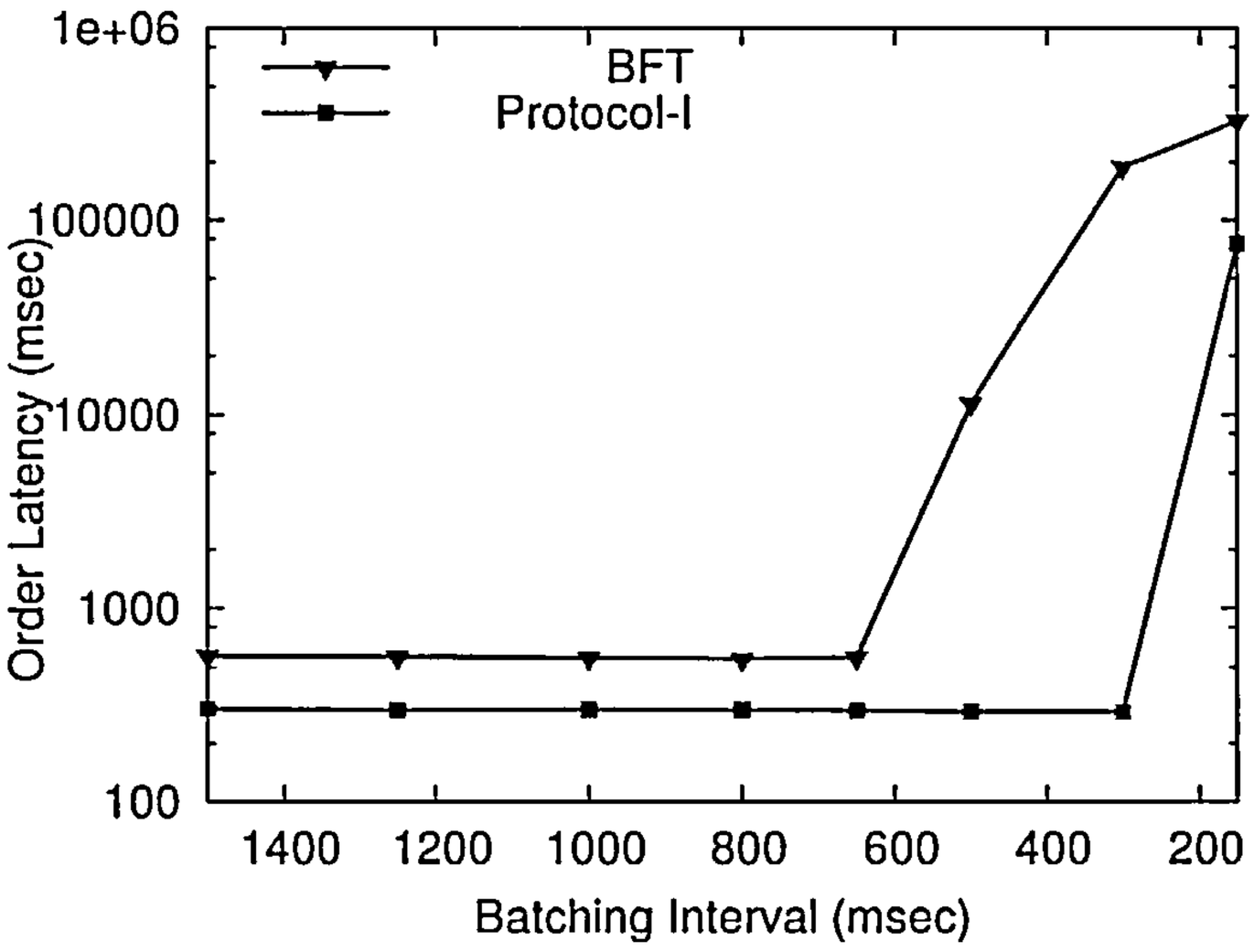
4.9.4.1 Order Latency

Order latency was observed to follow the same trend as found in study I. The values of the two protocols were found to be very close to each other with a difference of about 3 to 7 msec both with $f = 1$ and 2 using LAN setup. However it was interesting to see how the difference grows bigger as the network delays get longer. It was expected that Protocol-I will achieve major performance benefits over BFT in slower network settings because it makes use of the synchronous fast link within FS process. This makes

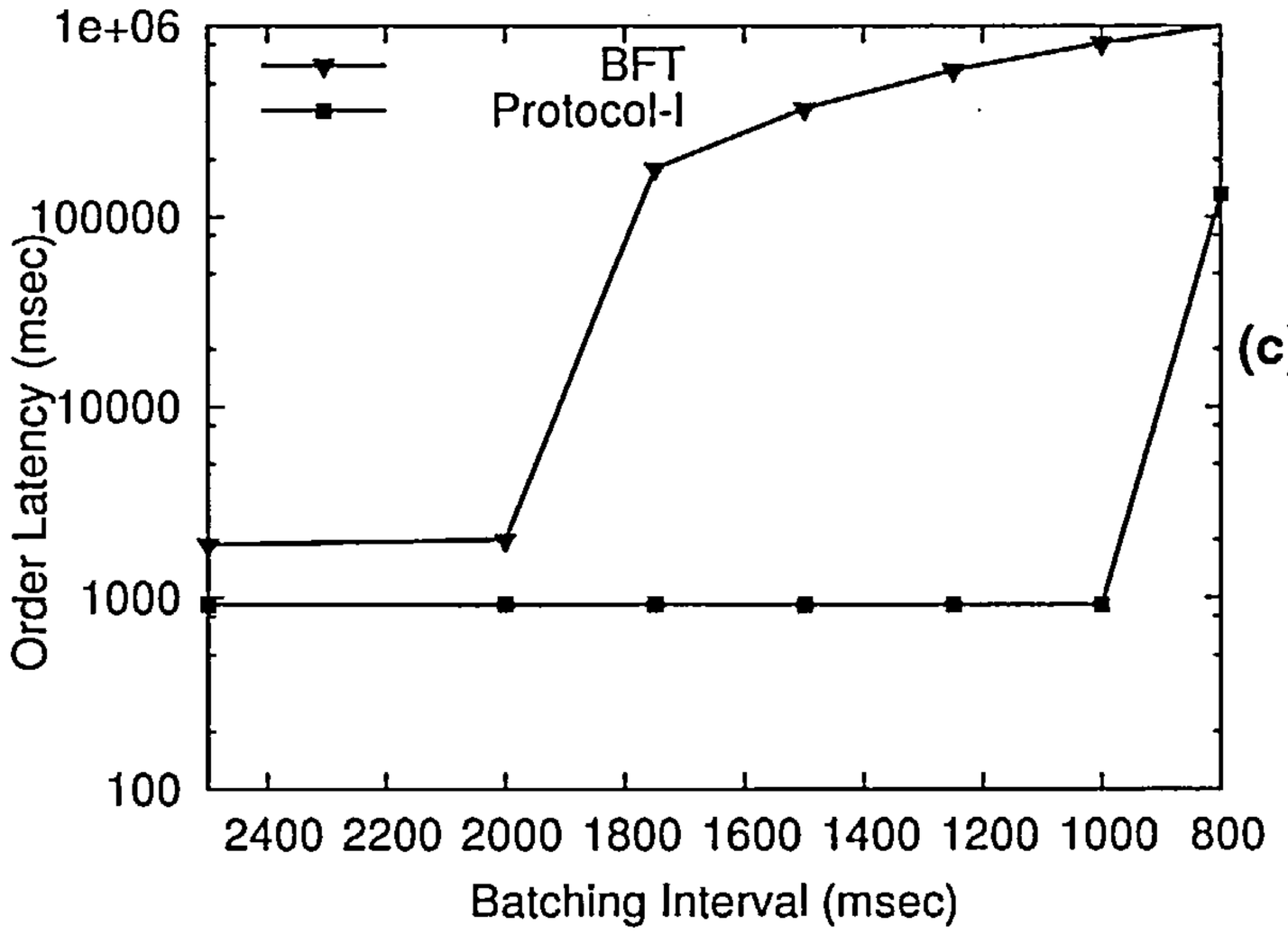
Protocol-I to transmit messages on slower links less number of times as compare to BFT. Whereas despite presence of similar fast links between processes of BFT, the difference in the performance of the two protocols was still expected to be substantial because BFT was not designed to make use of such a setup and hence involves more phases of communication over slower links. These predictions were found to be true and were confirmed by experiment results. The difference between the two protocol latency values grows to about 120 (100) msec for p_2 (p_4), $f = 1$ and 265 (240) msec for p_3 (p_7), $f = 2$ when fast WAN is used, with Protocol-I performing better in both cases (see Table 4.5 and 4.6). This difference grows to 486 (339) msec for p_2 (p_4), $f = 1$ and 1042 (869) msec for p_3 (p_7), $f = 2$ when slow WAN is used. As for the saturation point, BFT reaches saturation much earlier than Protocol-I as a slower network is used. For instance, as depicted by fig 4.12 which plots order latency observed at p_3 for f =2, BFT reaches saturation close to a batching interval of 500 msec whereas Protocol-I works in steady state until 150 msec for fast WAN configuration. For slow WAN setup, the difference becomes more pronounced with BFT saturating at 1750 msec as compare to 800 msec for Protocol-I.



(a) LAN



(b) Fast (Inter-city) WAN



(c) Slow (Inter-continental) WAN

Figure 4.12. Order latency at p_3 for $f = 2$ using various network configurations.

Steady state latency values along with saturation threshold batching interval values are summarized in Table 4.5 and 4.6 below.

f = 1	Steady State Order latency for p2		
		Protocol-I	BFT
	LAN	48.2	51.6
	Fast WAN	155.3	273.3
	Slow WAN	411.5	897.2
	Steady State Order latency for p4		
		Protocol-I	BFT
	LAN	40.7	45.2
	Fast WAN	137.5	237.9
	Slow WAN	350.8	689.5
	Threshold Batching Interval for all processes		
		Protocol-I	BFT
	LAN	< 50	50
	Fast WAN	100	150
	Slow WAN	300	800

Table 4.5. Steady-state order latency and Threshold Batching Intervals for $f = 1$.

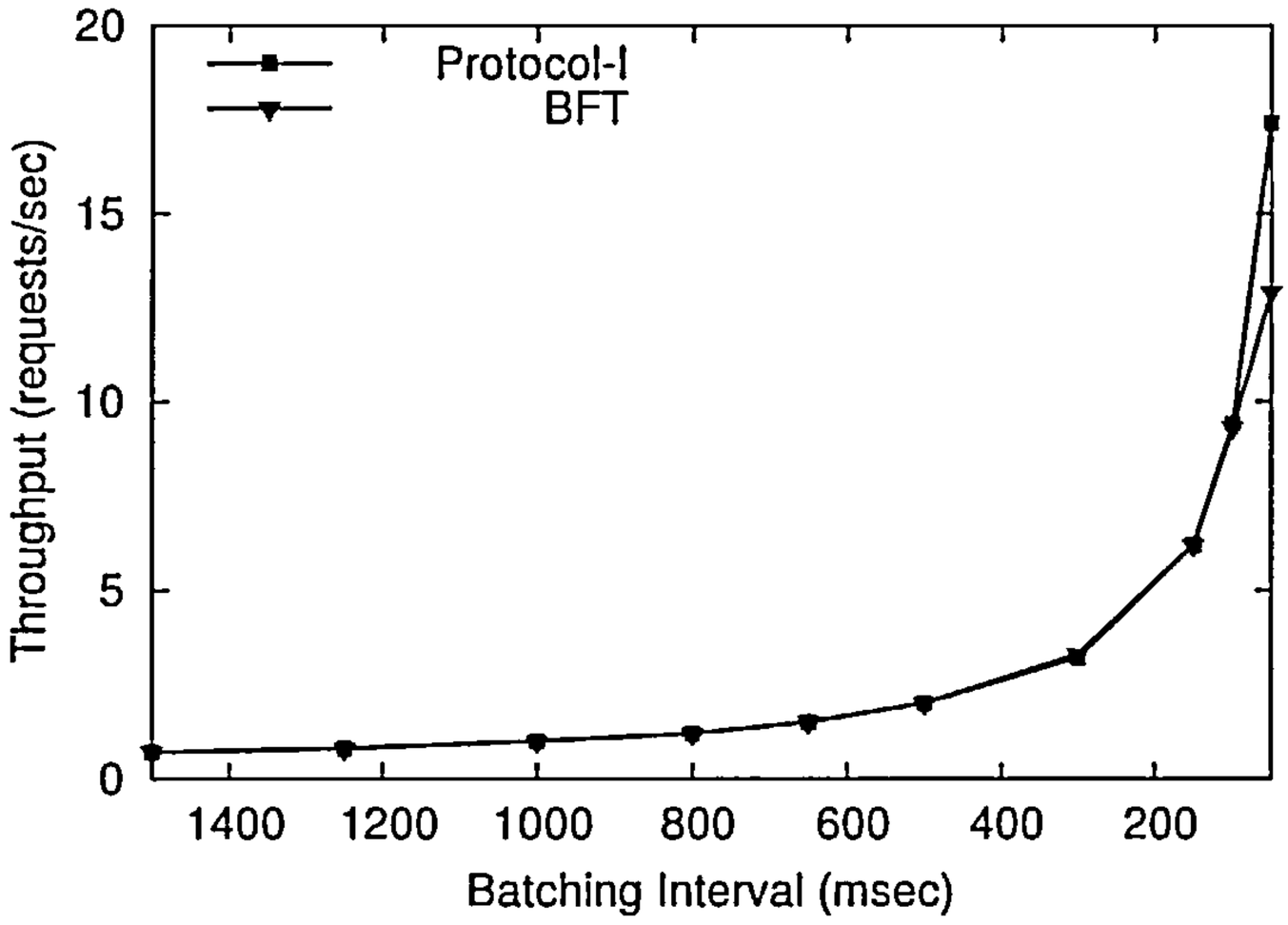
f = 2	Steady State Order latency for p3		
		Protocol-I	BFT
	LAN	61.0	67.2
	Fast WAN	300.7	565.2
	Slow WAN	917.8	1959.4
	Steady State Order latency for p7		
		Protocol-I	BFT
	LAN	65.7	72.8
	Fast WAN	351.0	588.7
	Slow WAN	1089.4	1958.4
	Threshold Batching Interval for all processes		
		Protocol-I	BFT
	LAN	< 50	50
	Fast WAN	150	500
	Slow WAN	800	1750

Table 4.6. Steady-state order latency and Threshold Batching Intervals for $f = 2$.

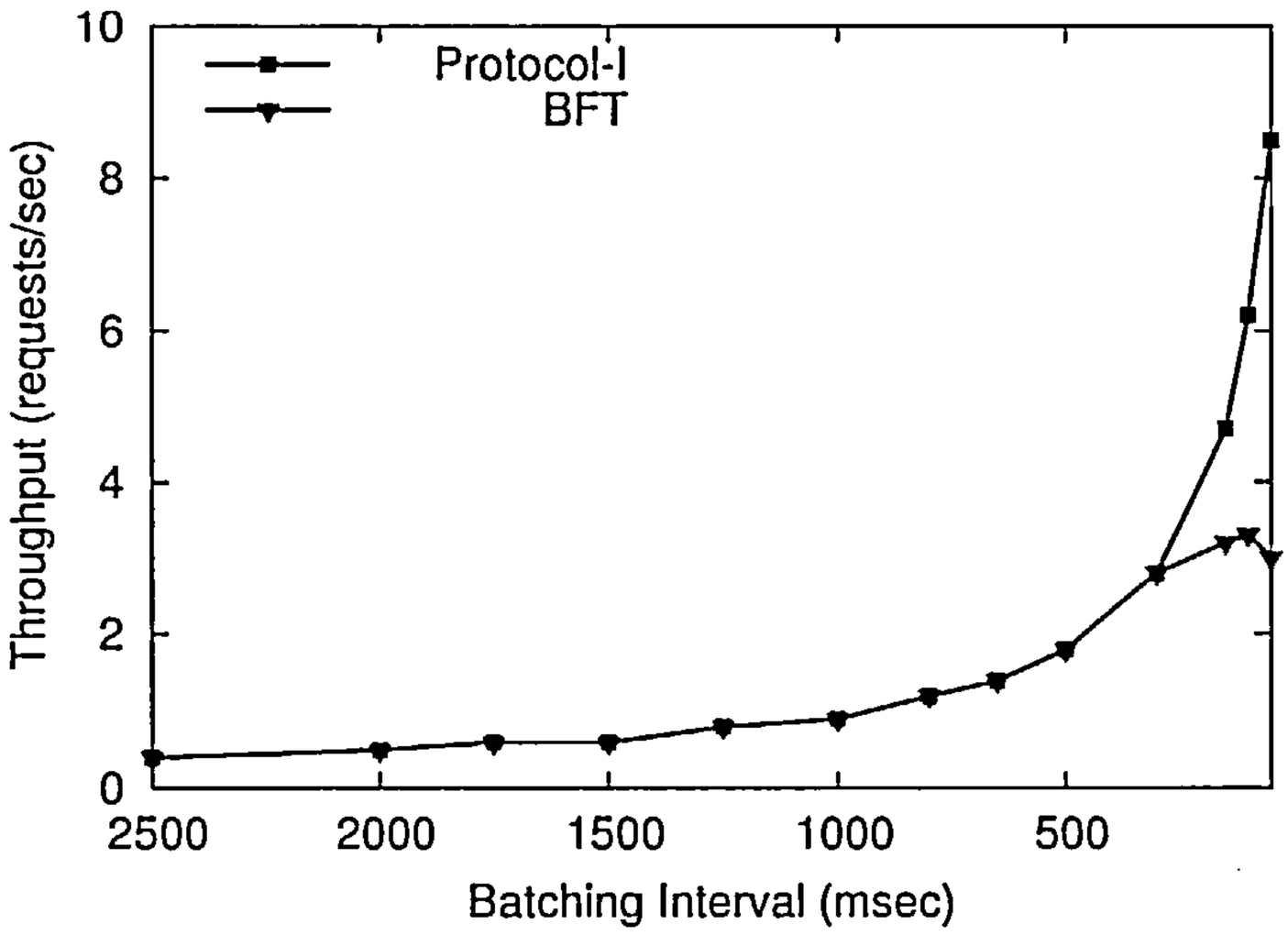
4.9.4.2 Throughput

Throughput also follows the same trend as observed in Study I. Performance decreases early i.e. at high batching intervals as slower network configurations are used. For instance, as can be seen from fig 4.13, when $f = 1$, throughput reached the highest value

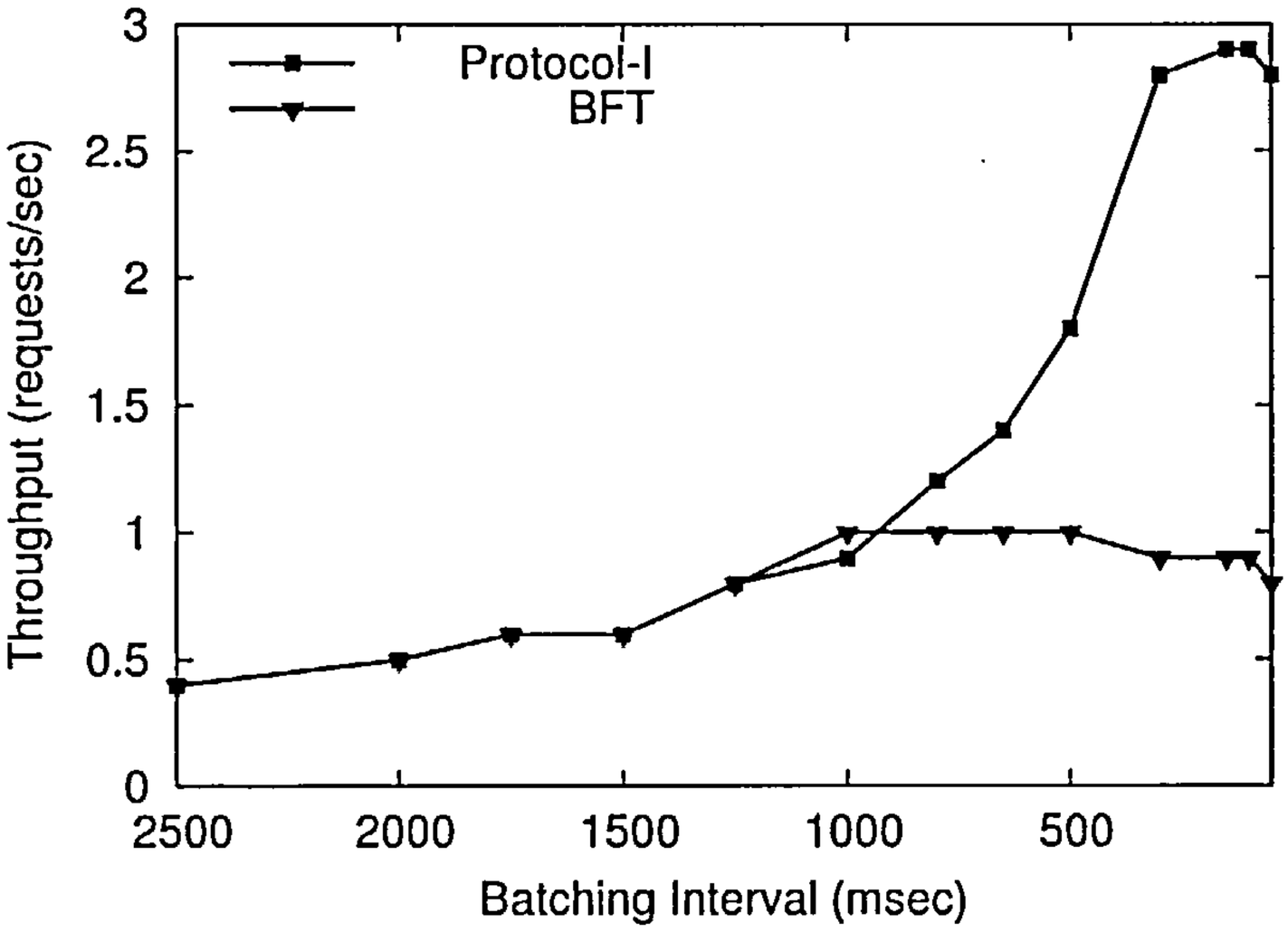
of 17.4 and 12.9 requests per second for Protocol-I and BFT respectively in LAN setting for the batching interval values used, while the highest values for the same were found to be 8.5 and 3.0 in fast WAN and 2.8 and 0.8 in slow WAN. Fig 4.13(a) shows that only the beginning of the performance slow down for BFT could be observed for LAN in the tested set of runs where the trend is still increasing for both protocols. Fast WAN plot (Fig 4.13(b)) depicts the throughput value at which BFT stabilizes (in saturation region) while Protocol-I showed increasing trend. Slow WAN readings show the stable values for both Protocol-I and BFT (Fig 4.13(c)).



(a) LAN



(b) Fast (Inter-city) WAN



(c) Slow (Inter-continental) WAN

Figure 4.13. Throughput at p_2 for $f = 1$ using various network configurations.

Exact slowing down trend can be seen in the Table below where throughput values are listed for each batching interval value for $f = 1$ and 2 in Table 4.7 and 4.8 respectively. Comparing highest achieved throughput values for the two WAN configurations, earlier observations are confirmed by the pronounced decrease in performance of the two protocols when number of processes increases - f changes from 1 to 2.

Batching Interval (msec)	Throughput (Orders committed per second)					
	LAN		Fast WAN		Slow WAN	
	Protocol-I	BFT	Protocol-I	BFT	Protocol-I	BFT
50	17.4	12.9	8.5	3.0	2.8	0.8
100	9.4	9.3	6.2	3.3	2.9	0.9
150	6.2	6.2	4.7	3.2	2.9	0.9
300	3.2	3.3	2.8	2.8	2.8	0.9
500	2.0	2.0	1.8	1.8	1.8	1.0
650	1.5	1.5	1.4	1.4	1.4	1.0
800	1.2	1.2	1.2	1.2	1.2	1.0
1000	1.0	1.0	0.9	0.9	0.9	1.0
1250	0.8	0.8	0.8	0.8	0.8	0.8
1500	0.7	0.7	0.6	0.6	0.6	0.6
1750			0.6	0.6	0.6	0.6
2000			0.5	0.5	0.5	0.5
2500			0.4	0.4	0.4	0.4

Table 4.7. Throughput at p_2 for various batching intervals for f = 1.

Batching Interval (msec)	Throughput (Orders committed per second)					
	LAN		Fast WAN		Slow WAN	
	Protocol-I	BFT	Protocol-I	BFT	Protocol-I	BFT
50	17.1	14.2	3.2	1.5	0.9	0.4
100	9.6	9.6	3.2	1.5	1.0	0.4
150	6.4	6.4	3.2	1.6	1.0	0.4
300	3.3	3.2	2.8	1.7	1.0	0.4
500	2.0	2.0	1.8	1.8	1.0	0.4
650	1.5	1.5	1.4	1.4	1.0	0.4
800	1.2	1.2	1.2	1.2	1.0	0.5
1000	1.0	1.0	0.9	0.9	0.9	0.5
1250	0.8	0.8	0.8	0.8	0.8	0.5
1500	0.7	0.7	0.6	0.6	0.6	0.5
1750			0.6	0.6	0.6	0.5
2000			0.5	0.5	0.5	0.5
2500			0.4	0.4	0.4	0.4

Table 4.8. Throughput at p_3 for various batching intervals for f = 2.

4.9.5 Summary of Observations for Study II

General observations from the results of Study II can be summarized as below.

- Observation made in Study I about BFT saturating the system at lighter load as compared to Protocol-I was confirmed in Study II.
- BFT performance degrades more than that of Protocol-I as number of processes in the system increases.
- Tremendous increase in the performance gap between BFT and Protocol-I was observed as network configuration was changed from LAN to inter-continental WAN i.e. when slower asynchronous links were used. Protocol-I was found to perform at least 45% better than BFT (in terms of latency) in the inter-continental WAN setup.

4.10 Sources of Random Delays

While inspecting the detailed logs of measured values during experiment runs, spikes of anomalous values were sometimes observed in various sets of readings. Although the effect of these randomly appearing high values becomes negligible when average is taken over hundreds of readings, this subsection investigates the possible causes of these anomalous additional delays.

There are two types of random delays involved in all the experiments conducted. The first one is the one which is present inherently in all runs and the second one, which is only present in emulated network experiments, is either injected deliberately or is inherent in the tools/techniques used for emulation.

Type 1: There can be two sources of this randomness

1. Java Thread Scheduling: Since each of the three protocols was programmed as a multithreaded application, each program run was using Java scheduler in the background to allocate processor time to all the running threads. Since all the threads were meant to work in parallel, they all were given same priority. With this setting in place, switching between threads is highly dependent on the algorithm implemented by the scheduler. Most of the JVM use time slicing algorithms. For an application like the ones tested in this thesis where almost every message needs to be multicast, it can never be said with certainty how long a message (ready to be multicast) will have to wait due to thread scheduling before and during the multicast operation.
2. Traffic load on the underlying LAN: Since the cluster used for all the experiments was not a dedicated LAN but instead a normal campus cluster, experiments

conducted at various times could have faced different network conditions like heavy traffic load due to a close coursework deadline or end of term time (exam time), light/no traffic load due to vacation time, random surge of traffic due to a student downloading some data using the cluster etc. To make sure all runs encounter almost similar conditions, if not exactly same, all experiments were run at late night/early morning time when traffic load was less likely to be high. Also it was made sure that at least the machines used for the experiments were not used by any other user. Despite these efforts, network conditions during all experiment runs can not be guaranteed to be exactly same.

Type 2: This type of randomness comes with the emulator and can be because of two factors

1. Java Thread Scheduling for emulator threads: Emulator is also programmed in java as a multithreaded application so the factor mentioned in source 1 of the first type of randomness applies to emulator as well.
2. Traffic load injected by emulator: This is the random delay injected to emulate random delays that are present in an asynchronous network like the Internet due to ever changing traffic. This is produced by injecting synthetic traffic by the emulator and depends upon arrival count and arrival packet size, taken from trace file in each unit time. Since these two parameters do not hold a constant value but follow the distributions used, the number of bits of synthetic traffic, waiting in sender's buffer to be placed on transmission line before the first bit of an application packet, is not constant. Hence each packet will suffer some random delay before its first bit is placed on the line.

To summarize, we denote the above mentioned factors as follows:

E_d = Time it takes for the emulator to make an emulation decision (almost constant and can be assumed negligible).

J_d = Waiting time as a ready thread for Java thread scheduler (Source 1, Type 1 and 2).

B_d = Waiting time in the buffer because of synthetic traffic (Source 2, Type 2 above).

P_d = Propagation delay of the emulated link involved (constant).

T_d = Transmission delay for the emulated link involved (constant, depending upon packet size and bandwidth).

L_d = Delay due to background traffic on the underlying LAN (Source 2, Type 1).

For emulated network configurations, the total delay a packet will experience from the point it is ready to be sent from application's (protocol's) perspective and the time it actually reaches the destination can be given by the following equation.

Total Delay = $E_d + J_d + B_d + P_d + T_d + L_d$

Whereas for simple LAN experiments it can be given as

Total Delay = $J_d + L_d$

Delays involved at receiving end are not considered here but it is assumed that once a packet arrives at receiver, it would only incur Java thread scheduling delay before it can be processed by the application.

4.11 Summary

This chapter presented the design and analysis of Normal part of Protocol-I. Protocol-I is an advanced total order protocol in the sense that it does not only accommodate the effects of failing state of FS process but also uses relatively less restricted set of assumptions. Here, the construction of FS process assumes the timing bounds to be accurate as before but allows both constituent processes to fail one after another. However, the two failures are assumed to be at least $2D$ time apart from each other, where D is the finite (but unknown) bound on communication delays between any two correct processes over the asynchronous network.

We showed that Normal part of Protocol-I is a three phase communication algorithm. Message communication in the first phase is only performed between the paired processes of the coordinator FS process. Qualitatively comparing it with the three phases of BFT, we showed that this first phase is the major difference between the two protocols which keeps the total number of messages communicated in Protocol-I at the lower end.

Normal parts of the two protocols were also compared quantitatively. We presented two performance studies based on results form two sets of experiments. First study also considers a simple crash-tolerant protocol, derived from Protocol-I, to evaluate the overhead in performance of the other two Byzantine fault-tolerant protocols due to higher tolerance level. We executed the three protocols on a LAN cluster and compared the latency values by varying parameters like batching interval, cryptographic techniques and fault-tolerance degree f. The results confirmed the qualitative analysis and showed that Protocol-I performs better than BFT. Also, it was noted that use of

encryption in Byzantine fault-tolerant protocols is the major bottleneck in performance, absence of which in crash-tolerant protocols lets the latter perform much better.

Second performance study demonstrates performance gains of Protocol-I in emulated WAN settings. WAN emulator of [AS07] was used to emulate two WAN configurations; inter-city WAN referred to as fast WAN and inter-continental WAN referred to as slow WAN. Performance of Protocol-I was found to be substantially better than BFT in slow WAN settings.

Chapter 5

Protocol-I: Install Part

This chapter presents Install part of Protocol-I. Every process in the system executes Install when it receives fail-signal from the coordinator. Hence, the system moves into the next configuration. Install is the most complicated part of Protocol-I as it deals with the possible Byzantine behaviour of the signalled FS processes. The aim is to transfer the system from current configuration to the next appropriate one in a live and safe manner.

The chapter starts by listing the objectives of Install. Then it highlights the problems that need to be overcome to achieve the objectives. The discussion illustrates that a faulty 4-state FS process may not just behave in a benign two-facing manner but is also capable of exhibiting malicious behaviour. The initial discussion also touches briefly the possible solution strategies.

Algorithm steps are covered in detail in sections 5.4 - 5.10. These sections include the description of message structures used and discussion on the rational behind each step of the algorithm. Section 5.11 presents the performance study. We first compare Install of Protocol-I with that of BFT in a qualitative manner. The comparison shows that Protocol-I is more expensive in terms of number of communication steps. This is mainly attributed to the inclusion of Byzantine state in the FS process model. Results from experiments conducted on two emulated WAN configurations are then presented which confirm the qualitative analysis.

Finally an optimized version of Install named Install-II is proposed. A discussion on comparison of Install-II with Install is also presented at the end.

5.1 Introduction

Any correct process running Normal part of Protocol-I switches the execution to Install part on receiving fail-signal from the current coordinator. Install part will ensure that the next eligible process is installed as the new coordinator. Once installation is complete, execution is switched back to Normal part. In short, Install part of the protocol is responsible for transition from one configuration to the next appropriate configuration.

Notations P_c and P_{c+} are used throughout this chapter to denote signalling coordinator process and the next eligible coordinator, respectively; for brevity, the next eligible coordinator is considered to be an FS process throughout this chapter and the case for a non-FS eligible process is separately discussed. Also P_{c-} is used to represent the predecessor coordinator of P_c.

5.1.1 The Big Picture

Install part is more complex than Normal part. This is true for both Protocol-0 and Protocol-I. However, Install of Protocol-I is particularly sophisticated due to its handling of four-state FS processes with failing and Byzantine states (subsection 3.6.2.3). Its principles can be understood rather easily by imagining for the time being a special client who is continuously observing the system. We call this special client *observer client* and we will suppose that it can observe every event occurring in the system (like an omniscient observer), except that it may not know with certainty of the orders committed by correct processes immediately after the coordinator P_c has fail-signalled.

Suppose that when coordinator P_c fail-signals, the observer client prepares a signed message called *START* and multicasts it to all processes in the system. Note that when a message is signed by a client, it cannot be undetectably corrupted by the adversary. *START* comprises of a list called *TransferHistory* which contains all *ORDER* messages that were issued by P_c in his regime and have order numbers in the range $[CC_{mx}, PC_{mx}]$, where CC_{mx} and PC_{mx} are two integers defined in the following way.

CC_{mx} is the largest o that the observer client knows to be *certainly* committed by some correct process.

PC_{mx} is the largest o that the observer client knows to be *possibly* committed by some correct process. That is, it is known with certainty that the order number (PC_{mx}+1) or above is *certainly not* committed by any correct process.

The eligible coordinator P_{c+} treats this *START* message as a client request and orders it with one restriction that the order number o assigned to *START* must be $PC_{mx}+1 = start\_o$ (say). Note that this restriction allows any process to issue an *ORDER* message for *START* since *start_o* is fixed and is only a function of the unforgeable contents of *START*. Install part allows (initially) only the eligible P_{c+} to multicast *ORDER*(*start_o*) for *START*. Once a correct process commits $ORDER_{c+}$(*start_o*), it starts executing Normal part with P_{c+} as the (new) coordinator. That is, P_{c+} is installed

by that correct process and Install part completes successfully once all correct ones have installed P_{c+}.

Of course, P_{c+} may fail before any or all of the correct process can commit $ORDER_{c+}(start\_o)$. The observer client is assumed to know this and does not issue a new *START* but leaves it to the next eligible process to figure this out. That is, the next eligible needs to figure out whether to *re-use* the existing *START* or to expect a new, tailor-made *START* from the observer client. This problem is addressed as "Identifying Source Number" in section 5.8.

In reality, there is no observer client observing the system activities. Preparation of *START* needs to be done by the eligible coordinator itself. Furthermore, in the presence of assumption 2A (section 4.1), *START* needs to be made incorruptible like the client request. For this purpose, the eligible coordinator has the double-signed *START* it prepares additionally signed by $(f_c - 1)$ processes.

The major challenge is for the constituent processes of the eligible coordinator FS process to compute *TransferHistory*, have it agreed-upon between themselves and generate double-signed *START*. This is discussed at length in section 5.7.

The over all algorithm for Install part is outlined in section 5.4 and details given in section 5.6. Next, we state the objectives that Install aims to achieve and the challenges that the eligible process faces in achieving these.

5.2 Objectives

Suppose that current coordinator P_c has fail-signalled and *eligible*() = *c+*. The overall objective for the next coordinator P_{c+} is to compute a starting order-number, termed as $start\_o_{c+}$, with which it can begin its ordering regime. $start\_o_{c+}$ will have the following two properties.

1. It will be larger than any *o* of *ORDER*(*o*) which has become committed at some correct process, and
2. It will also be small enough to ensure that order-numbers assigned to client requests tend to be sequential as if coordinator change never occurred.

First property helps the protocol satisfy safety requirement i.e., it ensures that P_{c+} does not re-use any *o* of *ORDER*(*o*, *r*) that has been committed at some correct process, while the second attempts at providing continuity in order numbers issued by each successive coordinator.

Install part achieves the objective by helping P_{c+} to

a) Identify CC_{mx} and PC_{mx} (as described above).

b) Construct *TransferHistory* containing all $ORDER(o)$, $CC_{mx} \leq o \leq PC_{mx}$.

The starting order number, $start\_o_{c+}$ is simply computed to be one more than the largest o in *TransferHistory*.

Computation of *TransferHistory* by P_{c+} is not as easy as it may appear. In the next section, we present the problems that P_{c+} faces in achieving its objective.

5.3 Problems

We present below a list of problems that are encountered by P_{c+} and the way each of them is resolved.

5.3.1 Consultation with other processes

P_{c+} cannot compute *TransferHistory* on its own as it can never be sure that it has received all messages ever multicast by P_c to any correct process in the system. This is because P_{c+} cannot know whether P_c was in failing state when it received fail-signal from P_c and if so, for how long. Also, due to the communication system being asynchronous, there is no known finite amount of time until which P_{c+} could wait to ascertain it has received all messages ever sent to it (see uncertainties 1 and 2 in subsection 3.1).

P_{c+} therefore seeks information related to *TransferHistory* by requesting the history of acknowledged order messages from every acceptor p_i in the system. We denote this history of acknowledged messages as $AckHistory_i$ which is sent as a part of $STATUS_i$ message similar to the way $OrderHistory_i$ is sent in Protocol-0 (chapter 3, subsection 3.4.4). For reasons of liveness, P_{c+} should only be made to wait to receive responses from a quorum of Q_c processes ($Q_c = \lfloor (n_c + f_c) / 2 \rfloor + 1$), unless it can ascertain that all those responded so far cannot be correct. It therefore waits for at least Q_c responses and compute *TransferHistory* from the information gathered.

5.3.2 Dealing with Incomplete *AckHistory*

One major difference between the solicitation process of Protocol-0 and Protocol-I is that in the latter, all non-coordinator FS processes work in passive mode and hence the constituent processes produce *AckHistory* independently. Therefore, when $AckHistory_i$ is sent by p_i to P_{c+}, it has not been verified by the counterpart process p'_i within the FS

process P_i. In this scenario, if p_i is faulty, like any other unpaired faulty process, it can conceal the fact that it had acknowledged an *ORDER* message sent by P_c during the normal run and hence can undetectably present an incomplete message history to P_{c+} in Install part.

Intersection property of quorums comes handy to P_{c+} to resolve this issue. Let q_1 be a quorum which participated in commitment of *ORDER*(o) at a correct process. Let q_2 be the quorum which sent *AckHistory*s to P_{c+}. Even if, in the worst case, q_1 and q_2 have f_c faulty processes in common which are masking *ORDER*(o) from P_{c+}, the intersection property of quorums guarantees that there will be at least one common correct process in the two quorums which will send *ORDER*(o) to P_{c+}. But this brings with it the following caveat:

Caveat: The presence of even a *single* $AckHistory_i$ (amongst all Q_c *STATUS* messages) containing a given *ORDER*(o) must be taken as an indication that *ORDER*(o) might have become committed at some correct replica.

5.3.3 Dealing with Planted *AckHistory*

Assumption 2A (Section 4.1) allows the fail-signalled P_c to become Byzantine faulty with the passage of time. If that happens, P_c cannot be ruled out from 'planting' seemingly-correct *ORDER* messages and thereby attempting to, say, redundantly order a given (probably garbage-collected) client request. A planted *ORDER* message can introduce the following additional problem: while P_{c+} is waiting for Q_c *AckHistory*s, a Byzantine faulty process p_i can include the planted *ORDER* in $AckHistory_i$ it sends to P_{c+} as if it had acknowledged such plant while P_c was still acting as the coordinator. In this way, the faulty process is not only capable of providing an incomplete but also a misleading $AckHistory_i$ to P_{c+}.

Plants can be one of two types depending upon which field of an *ORDER*(o, r) is being re-used by the faulty coordinator. Suppose that *ORDER*(o, r) was planted by Byzantine faulty P_c and included by a faulty process p_i in its $AckHistory_i$.

1. *ORDER*(o, r) is called a *spurious plant* if there exists an *ORDER*(o', r), $o' < o$. (r is re-used),
2. *ORDER*(o, r) is called a *conflicting plant* if there exists an *ORDER*(o, r'), $r' \neq r$ (o is re-used).

All *spurious plants* need to be identified and discarded by P_{c+}. Discarding a spurious plant is always safe as a spurious plant cannot have been committed at any

correct process. This is due to the $2D$ time restriction in assumption 2A: consider that *ORDER*(o, r) sent by P_c is a spurious plant, which means both p_c and p'_c have failed and, by assumption 2A, at least $2D$ time has elapsed subsequent to the first of these failures has been observed by the second to fail. The first D of this $2D$ time is enough for the *fail-signal*, multicast by the first correct process after observing the failure, to reach all correct processes. (The significance of the second D will become obvious later). This means that at least every correct process knows about P_c's failure before P_c gets the ability to produce *ORDER*(o, r). Hence, no correct process will acknowledge this plant. However, only at most f_c faulty processes can send *ACK*(o, r). But since this is not sufficient for commitment, *ORDER*(o, r) can never get committed at any correct process.

For the case of a *conflicting plant*, we note that by definition, if *ORDER*(o, r) is a conflicting plant against *ORDER*(o, r'), the same can be said for *ORDER*(o, r') against *ORDER*(o, r). Hence, P_{c+} needs to carefully inspect all *ORDER*s that conflict against each other. P_{c+}'s objective is to make sure that it identifies and retains that order message, if any, which has been committed at some correct replica among conflicting plants.

P_{c+} uses the following fact to filter the planted messages discussed above. Since there can only be at most f_c faulty replicas in the system at any given moment, the planted *ORDER*(o, r) cannot appear more than f_c times in any set of *AckHistory*s from a quorum. This implies that any *ORDER*(o, r) that appears less than (f_c+1) times in a set of Q_c *AckHistory*s, needs to be treated carefully; on the other hand, if an *ORDER*(o, r) is found to have been included in more than f_c *AckHistory*s, then it certainly cannot be a plant.

A. Conflicting plants

A conflicting plant can find itself in one of two situations.

1. Against a committed *ORDER*: Let us assume the presence of a plant ⟨*ORDER*, c, o, $D(r)$⟩, and also of the committed ⟨*ORDER*, c-, o, $D(r')$⟩, $r \neq r'$ in a set of Q_c *AckHistory*s received by P_{c+}. Let us consider the situation where $c = c$-. Since both order messages are doubly-signed and authentic, then both p_c and p'_c have failed and, by assumption 2A, at least $2D$ time has elapsed subsequent to the first of these failures has been observed. First D has already been described. Second D is the time needed for a correct process to receive *AckHistory*s sent by other correct processes. So, in the above scenario, if,

⟨*ORDER*, *c*, *o*, *D*(*r′*)⟩, is committed by some process, then before receiving a plant, a correct p_{c+} will already have *AckHistory*s from all correct processes with at least (f_c+1) of them having included ⟨*ORDER*, *c*, *o*, *D*(*r′*)⟩. However, only at most f_c faulty processes could include ⟨*ORDER*, *c*, *o*, *D*(*r*)⟩ in their *AckHistory*s. In this way p_{c+} manages to achieve the objective by choosing ⟨*ORDER*, *c*, *o*, *D*(*r′*)⟩. As for the other situation where $c \neq c$-, the install part of the protocol is designed in such a way that this situation does not occur, i.e. it is ensured that each upcoming coordinator has to deal with order messages produced by only *one* of the preceding coordinators. Details of how this is done are given in subsection 5.10.1.

2. Against an uncommitted *ORDER*: Here the uncommitted *ORDER*(*o*) is not guaranteed to be present in more than f_c *AckHistory*s, which makes it difficult to identify the plant. In such a situation, picking either of the two to be included in *TransferHistory* is safe as neither of them could have been committed at any correct process. Hence install part lets P_{c+} do the selection randomly (random selection is enforced by p_{c+} and followed by p'_{c+} within the FS process).

Summarizing the conflict handling mechanism, presented below is the precise procedure followed by P_{c+} when a conflict is encountered; *ORDER*(*o*, *r*) and *ORDER*(*o*, *r′*) are found in a set of Q_c *AckHistory*s.

1. If one of the conflicting *ORDER*s, say *ORDER*(*o*, *r*), is present in more than f_c *AckHistory*s, choose *ORDER*(*o*, *r*). Note that *ORDER*(*o*, *r*) is chosen on the grounds that it is not a plant. Although it may not have been committed at any correct process, due to safety reasons, it is considered as potentially committed.
2. If none of the conflicting *ORDER*s is present in more than f_c *AckHistory*s, choose any, say *ORDER*(*o*, *r*).

ORDER(*o*, *r*) is referred to as the *conflict-resolved order* in both cases above.

B. Spurious plants

The clients' sequence-numbering of their requests is used by P_{c+} to identify spurious plants. As mentioned in section 1.4 (chapter 1), every client cl_i sequentially numbers its requests and tags each request message with the assigned sequence number r_i. Moreover, every request message is signed by the client and clients are assumed to be non-faulty. Therefore, this sequence-numbering issued by a client cannot be undetectably modified and is used to detect spurious plants. Recall that order number assignment and order acknowledgment is performed in-sequence by all processes. This

in-sequence acknowledgement is with respect to both order number o and request number r_i (see conditions iii and iv of Acknowledge process in subsection 4.4.2). Hence, for two order messages $ORDER(o, r)$ and $ORDER(o', r')$, if r and r' are from same client cl_i with request numbers r_i and r'_i respectively then $o' < o$ iff $r'_i < r_i$. This ensures that for a spurious plant $ORDER(o, r)$, if there exists an $ORDER(o', r)$, $o' < o$, P_{c+} could know about the re-use of r. To identify spurious plants, every process p_j in the system maintains a vector *commit_count*, where *commit_count*[i] = the largest sequence number of request from client cl_i that has been committed at any given moment. By definition of *commit_count*[i], it is clear that this count can only be incremented and is updated once a higher request is committed. *commit_count*[i] is initialized to 0.

The next section presents an overview of the way Install part uses these approaches to achieve the objective.

5.4 Outline of Install Part

We present the Install protocol in terms of the macro steps involved in its execution. Install is divided into five phases of communication as shown in figure 5.1. The figure assumes $P_c = P_1$ and *eligible*() = 2. Hence it represents configuration switch from Σ_1 to Σ_2. However, description below is given in generic terms.

Phase I – Send $STATUS_i$: First phase begins at process $p_i \in \pi \cup \pi'$ when p_i receives a fail-signal from an FS process P_j. Multicast of fail-signal is shown as phase 0 in figure 5.1. Receipt of this fail-signal triggers Install part at p_i. p_i multicasts a $STATUS_i$ message in response to each fail-signal it receives. When the fail-signal is from a non-coordinator FS process i.e., $j \neq c$, phase I is the only phase executed at p_i after which the execution is transferred back to Normal part. On the other hand, when $j = c$, Install execution continues to get the next coordinator, generically referred to as the *eligible*() or P_{c+}, installed at p_i. Hence, the description of the remaining four phases below assume $j = c$.

Phase II – Send $START_{c+}$: The *eligible*() process P_{c+} collects valid *STATUS* messages from a quorum, prepares and multicasts a 2-signed $START_{c+}$ message to all processes. This is the message which P_{c+} uses to multicast *TransferHistory* which is necessary to change configuration. Preparation of $START_{c+}$ is a complex process and is shown as part A of phase II in figure 5.1 and will be described in detail shortly.

Phase III – Sign $START_{c+}$: p_i receives $START_{c+}$, signs it to form $START_{c+,i}$ (suffixes show signatories) and sends it back to P_{c+}.

Phase IV – Send (f+1)-$START_{c+}$ and $ORDER_{c+}(start\_o_{c+})$: After receiving (f_c-1) messages from phase III, P_{c+} prepares an (f+1)-$START_{c+}$ message, which is nothing but $START_{c+}$ plus (f_c-1) signatures. (f+1)-$START_{c+}$ is then multicast to all processes. This message will be treated as a sort of client request by destinations and P_{c+} prepares and multicasts $ORDER_{c+}(start\_o_{c+})$ for this (f+1)-$START_{c+}$. The order number assigned to (f+1)-$START_{c+}$, $start\_o_{c+}$, is one more than the largest o in the *TransferHistory* inside $START_{c+}$ (see section 5.2).

Phase V – Commit $ORDER_{c+}(start\_o_{c+})$: This is similar to the *COMMIT* phase of Normal part of the protocol wherein $ORDER_{c+}(start\_o_{c+})$ is committed. Each p_i multicasts $ACK_i(start\_o_{c+})$ and waits to receive $ACK_i(start\_o_{c+})$ from a quorum.

At the end of phase V when $ORDER_{c+}(start\_o_{c+})$ becomes committed, installation of P_{c+} as coordinator is completed; system moves to configuration Σ_{c+} and execution is switched back to Normal part.

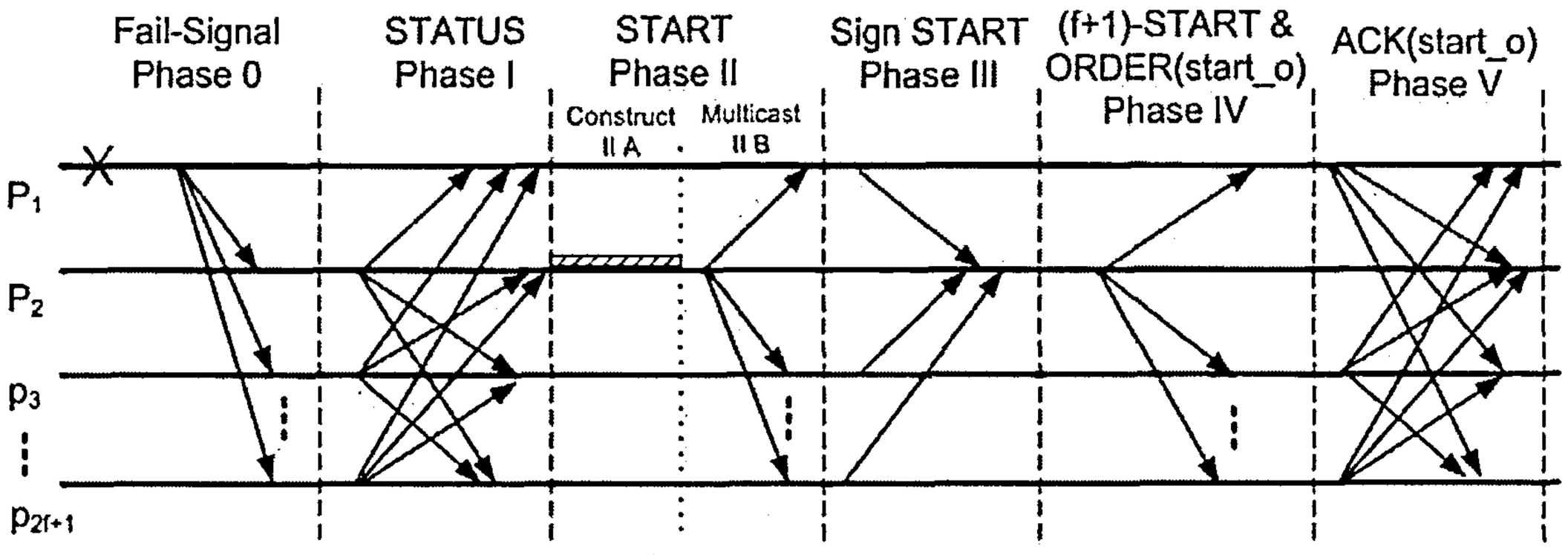


Figure 5.1. All 5 phases of Install part of Protocol-I

Following sections present the detailed structure of the messages mentioned above and also the description of each protocol step.

5.5 Data Structures

5.5.1 Messages

The contents of the messages used by Install part are given below. We first introduce the notation $m\_c_i(j)$ and $A_i(j)$. Recall that $max\_committed_i$ refers to the largest o committed by p_i at any given moment (defined in subsection 4.4.2). Let $max\_committed_i(j)$ be the largest o for which an $ORDER_j(o)$ is committed by p_i only amongst those *ORDER*s sent by P_j. We use $m\_c_i(j)$ as a short form for

max_committed$_i(j)$. $m\_c_i(j)$ is initialized to 0 for all j, $1 \leq j \leq (f+1)$. Similarly, A_i is the largest o for which an $ACK_i(o)$ is produced by p_i (subsection 4.4.2). We use $A_i(j)$ to denote largest o for which an $ORDER_j(o)$ is acknowledged by p_i only amongst those *ORDER*s sent by P_j.

5.5.1.1 STATUS$_i$(j)

$STATUS_i(j)$ message is multicast by process p_i in response to receiving a fail-signal from P_j, $j \geq c$, where c is the rank of current coordinator at p_i. The purpose of this message is to provide the next coordinator with the history of order messages (if any) that were sent by P_j and *ACK*ed by p_i. Note that once p_i knows that P_j has fail-signalled, it stops acknowledging any order message from P_j.

$STATUS_i(j)$ has following three components, as shown in figure 5.2(a).

(a) ***FS*$_j$:** The received *fail-signal* message from P_j,

(b) ***HistoryProof*:** *proof_of_commitment* for $ORDER(m\_c_i(j))$.

(c) ***AckHistory*$_i$:** List of $\langle ACK(o), ORDER(o)\rangle$ pairs for every $ORDER_j(o)$ message sent by P_j which p_i acknowledged with $o > m\_c_i(j)$. Note that $o = A_i(j)$ will be the largest o in the list.

Notes:

- *HistoryProof* will be null if p_i never installed P_j as coordinator or in other words, $m\_c_i(j) = 0$. Note that installation of P_j can only be done by committing $ORDER_j(start\_o_j)$. Moreover, if P_j was installed but no other $ORDER_j(o)$ was committed after the installation then $m\_c_i(j) = start\_o_j$. Hence, lower bound on $m\_c_i(j)$ is $start\_o_j$.
- When $m\_c_i(j) = 0$, $AckHistory_i$ will contain either of the following.

  (i) If p_i had acknowledged $ORDER_j(start\_o_j)$ i.e., P_j fail-signalled while executing Install as *eligible*() = j and managed to execute at least the first four phases successfully (see figure 5.1), $AckHistory_i$ will consist of singleton pair $\langle ACK(o), ORDER(o)\rangle$, where $o = start\_o_j$, Or

  (ii) $AckHistory_i$ will be empty, in all other cases, except when $c = 1$.

5.5.1.2 $START_{c+}(c)$

This message is multicast by the new coordinator P_{c+}. It also contains three fields. Figure 5.2(b) illustrates the structure of $START_{c+}(c)$.

(a) ***Signalled*$_{c+}$:** The set of all received *fail-signal* messages by P_{c+}. This justifies P_{c+}'s attempt to execute Install as *eligible*().

(b) ***c*:** Last installed coordinator number.

(c) ***TransferHistory*:** List of all *ORDER* messages which must be committed by at least a quorum prior to installation of P_{c+}. It contains *ORDER*(*o*) messages with *o* in [CC_{mx}, PC_{mx}]. The details of how *TransferHistory* is constructed are given in section 5.7.

5.5.1.3 $START_{c+,i}(c)$

This message is constructed by p_i in response to $START_{c+}(c)$. It shows p_i's agreement over contents of $START_{c+}(c)$ and is constructed by signing $START_{c+}(c)$. Every process p_i signs the doubly signed $START_{c+}(c)$ to form $START_{c+,i}(c)$ and sends it back to P_{c+}.

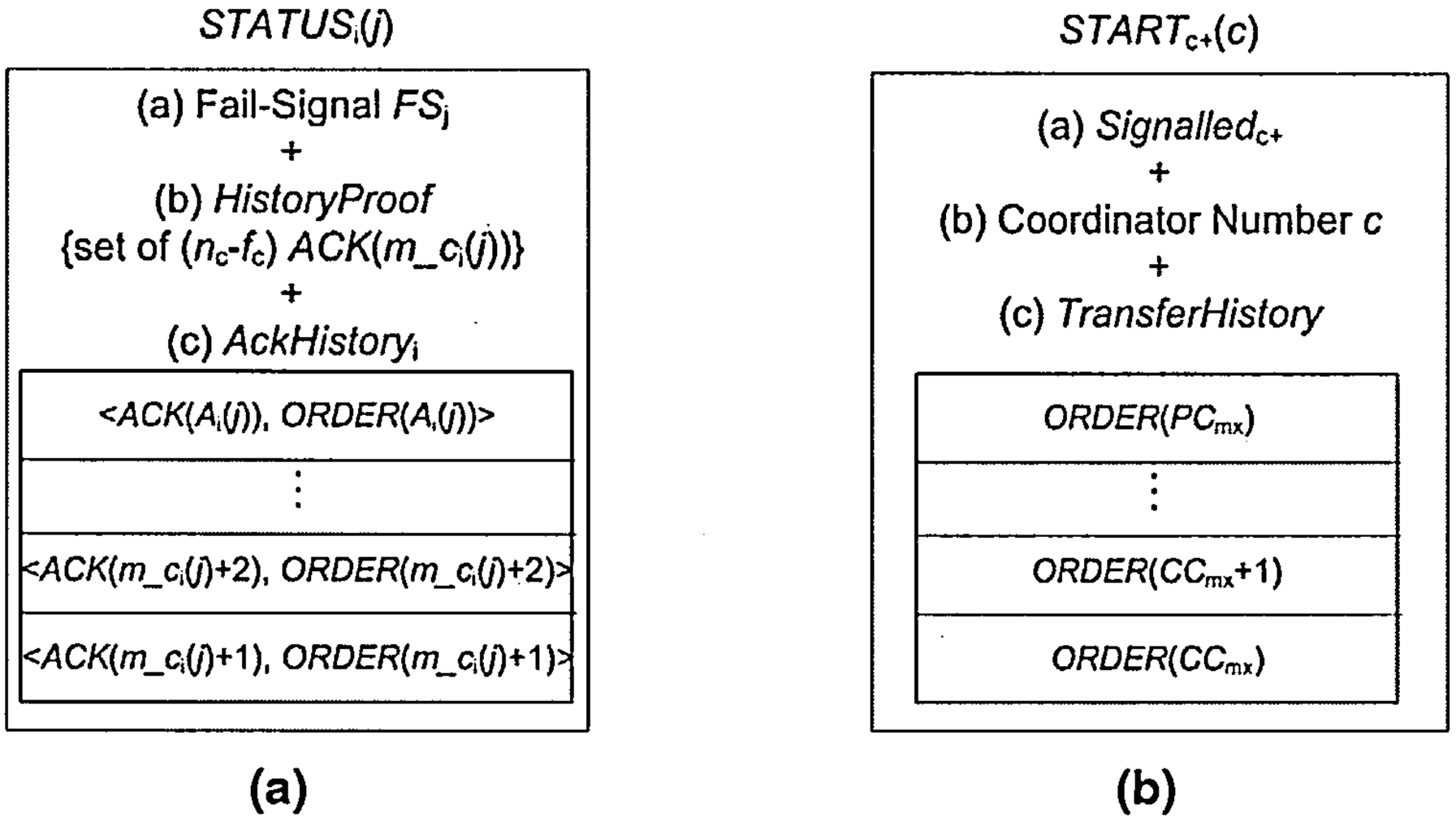


Figure 5.2. Structure of (a) $STATUS_i(j)$ and (b) $START_{c+}(c)$ messages

5.5.2 Variables and other data structures

Processes p_i and p'_i maintain variables *c*, $Signalled_i$, *Order_Pool*, *Ack_Pool*, and function *eligible*() – all of which having same meaning as in Protocol-0 (subsection 3.4.3.2). Additionally variables A_i, $max\_committed_i$, *n*, *f*, data structure *proof_of_commitment* and predicate *committable*(*o*) are maintained as defined in subsection 4.4.2 and 4.3.1. Also, n_k , f_k and $Acceptors_k$, $1 \leq k \leq f+1$, are maintained as constants for each configuration Σ_k as defined in subsection 4.3.1. Furthermore the following are also maintained:

commit_count = vector containing the largest committed request number for every client (initially *commit_count*[*k*] = 0 for every cl_k);

Status_Pool(j): set of *STATUS(j)* messages sent/received (initially empty);

Start_Pool: set of $START_{i,j}(k)$ messages received from p_j (initially empty);

5.6 Algorithm Steps

This section discusses the actions performed during Install. These steps are shown in figure 5.3. They are expressed as 6 concurrent tasks. The first five of these correspond to five phases of Install described in section 5.4 (figure 5.1). Whereas the sixth task represents the terminal activities executed at the end of phase V.

Referring to figure 5.1, it can be noted that the various phases have processes acting on certain events. For example, the process that satisfies *eligible*() acts in phase II and IV, every process acts in phase I and V and every process excluding *eligible*() behave as actors in phase III. We present the execution steps of Install in generic terms with respect to a process $p_i \in \pi \cup \pi'$. In the presentation below p_i receives messages from other process p_j and p_k and acts on them only if it needs to act in the respective phase. For example, p_i will produce $START_i(j)$ and $ORDER_i(start\_o_i)$ only when i = *eligible*().

```
while (true) do →
{                                                    /Task 1
(FS_j received or STATUS_k(j) received) →
{
I1.1  if (P_j ∉ Signalled_i) →
      {
I1.2        enter P_j into Signalled_i; // record fail-signalling by P_j
I1.3        prepare & multicast STATUS_i(j) to all p ∈ π∪π';
I1.4        if (c = j) →
            {
I1.5              s = c;
I1.6              c = -1; // Install part execution continues
I1.7              Order_Pool = Order_Pool\{ORDER(o>A)};
            }
      }
I1.8  if (STATUS_k(j) received) →
      {
I1.9        if (p_k ∈ Acceptors_j) →
I1.10             enter STATUS_k(j) into Status_Pool(j);
      }
}
||                                                   /Task 2
if (i = eligible() and START_i(s) not sent) →
{
I2.1        prepare START_i(s)= construct_Start(); //Phase IIA
I2.2        multicast START_i(s) to all p ∈ π∪π'; // Phase IIB
}                                                              contd...
```

Figure 5.3 Algorithm for Install Part

```
||                                                          /Task 3
(START_k(j) received) →
{
I3.1  if (k = eligible() and i ≠ k) →
I3.2        send START_k,i(j) to p_k and p'_k;
}
||                                                          /Task 4
(START_i,k(j) received) →
{
I4.1  if (p_k ∈ Acceptors_j and i = eligible() ) →
      {
I4.2        enter START_i,k(j) into Start_Pool;
I4.3        if ( |Start_Pool| = (f_j-1) ) →
            {
I4.4              prepare & multicast (f+1)-START_i(j) to all p ∈ π∪π';
I4.5              prepare & multicast ORDER_i(start_o_i) to all p ∈ π∪π';
            }
      }
}
||                                                          /Task 5
(ORDER_k(start_o_k) and (f+1)-START_k(j) received and k = eligible()) →
{
I5.1  if (ACK_i(start_o_k) not sent) →
      {
I5.2        Order_Pool = Order_Pool \
                  {ORDER(o): o ≥ maximum {max_committed_i + 1, CC_mx} };
I5.3        enter ORDER_k(start_o_k) into Order_Pool;
I5.4        prepare & multicast ACK_i(start_o_k) to all p ∈ π∪π';
I5.5        enter ACK_i(start_o_k)into Ack_Pool;
I5.6        A_i = start_o_k;
I5.7        while (∃ o < CC_mx : ORDER(o) ∉ Order_Pool) do →
            {
I5.8              invoke retrieve(o);
            }
      }
}
||                                                          /Task 6
(ACK_j(start_o_k) received) →
{// assuming ORDER_k(start_o_k) corresponds to (f+1)-START_k(s)
I6.1  if p_j ∈ Acceptors_s →
      {
I6.2        enter ACK_j(start_o_k)into Ack_Pool;
I6.3        if (committable(start_o_k) = true and k = eligible() ) →
            {
I6.4              c = k;
I6.5              enter into Σ_c;
            }
      }
}
} // end while
```

Figure 5.3 Algorithm for Install Part

5.6.1 Description

The first 5 tasks of figure 5.3 correspond to the respective 5 phases of Install part (fig 5.1). Task 6 is the terminal task that ends phase V and hence the Install part. Description of each task is given below.

Task 1 – This task corresponds to phase I of Install. p_i multicasts $STATUS_i(j)$ message (line I1.3) after receiving a fail-signal from P_j for the first time (lines I1.1 and I1.2). Furthermore, if p_i finds that the coordinator has fail-signalled (line I1.4), it records c in variable s (line I1.5) and sets c to an invalid coordinator number (-1) in line I1.6. In this case, it also deletes all accepted but unacknowledged *ORDER* messages from its *Order_Pool* (line I1.7). p_i also enters $STATUS_k(j)$ messages received from other acceptors in its *Status_Pool*(j) (line I1.10).

Task 2 – This task simply makes the *eligible*() process prepare $START_i(j)$ message and multicast it to all processes (lines I2.1 and I2.2). The *eligible*() process uses the configuration-related parameters as those used in the signalled coordinator's configuration Σ_j throughout install execution e.g., Q_j and $Acceptors_j$. Details of the steps involved in *construct_Start*() are to be explained shortly in section 5.7.

Task 3 – Every process, except *eligible*(), signs and sends $START_{k,i}(j)$ to the *eligible*() process as shown in figure 5.1.

Task 4 – *eligible*() process accumulates ($f_j - 1$) signatures (line I4.2 and I4.3) after P_j's failure. It prepares and multicasts (f+1)-$START_i(j)$ and corresponding $ORDER_i(start\_o_i)$ (line I4.4 and I4.5).

Task 5 – This represents the last communication phase of Install. Every process p_i, on receiving (f+1)-$START_k(j)$ and corresponding $ORDER_k(start\_o_k)$, acknowledges the receipt by preparing and multicasting $ACK_i(start\_o_k)$ (line I5.4). Since $ORDER_k(start\_o_k)$ is a special *ORDER* message that corresponds to (f+1)-$START_k(j)$ which contains *TransferHistory*, a list of some *ORDER* messages, acknowledging $ORDER_k(start\_o_k)$ is taken as acknowledgement for each $ORDER(o')$, $CC_{mx} \leq o' \leq PC_{mx}$, in *TransferHistory*. Therefore, before preparing $ACK_i(start\_o_k)$, p_i deletes every acknowledged but uncommitted $ORDER(o)$, $o \geq CC_{mx}$, from its *Order_Pool* (line I5.2). This is to make sure that p_i does not end up having duplicates in its *Order_Pool* by acknowledging $ORDER_k(start\_o_k)$ and adding all *ORDER*s in *TransferHistory* in its *Order_Pool*. After multicasting $ACK_i(start\_o_k)$, p_i updates its *Order_Pool*, *Ack_Pool* and A_i (lines I5.3, I5.5, I5.6).

Note that at this stage there may be some $ORDER(o)$, $o < CC_{mx}$, which may have not been (even) accepted at p_i. This is trivial to deal with as p_i can invoke a *retrieval mechanism* to recover all such messages. This is represented as invoking `retrieve()` in line I5.8 and is explained in section 5.9. In short, since $ORDER_j(CC_{mx})$ is certainly committed at some correct process, at least (f_j+1) correct processes would have acknowledged it. Since all *ACK* messages are prepared in sequence, it is guaranteed that if p_i multicasts a request to retrieve any $ORDER_j(o)$, $o < CC_{mx}$, p_i will receive at least (f_j+1) responses. This will also be true for any $ORDER_a(o)$, $a < j$, which will have supporting (f_a+1) *ACK* messages. The latter is possible because after failure of a coordinator P_j, the *eligible*() process P_k only gets installed at any process p_i after $ORDER_k(start\_o_k)$, and hence every $ORDER_j(o')$, $CC_{mx} \leq o' \leq PC_{mx}$, becomes committed with quorum size Q_j (and not Q_k). Hence, new coordinator starts its regime after making sure that all orders from its predecessor coordinator become committed with the preceding configuration parameters.

Task 6 – This is the last task of Install part involving only computation. It ends when $ORDER_k(start\_o_k)$ gets committed at p_i. Again, it is important to point out that the quorum used for this commitment is the one used in previous configuration (line I6.3, also see suffix of *Acceptors* in I6.1). It is only after $ORDER_k(start\_o_k)$ gets committed, p_i updates the coordinator number to the process number of the *eligible*() (lines I6.4) and enters into new configuration (line I6.5). P_k is considered as current coordinator and Normal part execution begins. Note that it is implicit here that since Normal part execution is resumed with a new value of $c = k$, configuration parameters will be used correspondingly i.e., n_k, f_k, Q_k and $Acceptors_k$.

5.6.2 An Example Scenario

Let us consider a system working in Σ_a with P_a as coordinator. Say, P_a fail-signals after multicasting $ORDER(x)$, $x > 1$. P_b receives FS_a and executes Install as *eligible*(). P_b waits for Q_a *STATUS* messages and multicasts $START_b$. Let us assume $ORDER(x)$ was committed at some process, hence, it would be included in $START_b$. Say, P_b is successfully installed at some Q_a processes. Let p_i be a process connected via slow links and does not receive FS_a for a long time. Hence p_i will still be working in Σ_a. Say p_i has only received orders up till $ORDER(u)$, $u < x$.

Now assume that P_b multicasts orders up to *ORDER*(*y*) and fail-signals. Let P_c be the *eligible*() process, which is successfully installed at Q_b processes. Like before, assume that *ORDER*(*y*) was committed, hence it will certainly be a part of $START_c$. Now, when at this stage p_i receives $START_c$, which contains FS_a and FS_b, it knows that P_c is the current coordinator. But $START_c$ does not contain messages sent by P_a which p_i did not receive. Hence, p_i will try to retrieve these messages. We note here that since $START_b$ contains *ORDER*s from P_a and $START_b$ is committed by Q_a, out of which at least (f_a+1) will be correct processes knowing about all committed orders from P_a. p_i is guaranteed to get at least (f_a+1) responses for all orders from P_a and at least (f_b+1) responses for all orders from P_b. Hence, slow processes can get synchronized with others at any time and can fill all missing orders from any predecessor coordinators.

To summarize, P_a, P_b and P_c are three consecutive coordinator processes. P_a and P_b fail-signalled after ordering some requests. We note the following two points.

1. *TransferHistory* of $START_b$ contains only $ORDER_a$ messages (by P_a) and that of $START_c$ contains only $ORDER_b$ messages.
2. $START_b$ (respectively $START_c$) is committable when Q_a (resp. Q_b) processes acknowledge.

Hence, at least Q_a (resp. Q_b) processes will know about all orders from P_a (resp. P_b) before P_c can start ordering new requests. Since from Q_a (resp. Q_b), at least f_a+1 (resp f_b+1) processes are correct, Therefore, it is easy to conclude that any process p_i, that needs to retrieve some missing messages from any of the predecessor coordinators P_x, is guaranteed to receive at least (f_x+1) responses.

5.6.3 Exceptional Case: Last Coordinator

When *eligible*() = (*f*+1) (not f_i+1), p_{f+1}, which is going to be the last coordinator, installs itself. This is the only exceptional case where a non-FS process executes Install as *eligible*(). Therefore we note below a couple of exceptions needed in the regular execution of Install part.

1. Being a non-FS process, p_{f+1} only prepares single-signed $START_{f+1}$ message.
2. Since all *f* failures have occurred in the system, $START_{f+1}$ message no longer needs to be secured with the help of (f_c+1) acceptor signatures (more on this in subsection 5.10.2). Hence phases III and IV shown in figure 5.1 are not executed by any process when the system moves to the last configuration $\Sigma_{(f+1)}$.

5.7 Construction of $START_{c+}$(c)

$START_{c+}(c)$ message, like any other doubly-signed message, is constructed in two steps i.e., first p_{c+} creates and signs the message and then p'_{c+} verifies and double-signs it (see figure 3.4 for output endorsement steps). However, $START_{c+}(c)$ is a special message whose construction relies on the set of $STATUS_i(c)$ messages *received* so far. Two issues arise because of this reliance as described below.

Firstly, we note that $START_{c+}(c)$ is prepared with the help of $STATUS_i(c)$ messages received from at least a quorum. Since quorum used by p_{c+} can be different from the one used by p'_{c+}, $START_{c+}(c)$ constructed by the two processes independently may not be same but yet correct. Therefore we define quorum used in the preparation of $START_{c+}(c)$ to be consisting of the first Q_c $STATUS_i(c)$ messages received at P_{c+}. Since p_{c+} and p'_{c+} are synchronized for the messages they receive (see ISIQ in figure 3.3, chapter 3), this will make $START_{c+}(c)$ construction identical at both processes. Therefore both p_{c+} and p'_{c+} wait for the arrival of *first* set of Q_c $STATUS_i(c)$ messages before starting construction of $START_{c+}(c)$ message. We note here that although it is this set that mainly contributes towards the contents of $START_{c+}(c)$ message, later arriving $STATUS_i(c)$ messages are also significant and may be needed for conflict resolution. To distinguish between the two, we assume that the *Status_Pool*(c) consists of two portions. One named *Foundation_Pool*(c), comprises of the first arriving Q_c $STATUS_i(c)$ messages, while the second named *Advanced_Pool*(c) contains all later arriving $STATUS_i(c)$ messages.

Second issue is concerned with the way p_{c+} (or p'_{c+}) resolves conflicts, if any found in $STATUS(c)$ messages of their *Foundation_Pool*(c). Recall that the evidence to identify a plant against a committed message $ORDER(o, r)$ is guaranteed to be available within the $STATUS_i(c)$ messages received by the time this conflict appears i.e. in whole *Status_Pool*(c) (*Foundation_Pool*(c) + *Advanced_Pool*(c)). This will be in the form of at least (f_c+1) $STATUS_i(c)$ messages having included $ORDER(o, r)$. Hence both p_{c+} and p'_{c+} are guaranteed to choose $ORDER(o, r)$ consistently. However for a plant against an uncommitted $ORDER(o, r)$, no guarantee can be given about the number of $STATUS_i(c)$ messages having included $ORDER(o, r)$. Since the time instant at which p_{c+} and p'_{c+} will be resolving this conflict may be different, the total number of $STATUS_i(c)$ messages in the *Advanced_Pool*(c) of the two processes at those respective moments may also be different. Hence it may happen that p_{c+} finds more than f_c $STATUS_i(c)$ messages

containing *ORDER*(o, r) while p'_{c+} does not or vice versa. This will cause inconsistency. We therefore propose below a solution to avoid such a situation.

Analysing the problem, we conclude that it can be solved if p_{c+} and p'_{c+} exchange the evidence that they have used from $STATUS_i(c)$ messages of *Advanced_Pool*(c) to construct their respective $START_{c+}(c)$ messages. Hence both p_{c+} and p'_{c+} bundle this evidence into a message called $PRE\text{-}START_{c+}(c)$ and send it to each other. $PRE\text{-}START_{c+}(c)$ has following two components

(a) Single signed $START_{c+}(c)$, individually constructed by both p_{c+} and p'_{c+}.

(b) *ProofBag*: list containing evidence corresponding to every *ORDER* included in the *TransferHistory* of $START_{c+}(c)$ in (a).

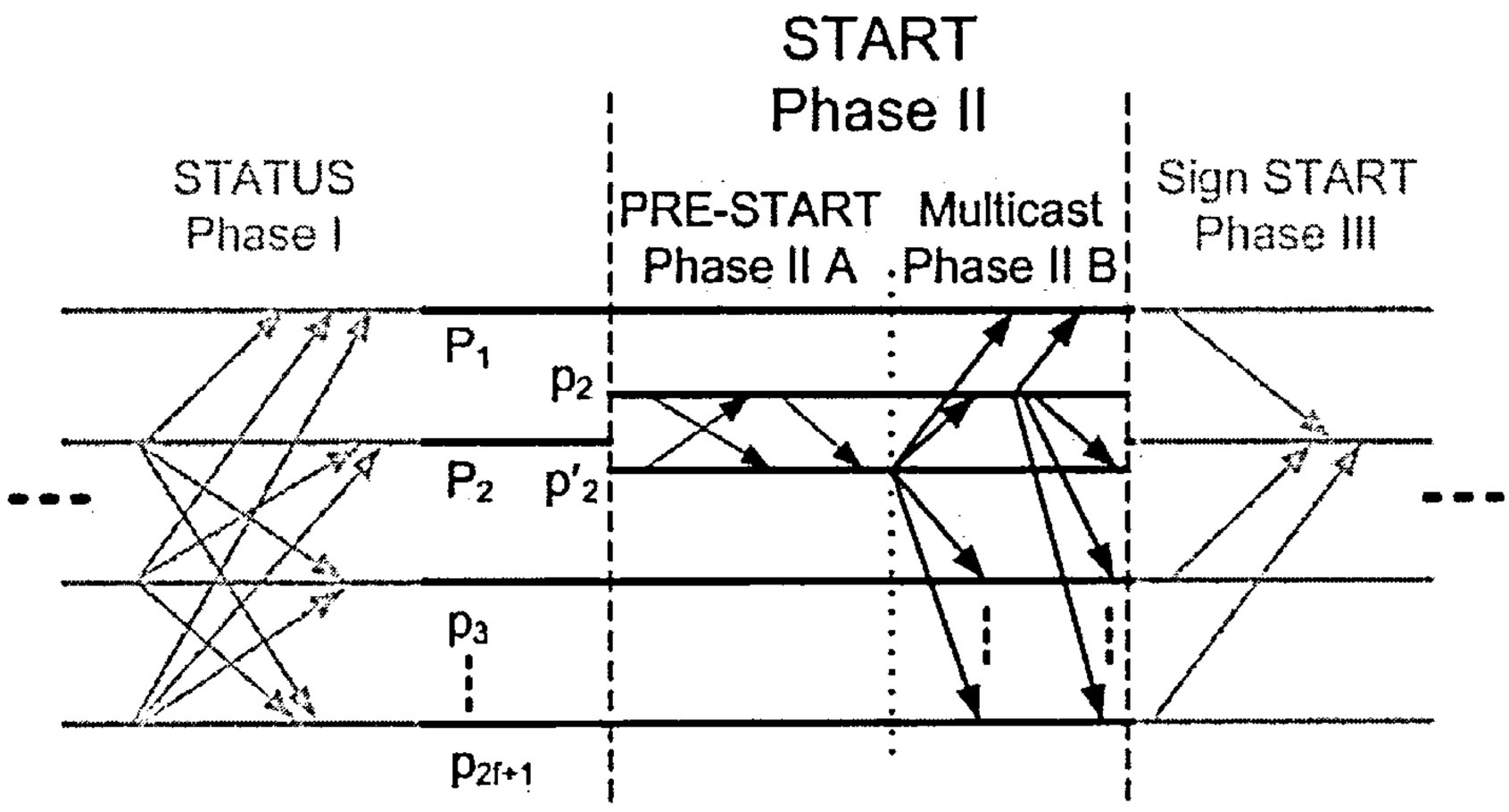


Figure 5.4. Extended representation of phase 2 of Install part

Once the $PRE\text{-}START_{c+}(c)$ messages are exchanged, normal output endorsement procedure is initiated. Figure 5.4 shows the detailed version of phase II from the 5 phases of Install part (Fig 5.1). Details of phase IIA, named *PRE-START* phase here, have been added. This phase depicts message communication between the two paired processes within the *eligible*() FS process (the only active FS process in the system) as described above. The latter is depicted as P_2 in figure 5.4. Recall that activities in this phase are a part of the *construct_Start*() procedure used in figure 5.3 (line I1.7). Figure 5.5 below shows an outline of steps executed in this procedure with the context that P_c has fail-signalled and *eligible*() = $c+$ is executing the Install.

```
Construct_Start()
{
CS1    wait until |Foundation_Pool(c)| = Qc;
CS2    prepare PRE-STARTc+(c) and send it to counterpart process;
CS3    wait until PRE-STARTc+(c) received from counterpart process;
CS4    prepare STARTc+(c) using the two PRE-STARTc+(c);
CS5    if (leader) → // for pc+
       {
CS6          send single-signed STARTc+(c) to counterpart;
CS7          wait until double-signed STARTc+(c) received from
       counterpart;
CS8          return double-signed STARTc+(c);
       }
CS9    else // for p'c+
       {
CS10         wait until single-signed STARTc+(c) received from
       counterpart;
CS11         verify and sign single-signed STARTc+(c);
CS12         return double-signed STARTc+(c);
       }
}
```

Figure 5.5. Steps for *START* construction; *construct_Start*()

Lines CS1 – CS4 of figure 5.5 are executed by both processes in the pair symmetrically, which represent preparation and exchange of $PRE\text{-}START_{c+}(c)$ between the two followed by preparation of $START_{c+}(c)$ using the two $PRE\text{-}START_{c+}(c)$ messages. (This is indicated by information flow between p_2 and p'_2 in PRE-START phase of figure 5.4.) What follows is the normal output endorsement procedure initiated by leader process p_{c+} (line CS6). Follower process p'_{c+} receives, verifies and double signs the $START_{c+}(c)$ sent by leader (lines CS10, CS11). Double-signed $START_{c+}(c)$ is returned back (line CS12) to the calling program so that it can be multicast to all (line I1.7 figure 5.3). Leader also returns this message after receiving it from the follower (line CS7 and CS8).

Rest of this section contains steps of preparation of $PRE\text{-}START_{c+}(c)$ and reconstruction of $START_{c+}(c)$, each in a separate subsection below. The procedure for preparation of these messages is same for both p_{c+} and p'_{c+}. We therefore describe it for one of them i.e. p_{c+} only.

5.7.1 Preparation of $PRE\text{-}START_{c+}(c)$

p_{c+} uses *Foundation_Pool*(c) to construct the two components of $PRE\text{-}START_{c+}(c)$; single-signed $START_{c+}(c)$ containing *TransferHistory* and the corresponding *ProofBag*.

Advanced_Pool(c) is used only to resolve conflicts and is mentioned explicitly where used. Recall that *Foundation_Pool*(c) contains $STATUS_i(c)$ messages, and a $STATUS_i(c)$ contains *HistoryProof* and $AckHistory_i$. The following construction procedure is described in terms of these two components.

p_{c+} goes through all *HistoryProof*s in *Foundation_Pool*(c) and selects the one containing the proof for the largest o among all. This will form the smallest o of *ORDER*(o) in *TransferHistory* i.e., CC_{mx}. We denote the proof for *ORDER*(CC_{mx}) as *HistoryProof*(CC_{mx}). *ORDER*(CC_{mx}) is added as first element in *TransferHistory* and *HistoryProof*(CC_{mx}) in *ProofBag*. Since *ORDER*(CC_{mx}) is definitely acknowledged by at least (f_c+1) correct processes, *TransferHistory* of $START_{c+}(c)$ is constructed to contain only *ORDER*($o \geq CC_{mx}$). Hence the first element of *ProofBag* justifies the lowest numbered *ORDER* in *TransferHistory*. If all the *HistoryProof*s in the *Foundation_Pool*(c) are null, first element of *ProofBag* will be set to null. In this situation p_{c+} will be expecting the lowest order number in all non-empty *AckHistory*s in *Foundation_Pool*(c) to be either 1 or $start\_o_c$. The former represents the case where c = 1 (P_c was first coordinator), while the latter is the case when P_c's attempt to install itself was unsuccessful i.e. $ORDER_c(start\_o_c)$ was sent by P_c but could not become committed.

There is a possibility of p_{c+} having $max\_committed_{c+} < CC_{mx}$. In this case, p_{c+} enters all *ACK*s from *HistoryProof*(CC_{mx}) in its *ACK_Pool* and considers *ORDER*(CC_{mx}) as committed. This requires $max\_committed_{c+}$ and *commit_count* to be updated accordingly. Updating $max\_committed_{c+}$ is easy but for *commit_count*, p_{c+} needs to be in possession of an authentic *ORDER*(o) and corresponding r for every o, $max\_committed_{c+} < o \leq CC_{mx}$. Since clients are assumed to be sending requests to all processes faithfully, receiving r corresponding to all these *ORDER*s is not considered a problem. Hence the update can only be done if p_{c+} has locally accepted each of these *ORDER*s. For every *ORDER*(o), $o < CC_{mx}$, not being accepted yet, p_{c+} triggers retrieval mechanism as described in section 5.9 to recover those messages.

As mentioned earlier, *TransferHistory* contains all *ORDER* messages that bear a chance of having been committed. This means that it must contain every authentic *ORDER*($o \geq CC_{mx}$) that was included in any $AckHistory_i$ in *Foundation_Pool*(c), as was indicated as a caveat in subsection 5.3.2. p_{c+} inspects all *AckHistory*s and for every successive *ORDER*(o, r) above CC_{mx}, it does the following.

a. If there exists an *ACK*(o) corresponding to *ORDER*(o, r) in at least (f_c+1) *STATUS*$_i$(c) messages, p_{c+} adds *ORDER*(o, r) in *TransferHistory* and a set of distinct (f_c+1) *ACK*(o) messages in *ProofBag*.

b. In case of an *ORDER*(o, r) having less than (f_c+1) supporting *ACK*(o) messages, p_{c+} first performs spuriousness check in the following way.

If r = request from client cl_i with request number r_i and

m = number of requests from cl_i that were assigned order numbers in the range [o-1, CC_{mx}+1] i.e. whose order messages have been included in *TransferHistory* then r_i should satisfy the following equation

$$r_i = commit\_count[i] + m + 1$$

For example if the latest committed request sent by cl_i is 10 i.e. *commit_count*[i] = 10. Suppose CC_{mx} is computed as 20. Therefore *TransferHistory* will contain *ORDER*s from 20 above. Let the *ORDER* of concern be *ORDER*(30) for which p_{c+} needs to perform spuriousness check. Say *ORDER*(30) corresponds to request number 13 from client cl_i i.e., r_i = 13. p_{c+} then checks if there are r_i - *commit_count*[i] - 1= 13 – 10 – 1 = 2 *ORDER* messages in *TransferHistory* below 30 corresponding to request numbers 11 and 12 from cl_i. If this is not the case then *ORDER*(30) is considered spurious.

Once *ORDER*(o, r) is identified as non-spurious message, it may fall into any of the following two categories:

Conflict-free Order: This is the situation when every *ACK*(o) in *Foundation_Pool*(c) refers to same *ORDER*(o, r).

Conflict-resolved Order: This is the situation when multiple conflicting *ORDER* messages are found corresponding to same o in *Foundation_Pool*(c). Conflict is resolved in the way explained in subsection 5.3.3 using *all AckHistory*s in the *Status_Pool*(c) (*Foundation_Pool*(c) + *Advanced_Pool*(c)) for evidence. Let *ORDER*(o, r) be the chosen conflict-resolved order.

For both cases above, p_{c+} adds *ORDER*(o, r) in *TransferHistory* and one of the corresponding *ACK*(o) from an *AckHistory* in the *ProofBag*.

5.7.2 Adjusting $START_{c+}(c)$ using received $PRE\text{-}START_{c+}(c)$

p_{c+} carries out the following steps to reconstruct $START_{c+}(c)$. Let us use suffix $c+$ for components of $PRE\text{-}START_{c+}(c)$ produced by p_{c+}, $c+'$ for the ones produced by p'_{c+} and suffix *start* for the ones that are recomputed for the final $START_{c+}(c)$. These suffixes are only used in this subsection for clarification purposes. Recall that CC_{mx} is computed by using *STATUS* messages in *Foundation_Pool* only. Since the messages in this pool at both p_{c+} and p'_{c+} are same, CC_{mx} computed at the two processes will also be same.

(a) p_{c+} constructs $TransferHistory_{start}$ for the new $START_{c+}(c)$ in the following way. For every successive $ORDER(o)$, starting from $o = CC_{mx}$, it selects $ORDER(o, r)$ from $TransferHistory_{c+}$ and adds it into $TransferHistory_{start}$ except the following two cases:

1. When there exists an $ORDER(o, r')$, $r \neq r'$, in $TransferHistory_{c+'}$ with more than f_c corresponding $ACK(o)$ messages as evidence in $ProofBag_{c+'}$, or
2. When $ORDER(o)$ does not exist in $TransferHistory_{c+}$ i.e. when largest o in $TransferHistory_{c+'}$ > largest o in $TransferHistory_{c+}$.

It is only in these two situations when $ORDER(o, r')$ from $TransferHistory_{c+'}$ is selected. Every order is re-checked for spuriousness in the way described for $PRE\text{-}START_{c+}(c)$ construction.

(b) It then computes PC_{mx} = the largest o in $TransferHistory_{start}$ and $start\_o_{c+} = PC_{mx} + 1$.

p_{c+} sends this newly prepared $START_{c+}(c)$ to p'_{c+} to get signed. p'_{c+} compares it with locally produced $START_{c+}(c)$ and signs it if match found. 2-signed $START_{c+}(c)$ is then multicast to all processes.

5.8 Identifying Source Number

Up till this point, a simple assumption has been made to explain the complicated task of construction of $START_{c+}(c)$. P_{c+} was assumed *to be working in* Σ_c when it received the fail-signal from P_c. This assumption may not always hold. We will refer to the big picture sketched in subsection 5.1.1 to explain the complication of the situation. Recall the initial assumption about having an observer client. It was highlighted that the eligible process faces a dilemma. That is, it needs to find out if it is supposed to re-use a *START* message already issued by the observer client or it should wait for a new, tailor-made *START*. Moreover, since there is no such observer client in reality, the eligible

process thus needs to decide whether there is a *START* message generated by the predecessor coordinator which it can re-use or it should prepare a new one. In order to make this decision, the eligible process needs to collect some details from other processes. For example, it needs to know the identification of the latest coordinator that was installed at any correct process and the latest process, say P_e, which fail-signalled during install execution as *eligible*() after transmitting $ORDER_e(start\_o_e)$, if any. Concisely,

1. If there exists a process P_e that executed install after the latest coordinator fail-signalled but could execute only first four phases and fail-signalled after transmitting $ORDER_e(start\_o_e)$, P_{c+} needs to identify this situation about P_e. This is to enable P_{c+} to re-use $ORDER_e(start\_o_e)$. Otherwise,
2. P_{c+} needs to identify the process number c of the largest ranked coordinator that was installed at any correct process. This is to enable P_{c+} to recognize committed and possibly committed *ORDER*s to be included in *TransferHistory*.

The process number identified by P_{c+} is called *Source Number* and is denoted by s. We show that the value of s can be identified with the help of *Status_Pool*. First we introduce a data structure named *Status_Board* maintained by P_{c+} and then explain the procedure for identification of s.

Recall that $STATUS_i(k)$ message is multicast by p_i in response to every fail-signal received (line I1.3 in fig 5.3). P_{c+} maintains a *Status_Pool*(k) for every $P_k \in Signalled_{c+}$. This collection is called *Status_Board*. *Status_Board* helps P_{c+} handle the identification of source number in the following way.

Recall that process p_i sends an empty $AckHistory_i$ and null *HistoryProof* when it receives fail-signal from a non-coordinator FS process, say P_u (see $STATUS_i(j)$ in subsection 5.5.1). Let us call such a $STATUS_i(u)$ message empty and a *Status_Pool*(u) containing all empty *STATUS*(u) messages empty pool. Hence an empty *Status_Pool*(u) represents that at least a quorum could not install P_u and never entered (and will never enter) configuration Σ_u. Note that the converse is not always true i.e. that the *eligible*() process could not be installed successfully does not necessarily result in empty *Status_Pool*. For example, an eligible coordinator P_e may fail-signal after executing first 4 phases of Install part. Some correct process p_v which may have multicast $ACK_v(start\_o_e)$ will include $ACK_v(start\_o_e)$ in its $AckHistory_v$. In other words, a non-empty *Status_Pool*(e) does not necessarily mean that P_e was installed successfully and

Σ_e was entered. However, any non-empty *Status_Pool*(e) contains important information.

Identifying the source number thus becomes simple for P_{c+}. P_{c+} computes s as the largest j, $j < c+$, such that every *Status_Pool*(k) has Q_j *STATUS*(k) messages and it is an empty pool, $j < k < c+$. This process number j will correspond to e or c, whichever applicable, as indicated in 1 and 2 above. Hence, by using *Status_Pool*(j), P_{c+} can either re-use $ORDER_j(start\_o_j)$, if any provided in one of the *AckHistory*s or inspect all *AckHistory*s to gather the *ORDER*s sent by P_j.

5.9 Retrieval Procedure

Process p_i may need to retrieve some $ORDER_c(o)$ from signalled coordinator P_c, which it had not accepted and is also not included in the *TransferHistory* of the received/prepared $START_{c+}(c)$ message. Hence every $ORDER_c(o)$, $o < CC_{mx}$, which has not been accepted by p_i needs to be retrieved. Therefore p_i multicasts a *REQUEST*(o) for every such $ORDER_c(o)$. Every correct process p_j, having had $ORDER_c(o)$ in its accepted messages, unicasts *REPLY*(o) to p_i containing $ORDER_c(o)$. p_i is guaranteed to get at least (f_c+1) distinct *REPLY*(o) messages because of the reasons explained in subsection 5.6.1 (see Task 5).

5.10 Discussions

This section provides additional insight to some aspects of Install part by discussing the rationale behind them.

5.10.1 Why only one *Status_Pool*(c) is sufficient to construct $START_{c+}(c)$?

A Coordinator P_c, $c \neq 1$, is not considered installed at a correct process p_i until $ORDER_c(start\_o_c)$ sent by P_c gets committed at p_i (lines I6.4-I6.8 in fig 5.2). This is when p_i enters configuration Σ_c and starts acknowledging any further *ORDER* messages sent by P_c. This procedure can cause three possible situations.

– **Situation 1:** P_c was installed at p_i successfully

  Outcome: $STATUS_i(c)$ multicast in response to FS_c will contain *proof_of_commitment* for $ORDER_c(max\_committed_i)$ in *HistoryProof*,

$max\_committed_i \geq start\_o_c$. If $max\_committed_i > start\_o_c$, $STATUS_i(c)$ contains *ORDER* messages sent by P_c during its regime.

That $max\_committed_i = start\_o_c$ represents a situation when no $ORDER_c(o)$, $o > start\_o_c$, sent by P_c was committed after the installation of P_c. If P_c had sent any *ORDER* message and if p_i had acknowledged any of them, then $AckHistory_i$ will contain all those acknowledged *ORDER*s. In this case, *HistoryProof* reflects the fact that *ORDER*s sent by predecessor coordinator P_{c-}, $c- < c$, have been committed. p_i had entered Σ_c.

- **Situation 2:** P_c fail-signalled during installation. p_i received FS_c when it had only acknowledged but not committed $ORDER_c(start\_o_c)$.

  Outcome: $STATUS_i(c)$ multicast in response to FS_c will contain $\langle ORDER_c(start\_o_c), ACK_i(start\_o_c)\rangle$ pair in $AckHistory_i$. This will be the only pair included in $AckHistory_i$ as no $ORDER_c(o)$, $o > start\_o_c$, sent by P_c could have been acknowledged by p_i. p_i had not entered Σ_c. Note that it is the only situation where a non-empty $STATUS_i(c)$ message indicates that P_c was not installed successfully and only an attempt was made.

- **Situation 3:** P_c fail-signals and FS_c reaches p_i even before it acknowledges $ORDER_c(start\_o_c)$.

  Outcome: $STATUS_i(c)$ multicast in response to FS_c will be empty. p_i had not entered Σ_c.

The three situations above show that a non-empty $STATUS_i(c)$ message carries information about last committed and all acknowledged but uncommitted *ORDER* messages sent by P_c. If there was no *ORDER* sent by P_c *after* its installation, a non-empty $STATUS_i(c)$ message carries information about all acknowledged but uncommitted *ORDER* messages sent by the predecessor of P_c (by having $ORDER_c(start\_o_c)$). Hence every non-empty $STATUS_i(c)$ message covers all acknowledged but uncommitted *ORDER* messages that p_i has at the time of transmission, if any. Since the main objective of Install part is to ensure that all those acknowledged *ORDER* messages that could have been committed at some correct process are taken care of when system moves from one configuration to the next, $Status\_Pool(c)$ is enough to gather this information. If this is found empty, which is the case for situation 3, the next coordinator needs to find out non-empty *Status_Pool* corresponding to the closest predecessor of P_c, which can provide this information.

Hence only one (the latest one) non-empty *Status_Pool* is needed by the next coordinator to construct *START*.

5.10.2 Why (*f*+1)-*START*?

This subsection highlights the need for (*f*+1) signatures to form (*f*+1)- $START_{c+}(c)$ and discusses why a doubly-signed $START_{c+}(c)$ is not sufficient. We first note the following two points.

A. $START_c(k)$ message, when included in a $STATUS_i(c)$ message is highly relied upon by an *eligible*() coordinator during installation to cover all acknowledged but uncommitted *ORDER* messages by predecessor coordinators (as mentioned in subsection 5.10.1)

B. Due to assumption 2A (subsection 4.1, chapter 4), Signalled state of an FS process is no longer a terminal state. A Byzantine faulty FS process with both constituent processes having failed, can plant a seemingly correct but actually corrupted doubly-signed message (see subsection 3.6.2.3 in chapter 3).

Say, *eligible*() = *c*- and P_{c-} is executing Install. Assume that phases III and IV of the Install protocol (see figure 5.1) responsible for collecting signatures and preparing (*f*+1)-$START_{c-}(k)$ are not executed and that phase V is instead used to commit the 2-signed $START_{c-}(k)$ prepared in phase II. Let us consider a situation when P_{c-} sends doubly signed $START_{c-}(k)$ while taking over as the coordinator, which is acknowledged by some acceptors. Furthermore, it fail-signals before $START_{c-}(k)$ could be committed at any correct process. P_c being *eligible*() starts installation process but fail-signals even before generating a $START_c(c-)$ message. As in A above, next *eligible*() process P_{c+} will rely on $STATUS_i(c-)$ messages containing $START_{c-}(k)$ to construct a $START_{c+}(c-)$. As in B above, P_{c-} could have become Byzantine faulty and can produce plant for $START_{c-}(k)$ which can cause multiple conflicting $START_{c-}(k)$ messages or even a single planted $START_{c-}(k)$ to appear in *Status_Pool*(*c*-). Planted $START_{c-}(k)$, if considered by P_{c+}, may cause violation of major objective of Install i.e. P_{c+} should not re-use any *o* of *ORDER*(*o*, *r*) issued by $P_{c'}$, $c' < c+$, for any $r' \neq r$, if *ORDER*(*o*) could have become committed at some correct replica (subsection 5.3). To avoid this catastrophic situation, $START_{c-}(k)$ needs to be made fool-proof which can only be done by getting it signed by more than f_{c-} processes. Hence due to A and B above, specially to survive successive coordinator failures, having an (*f*+1)-*START*, becomes necessary.

5.10.3 Garbage collection of Status_Board

We know that *Status_Board* is maintained by all acceptor processes containing a number of *Status_Pools*. We also note that a *Status_Pool* is solely used by an *eligible*() coordinator during construction of *START* message. As pointed out in subsection 5.10.1, for a process $P_i \notin$ *Signalled*, only one non-empty *Status_Pool*(j) is sufficient to complete this task, where j is the largest integer smaller than i for which *Status_Pool*(j) is non-empty. Hence P_i does not need to maintain any *Status_Pool*(k), $k < j$, in its *Status_Board*.

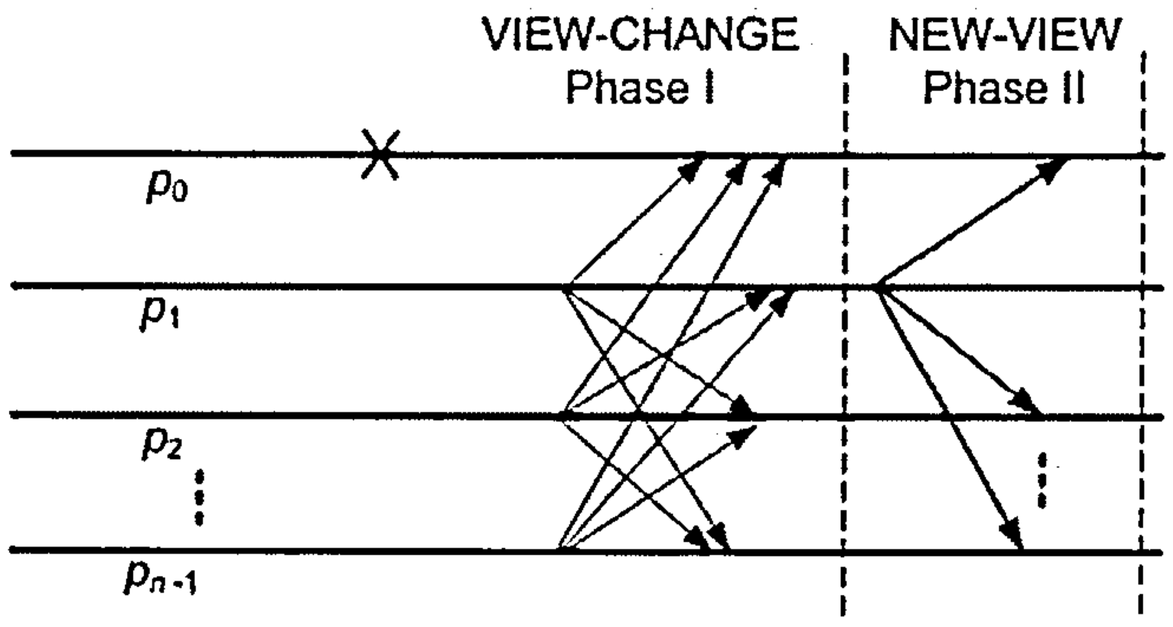
Optimization Remark: It may be noted that since *Status_Board* is only used by potential coordinators, every process $p_i \in \{\pi \cup \pi' \setminus \{\Pi \cup \{p_{f+1}\}\}\}$ does not need to maintain a *Status_Board* at all. This is considered an optimization which results in another optimization to reduce number of message transmissions by making all acceptor processes to send *STATUS* messages only to potential coordinators.

5.11 Performance Comparison

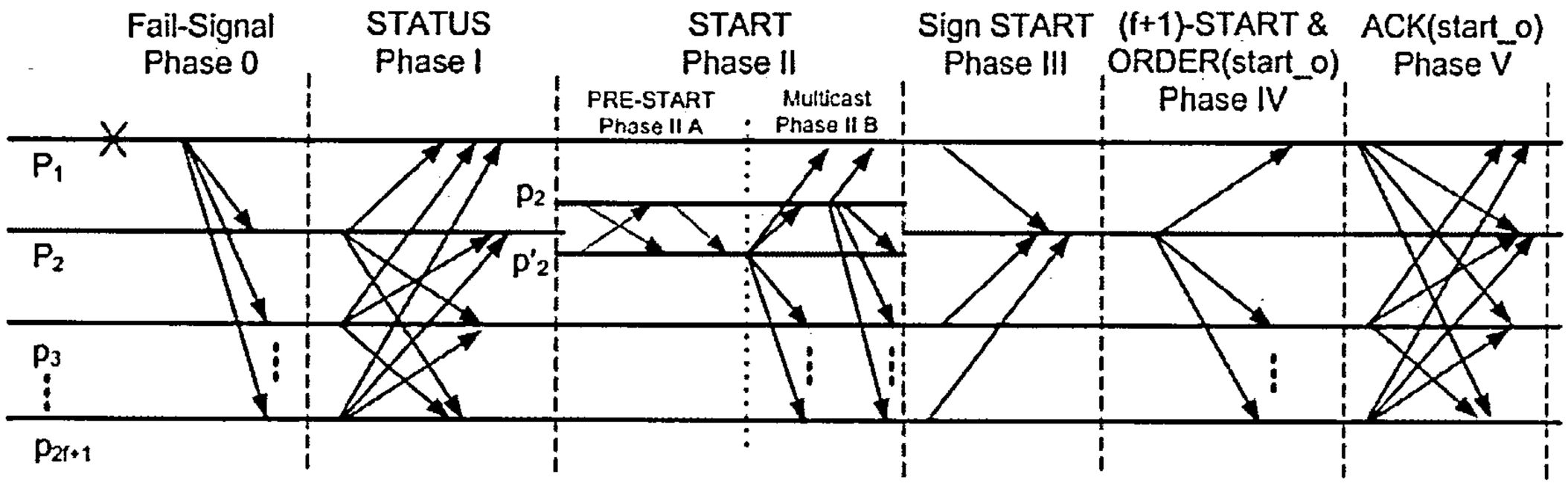
This performance study follows from Study II of chapter 4. We use the same network setup and WAN Emulator to measure and compare the performance of Protocol-I and BFT when a failure occurs. This is to measure the time these two protocols take to switch from one configuration or view (for BFT) to the next. This comprises of the time during which no requests are being processed (ordered) by the system. We call this time *Fail-Over Latency* and describe it more formally for both protocols below.

5.11.1 Defining Fail-Over Latency

If we compare Install part of Protocol-I with that of BFT (in BFT it is actually called View-Change part, but we will refer to it as Install for simplicity), we find that the two are rather dissimilar for a comparative study. The way Install is triggered and the point when it is terminated i.e. when control is transferred back to Normal part is different for the two protocols. This difference makes it particularly difficult to come up with a single definition of Fail-Over Latency which could be easily applied to both protocols. We reproduce the two-phase diagram for Install part of BFT from chapter 2 along with the multi-phase diagram of Install part of Protocol-I in figure 5.6 below.



(a)



(b)

Figure 5.6. Install part of the two protocols (a) BFT (b) Protocol-I

Looking at the two diagrams (figure 5.6), one may draw analogies between BFT phase I and Protocol-I phase I, and also between BFT phase II and Protocol-I phase IV. But actually there exist the following subtle differences among them.

Difference 1: Phase I of BFT is triggered at any process p_i autonomously when a timer timeouts. This is when p_i suspects its coordinator. Whereas in Protocol-I, it is triggered by the receipt of a fail-signal. Depending upon the timer and buffer values, suspicions by various processes can be far away from each other in time. But since a fail-signal is diffused into the system by a correct process, there exists a time frame during which every correct process would initiate this phase. That is, all correct ones may execute phase I of Protocol-I isochronously.

Difference 2: End of phase II of BFT declares the new coordinator to be installed, which is not the case yet for phase IV of Protocol-I. Here, the new coordinator is considered installed only at the end of phase V. Comparing phase II of BFT with phase V of Protocol-I, ignoring the difference in the number of message transmissions, we find that although these two phases mark the end of installation process, post-installation activities for both protocols are quite different. End of phase V in Protocol-I

marks the start of Normal part execution. For BFT although execution goes back to normal part and the system starts ordering new requests after phase II, but the processing of some older order messages coming from previous views is also continued. Hence depending upon the size of this backlog of old messages, BFT may not be able to perform as in normal case for some time. We refer this time as *Pre-Normal Phase* (depicted in fig 5.7). On the other hand in Protocol-I, Install part includes processing of this backlog and only fresh requests are dealt with once phase V ends.

Due to these differences, it was found difficult to identify similar *start* and *stop* points for the measurements of Fail-Over latency of the two protocols. We therefore summarize the execution switch between the two parts for both protocols in a simplistic timeline format in figure 5.7 and formally define Fail-Over Latency in terms of precisely what events it includes.

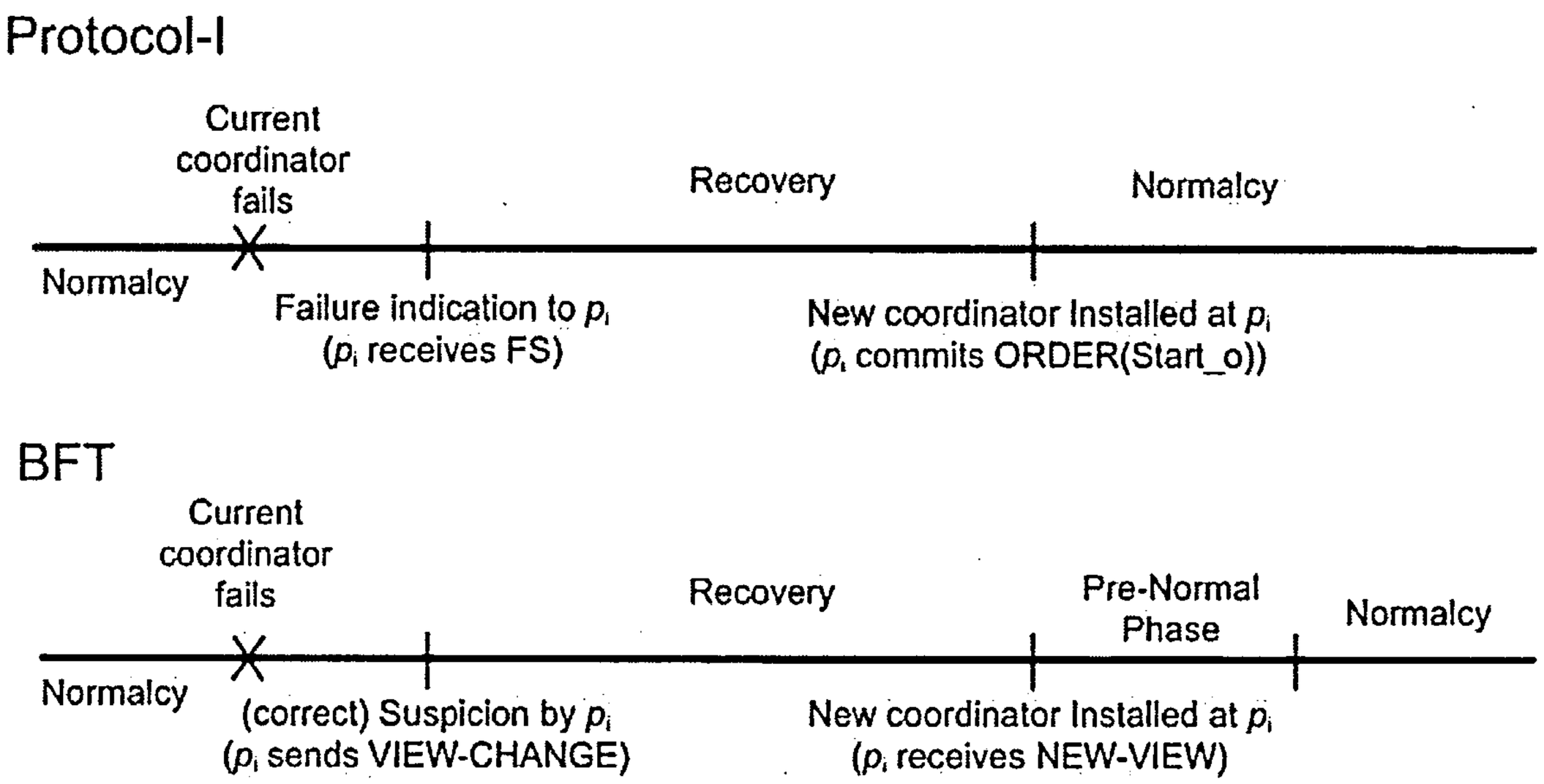


Figure 5.7. Timeline for events occurring during execution switch for the two protocols

The time period labelled as Recovery in figure 5.7 defines the fail-Over latency and is measured at every process p_i in the system. It can be defined as follows.

Fail-Over latency for BFT: It is the time interval between the moment a process p_i starts correctly suspecting current coordinator i.e. generates *VIEW-CHANGE* message and the instance it considers the new-coordinator as installed i.e. receives *NEW-VIEW* message.

Fail-Over latency for Protocol-I: It is the time interval between the moment a process p_i comes to know that current coordinator has failed i.e. it receives fail-signal and the

instance when it considers the new-coordinator as installed i.e. commits *ORDER*(*start_o*).

Formally defining fail-Over latency for the two protocols does not completely solve the issue of trying to setup identical environment for the latency measurement of the two protocols. Next subsection investigates the effect of various other parameters choices on this measurement.

5.11.2 Selecting parameter values

The purpose was to keep the latency measurement environment as identical as possible for the two protocols. While designing the experiments following questions were addressed

1. When should the fault be injected?

   A fault needed to be injected which can trigger transmission of fail-signal by the current coordinator in case of Protocol-I and transmission of *VIEW-CHANGE* message (suspicion) in BFT. The question arises that at what point should this be done. End of normal part execution for a request *r* i.e. commitment of *ORDER*(*o*, *r*) was an identical choice which marks the end of last phase of normal part for both protocols (see figure 4.5, chapter 4 for the three phase diagram). Selection of *o* was another issue. While the performance of Install part of Protocol-I is not effected by this selection, that of BFT may well be. This is because BFT includes all *ORDER* messages acknowledged since the latest stable checkpoint in *VIEW-CHANGE* message. For simplicity, let us assume that every checkpoint becomes stable immediately. This means that the bigger the difference between *o* and the last checkpoint number (the latest stable checkpoint) is, the bigger the size of the *VIEW-CHANGE* message will be. However Protocol-I includes all *ORDER* messages acknowledged since the latest committed *ORDER* i.e. *max_committed* in *STATUS* message. Since commitment is not a process that runs periodically like check pointing in BFT, instead it runs continuously in parallel to the ordering process, Protocol-I is not affected by the selection of *o*. Hence to give benefit to BFT we choose *o* to be the same as a checkpoint number. This will essentially keep the log size in *VIEW-CHANGE* to minimum which can even be 0. For Protocol-I, fail-signal is generated as soon as *ORDER*(*o*) gets committed. For BFT, *VIEW-CHANGE* is generated as soon as *CHECKPOINT*(*o*) gets stable.

2. What *batching_interval* to choose?

   It was decided to test the two protocols in normal load situation. This is because heavy load pushes the system into saturation and time measurement of a one-off event in that situation cannot be fully trusted. Hence we choose *batching_interval* values from stable region only.

5.11.3 Experiment Results

This subsection presents fail-over latency values measured during various experimental setups. f takes values of 1 and 2 and both protocols were run for the two emulated WAN settings described in chapter 4. Experiment results are divided into three sets of readings corresponding to the measurements taken in three weeks. Every point in the graphs shown is an average over 5 readings. We first present the overall average of all three weeks to present the bigger picture, followed by individual readings from each set. Since these experiments could not be repeated hundreds of times as was done in Order latency measurements (chapter 4) due to long setup time involved, we also present the error index found in each set to give a complete picture.

Table 5.1 shows the fail-over latency figures for both protocols with $f = 1$ using batching interval of 1500 msec. The readings are those measured at p_4 (p_3 for Protocol-I, see Table 4.0 for general identification scheme). The reason for choosing p_4 is that no other process among all four has a neutral role of a non-coordinator process in both protocols. For BFT coordinator role switches from p_1 to p_2, hence p_3 and p_4 are the non-coordinator processes participating in the execution. For Protocol-I, P_1 (p_1 and p'_1) fail-signals and p_2 takes over as the new coordinator. Hence only p_3, the last of the four processes, is the only non-coordinator normal process.

Table 5.1 shows that Protocol-I performs better than BFT in both network configurations. This is due to the fact that since p_2, the taking over coordinator in Protocol-I, is the last coordinator and a non-FS process, it skips phases IIA, III and IV of Install part (shown in figure5.6(b)). Phase IIA is only executed by an FS process and therefore cannot be executed by p_2. Since all f failures have already occurred, *START* produced by p_2 does not need to be secured by (f+1) signatures (as explained in subsection 5.6.3), hence the reason for skipping phase III and IV.

Network Configuration	Fail-Over Latency (msec) for p_4	
	Protocol-I	BFT
Fast WAN	201.3167	293.1333
Slow WAN	397.4833	1027.267

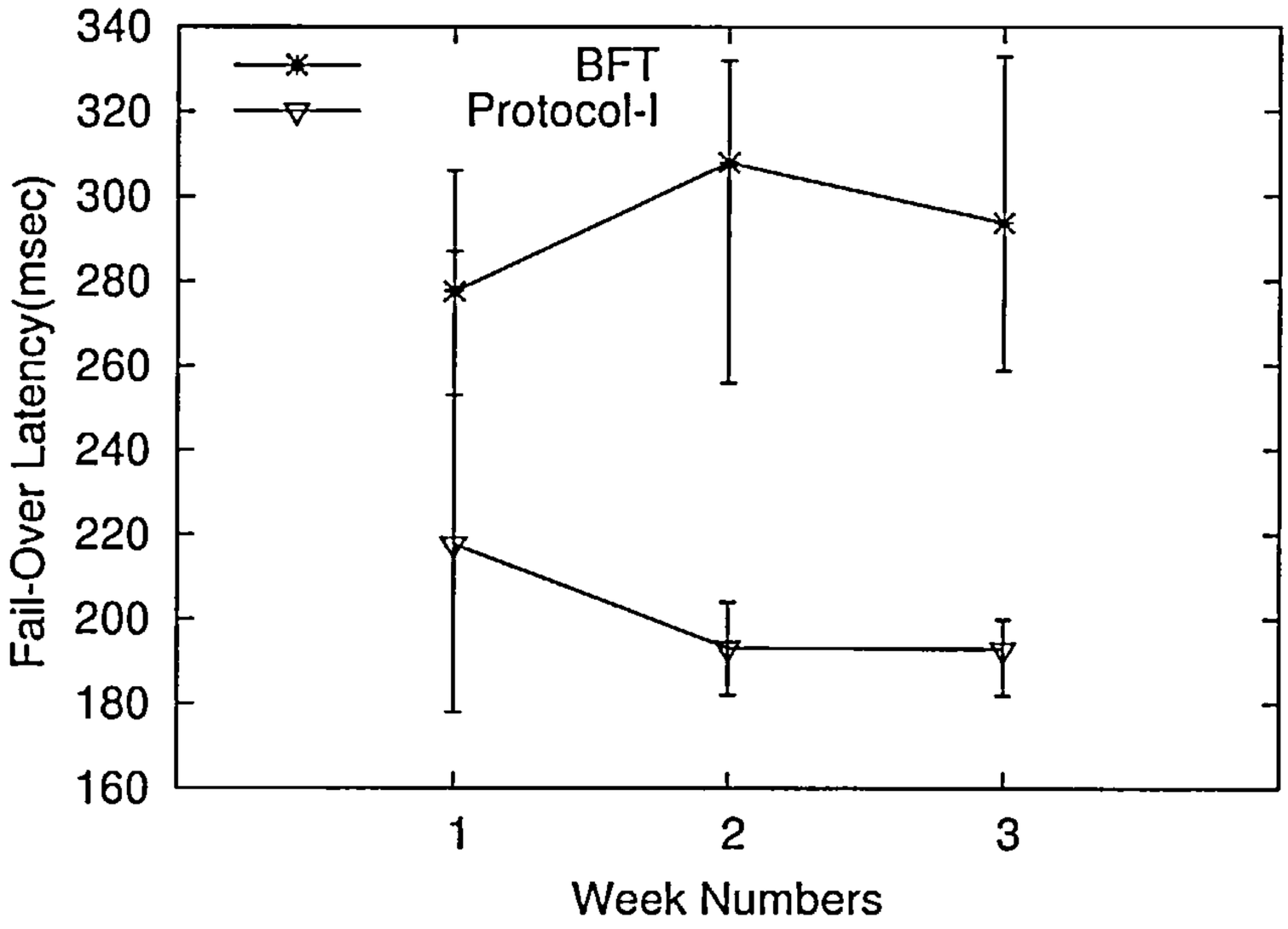
Table 5.1. Fail-over latency at p_4 for both protocols with $f = 1$ using Batching_interval = 1500 msec

Table 5.2 shows the fail-over latency figures for both protocols with $f = 2$ measured for one of the stable batching intervals of the corresponding network configurations. The readings are those measured at p_3 (p_2 for Protocol-I) and p_7 (p_5 for Protocol-I). The reason for choosing these processes is the same as described in chapter 4 i.e. to present figures for one of the paired and non-paired processes each. The results below clearly show that BFT outperforms Protocol-I. This is in accordance with the difference in the number of phases of communication executed in the two protocols as was shown in fig 5.6. However it is interesting to see that the performance gap decreases as a slower network configuration is tested. This is because when moving from a fast network configuration to a slower one, while all phases of communication presented in figure 5.6 for both protocols suffer high latency, phase IIA of Protocol-I remains same. Also latency in phase V of Protocol-I does not increase proportionally due to presence of faster links among the much slower ones.

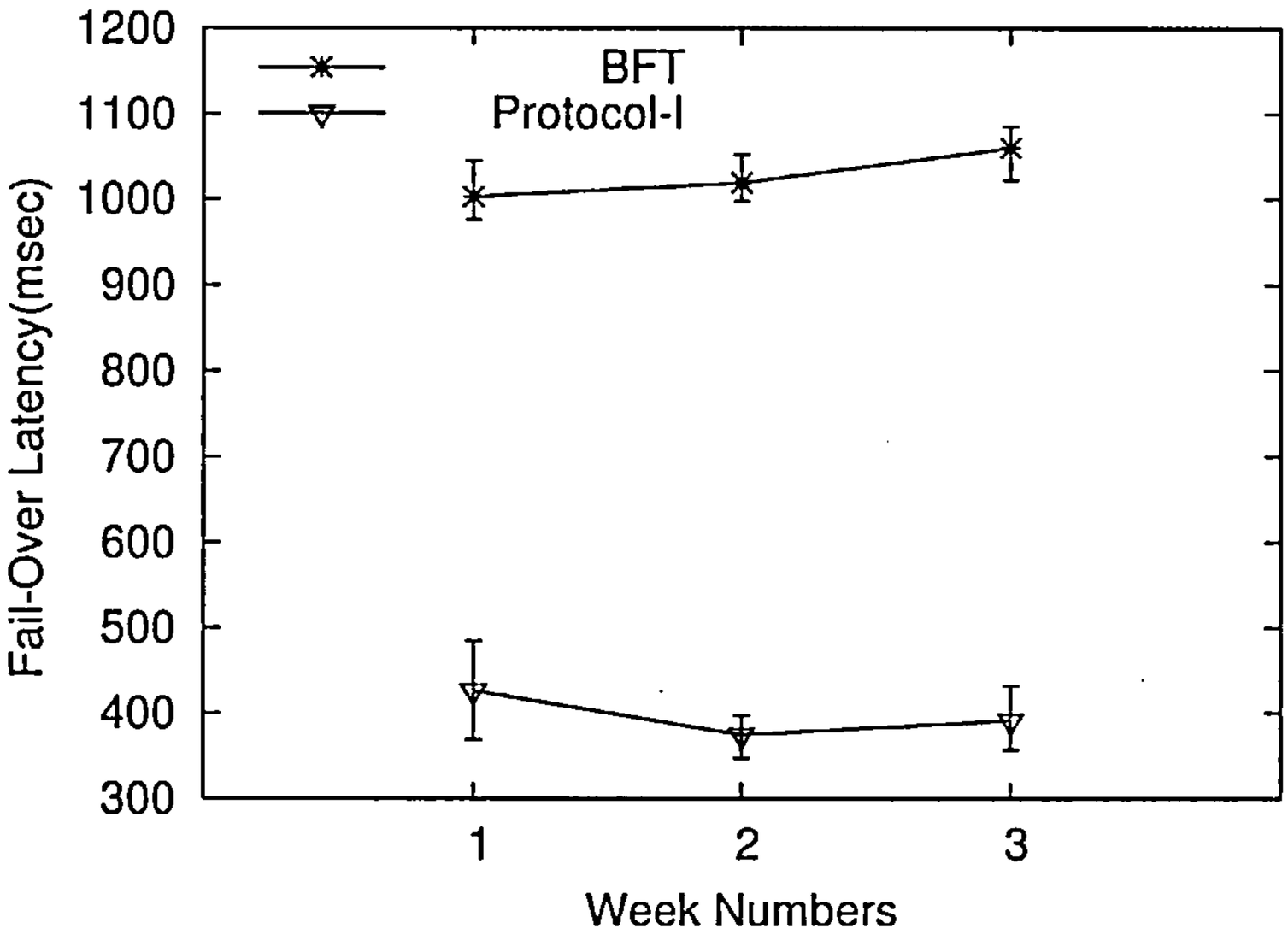
Network Configuration	Batching Interval (msec)	Fail-Over Latency (msec)			
		p_3		p_7	
		Protocol-I	BFT	Protocol-I	BFT
Fast WAN	1500	1264.5	284.0	1100.2	448.3
Slow WAN	2500	2967.4	1243.3	2516.2	1979.3

Table 5.2. Fail-over latency at p_3 and p_7 for both protocols with $f = 2$ using a stable Batching_interval

We present the individual set readings of fail-Over latency with respect to their week numbers with same batching intervals (as above) in graphical form below (Fig 5.8 to 5.10). These graphs also show the error index for each set.

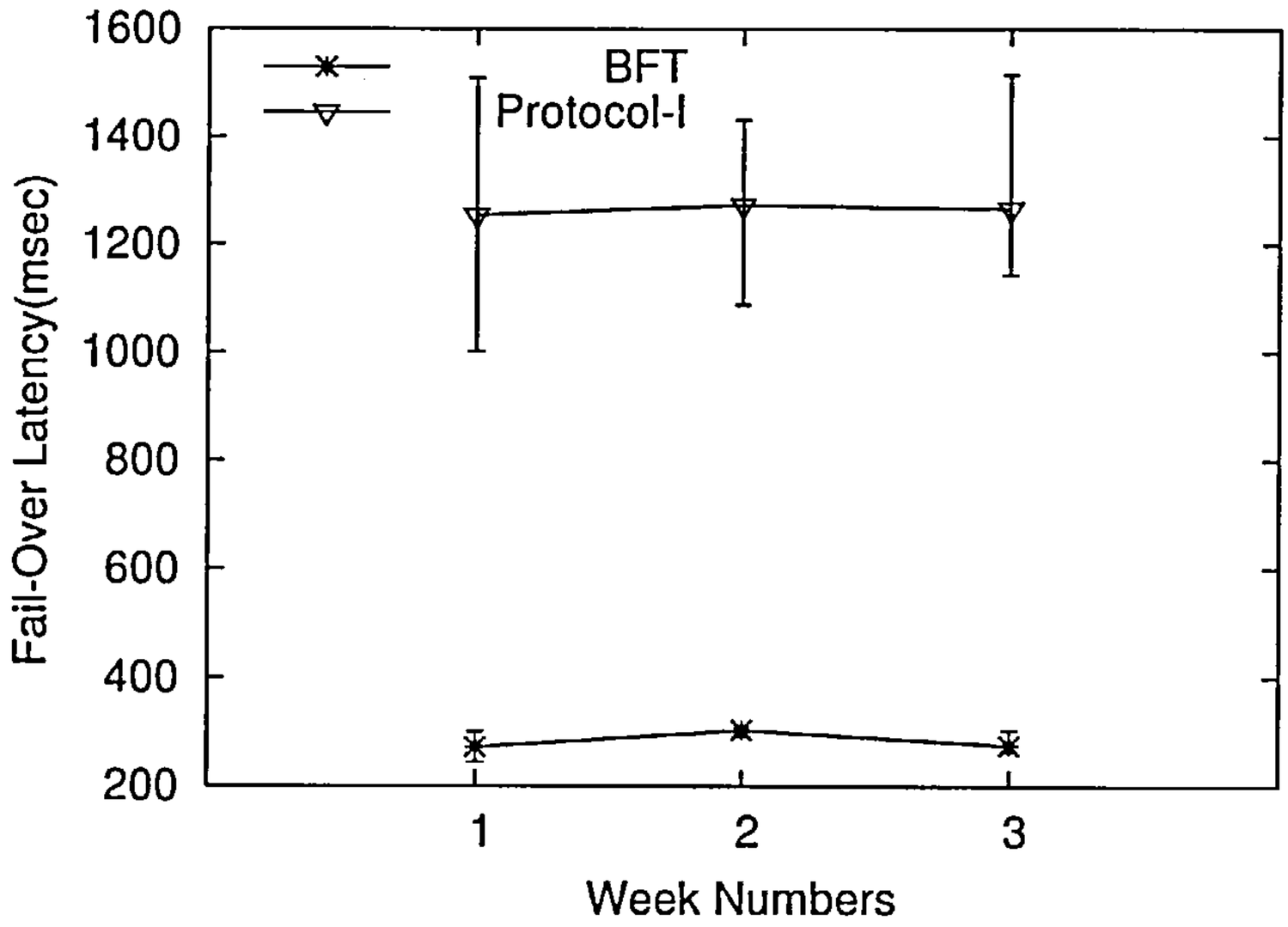


(a) Fast (Inter-city) WAN

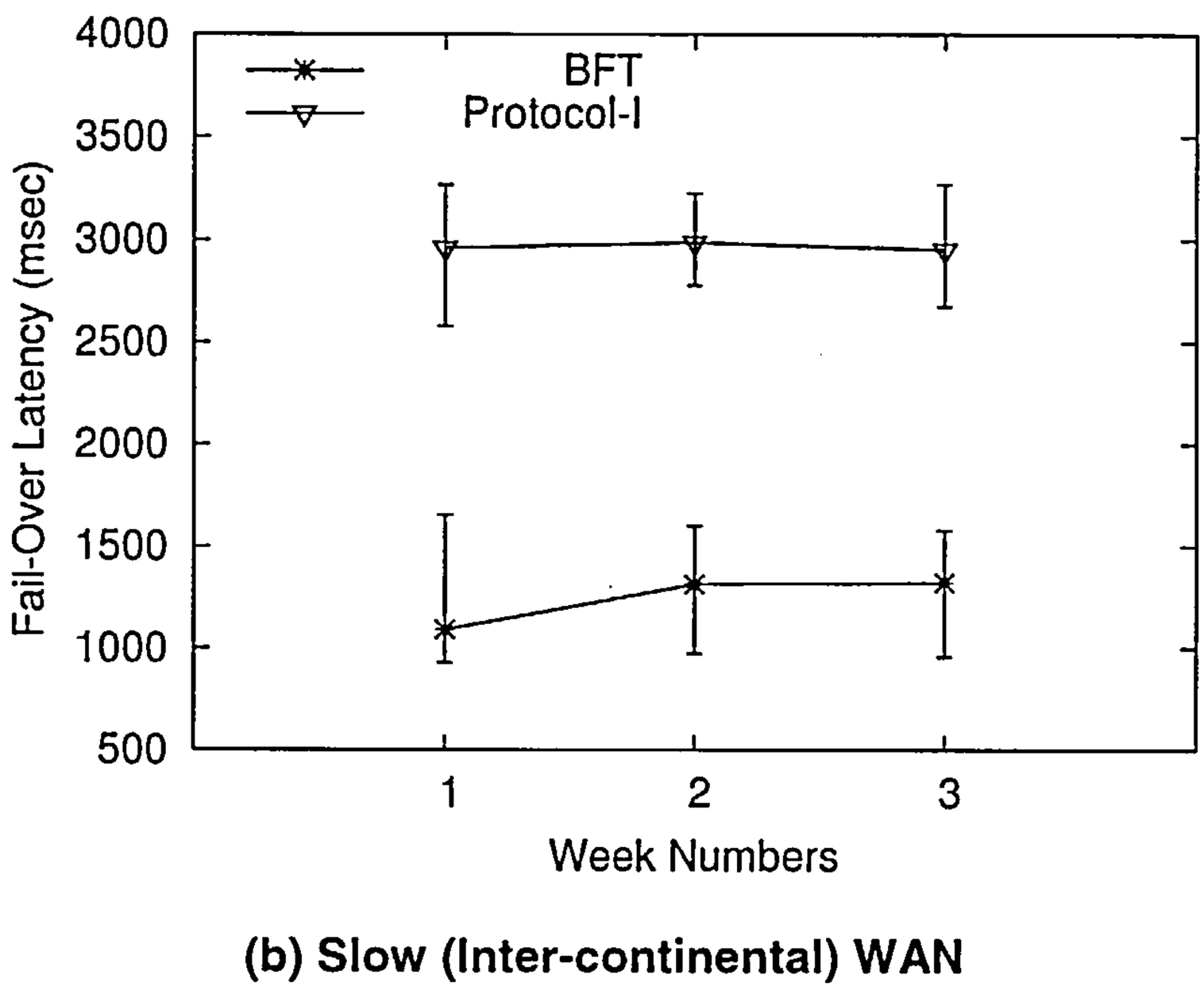


(b) Slow (Inter-continental) WAN

Figure 5.8. Fail-over latency at p_4 with $f = 1$ using various network configurations

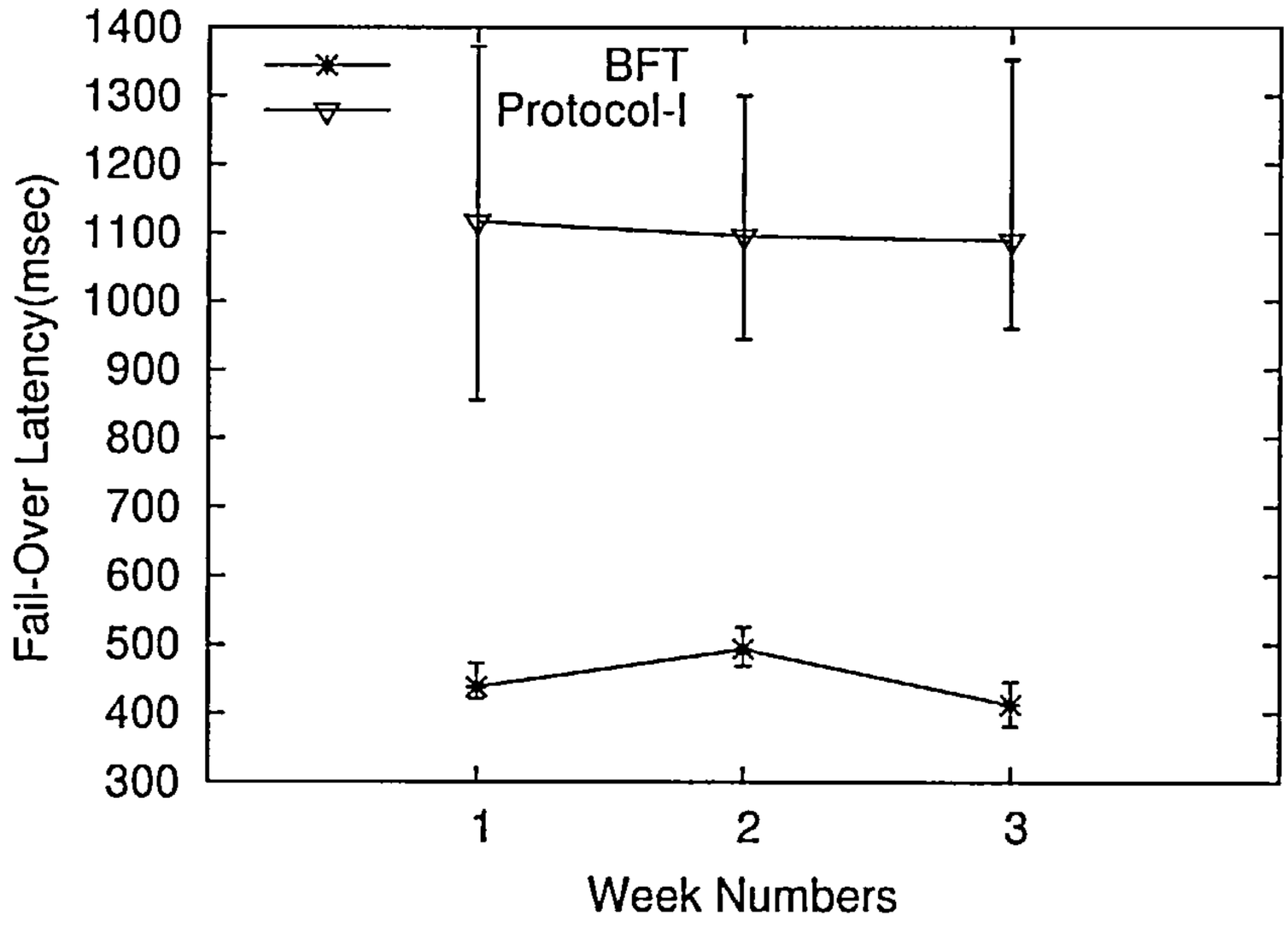


(a) Fast (Inter-city) WAN

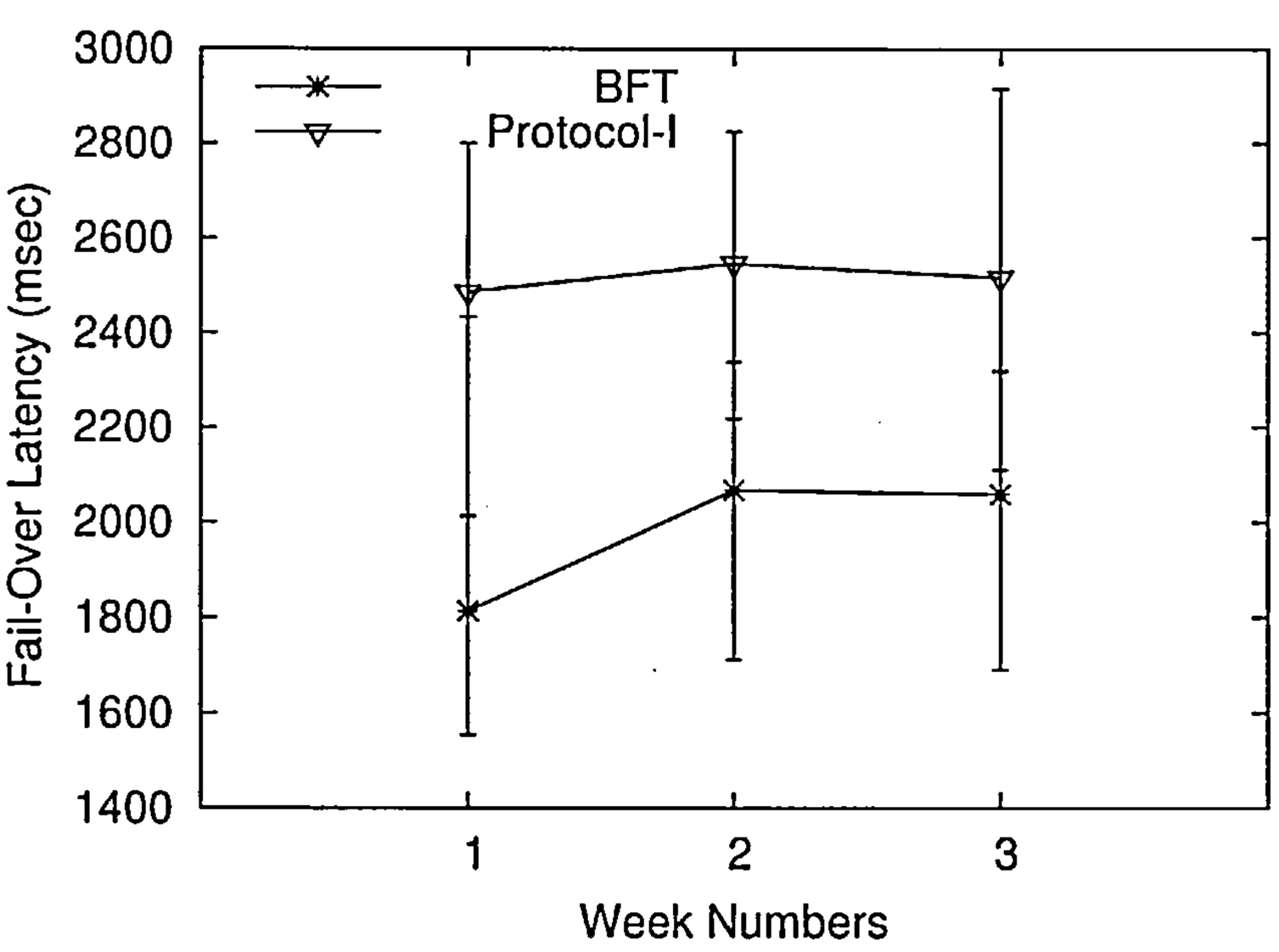


(b) Slow (Inter-continental) WAN

Figure 5.9. Fail-over latency at p_3 with $f = 2$ using various network configurations



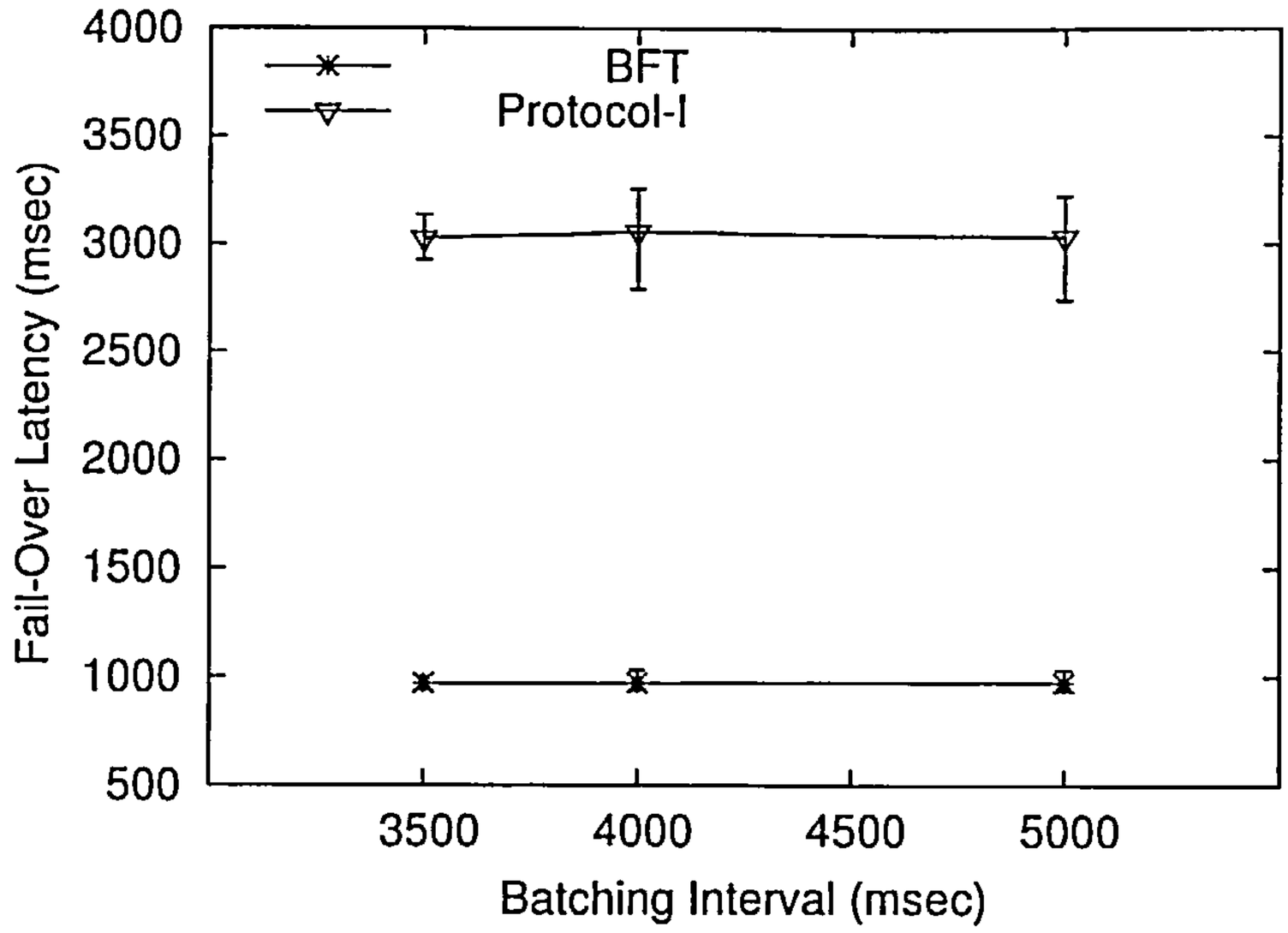
(a) Fast (Inter-city) WAN



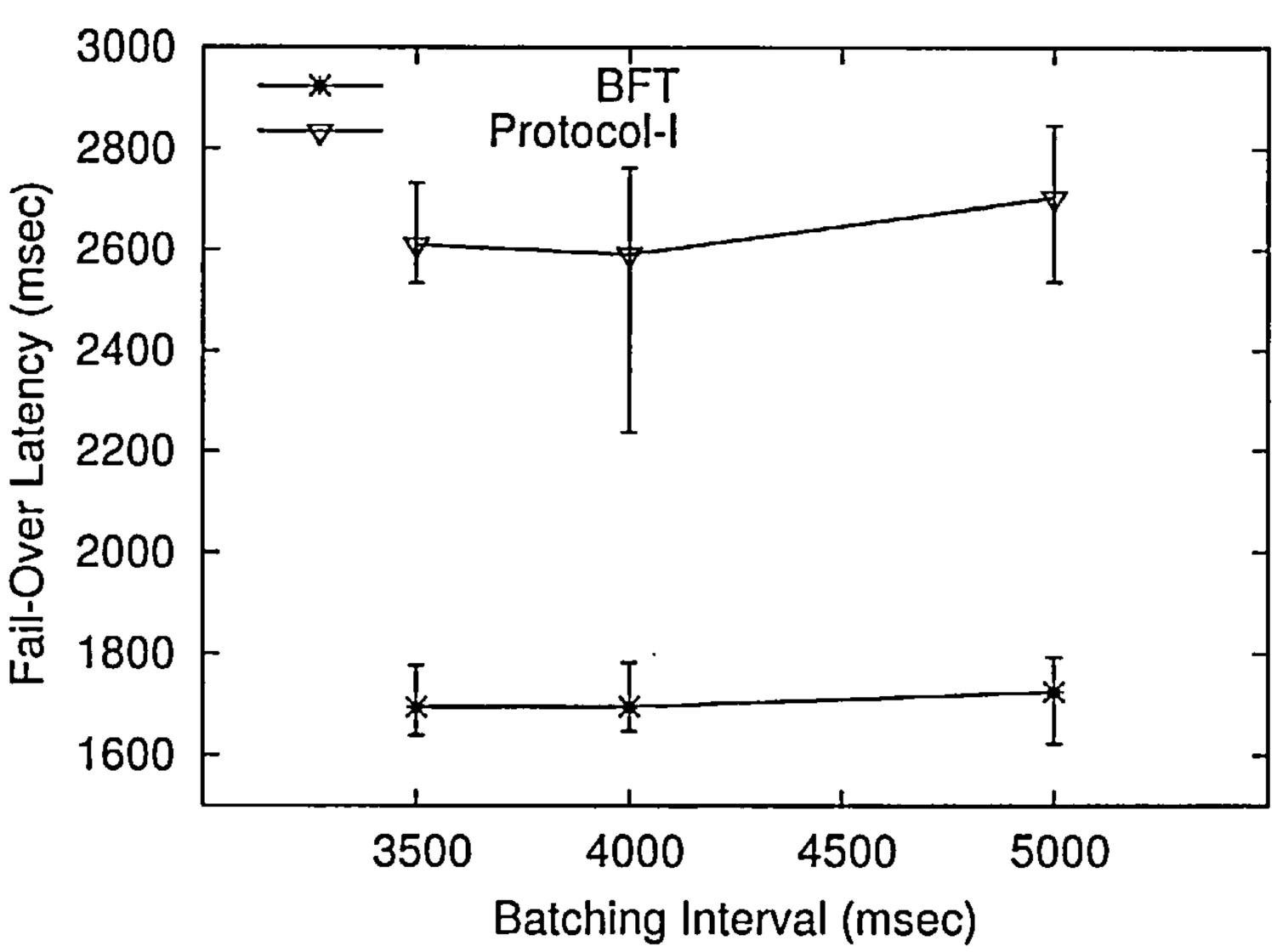
(b) Slow (Inter-continental) WAN

Figure 5.10. Fail-over latency at p_7 with $f = 2$ using various network configurations

The following two graphs (Fig 5.11) show that when the experiment was repeated for more batching interval values in stable region, fail-over latency was found to remain stable. This extended set of readings was taken in week 4.



(a) at p_3



(a) at p_7

Figure 5.11. Fail-over latency vs. Batching Intervals with $f = 2$ using Slow WAN at various processes

5.11.4 Observations

Results of the experimental study presented in above subsection quantify the performance gap between the two protocols due to difference in number of

communication phases (two for BFT and five for Protocol-I, see Fig 5.6). The observations can be summarized as follows.

- BFT mostly outperforms Protocol-I considerably.
- Protocol-I manages to reduce the performance gap in slower asynchronous network settings by making use of fast synchronous links.

As explained in section 5.10, each step executed in Install part is significant. Hence it is clear that the performance gap between the two protocols cannot be improved by possibly reducing the number of phases of communication executed by Protocol-I in this part. We therefore try to optimize Install part of Protocol-I by reducing the constant overhead size of a crucial message (*STATUS* message) exchanged during its execution. In other words, we try to compensate the overhead caused by the increased number of message communication phases by reducing the size of the largest message exchanged. The optimized version of Install is called Install-II. The following section discusses the execution steps of Install-II in detail.

5.12 Install-II – Optimized version of Install Part

Before introducing Install-II steps, we describe a few minor changes in Normal part of the protocol that are needed to implement Install-II. They are as follows.

1. *commitable*() that is defined in subsection 4.4.2 of chapter 4, is now redefined as follows: An *ORDER*(o) is said to be committed i.e. the predicate *committable*(o) is true at *any* p_i if the following conditions are true
   1. *ORDER*(o) is *accepted* locally.
   2. *ORDER*(o'), $o' \geq o$, is known to have been acknowledged by a quorum.

   Note that local acknowledgment of *ORDER*(o) by p_i is no longer a requirement for the commitment of *ORDER*(o); once condition 2 above is met, simple acceptance will suffice. Since $max\_committed_i$ is the largest o committed by p_i, every time an *ORDER*(o) is committed, it is set to maximum$\{o, max_committed_i\}$.

We note that while A_i, the largest o for which p_i has sent an *ACK*(o), increases only sequentially (due to the acknowledgment condition (iii) in subsection 4.4.2), since local acknowledgment of *ORDER*(o) by p_i is no longer a requirement for the commitment of *ORDER*(o), $max\_committed_i$ may not. For instance, if message transmission between the coordinator/client and p_i is unduly slow, p_i can receive

$ACK(o)$ from Q_c distinct replicas before it is able to acknowledge the $ORDER(o)$ it had received. Consequently, p_i can have $A_i < max\_committed_i$.

2. Every process p_i multicasts its $max\_committed_i$ in every $ACK_i(o)$ i.e., structure of $ACK_i(o)$ becomes $\langle ACK, c, o, D(ORDER(o)), i, max\_committed_i \rangle$.
3. Every process p_i additionally maintains a list called $Commit\_Board_i$. $Commit\_Board_i$ is a list of the largest $max\_commited_j$ value received so far by p_i in any $ACK_j(o)$ message from every process p_j in the system. Let $Commit\_Board_i[j]$ represents the largest $max\_commited_j$ value received. Hence p_i sets $Commit\_Board_i[j]$ to maximum$\{Commit_Board_i[j], max_commited_j\}$ every time it receives an $ACK_j(o)$ message.

As described earlier, Install-II aims to reduce the size of the largest message; in particular it avoids sending *HistoryProof* in the $STATUS_i(c)$ message. *HistoryProof* is a collection of Q_c messages and is a constant overhead included in all $STATUS_j(c)$ messages. Since the construction of $START_{c+}(c)$ in Install relies on this proof, some extra computational steps need to be taken in Install-II to cope with the absence of information.

Install-II has the same execution steps as Install and the protocol skeleton presented in figure 5.1 remains same. Changes are only needed in construction steps of $STATUS_i(c)$ and $PRE\text{-}START_{c+}(c)$. We describe these changes in the following subsections. We once again take the simple context of execution with P_c as current coordinator, P_j as the signalled FS process and P_{c+} as the *eligible*() process.

5.12.1 Construction of $STATUS_i(c)$

$STATUS_i(j)$ message is multicast by process p_i in response to receiving a fail-signal from P_j, $j \geq c$, where c is the rank of current coordinator. For $j > c$, $STATUS_i(j)$ prepared by p_i is empty. Hence, for simplification we only consider $j = c$. However, one exceptional case is when $c = -1$ and P_j being *eligible*() started executing Install but fail-signalled at least after completing first four phases. In this situation $ORDER_j(start\_o_j)$ multicasts by P_j, can get acknowledged by p_i and hence, $STATUS_i(j)$ would include this order. We will generally refer to P_c as signalled coordinator but will mention P_j explicitly to indicate the above mentioned exceptional case, when required.

p_i first computes low water-mark, denoted as LW_i. LW_i is defined as the largest element in $Commit\_Board_i$ which is less than or equal to at least Q_c elements.

***Claim 1*:** for a correct p_i, $LW_i \leq max\_commited_j$ of at least $(Q_c - f_c)$ correct p_j.

***Reason*:** There are at least Q_c processes whose *max_commited* values are larger than or equal to LW_i, of which at most f_c will be faulty.

LW_i can be easily computed as: rank all n_c values of $Commit\_Board_i$ in the non-increasing order (largest value ranked first) and choose LW_i to be the Q_c^{th} ranking value.

p_i then constructs $STATUS_i(c)$ message which contains

a. Received *fail-signal* message: FS_c,

b. $AckHistory_i$: List of $\langle ACK_i(o), ORDER_c(o)\rangle$ pairs for every acknowledged $ORDER_c(o)$ in the range

   i. $minimum\{LW_i, A_i\} \leq o \leq A_i$, if $c = 1$, or

   ii. $maximum\{start\_o_c, minimum\{LW_i, A_i\}\} \leq o \leq A_i$, if $c \neq 1$

c. Values of $max\_commited_i$, A_i and LW_i.

***Note1*:** b and c will be null when $j > c$. Furthermore, for the exceptional case mentioned above when $c = -1$ with $eligible() = j$ and $ORDER_j(start\_o_j)$ is acknowledged by p_i then $AckHistory_i$ produced in response to FS_j will be a singleton list containing $\langle ACK_i(start\_o_j), ORDER_j(start\_o_j)\rangle$.

***Note2*:** In b(ii) above, both $start\_o_c$ and $minimum\{LW_i, A_i\}$ will always be less than or equal to A_i. The latter is clearly true. For the former, the reason is given as follows. Firstly, by definition, $AckHistory_i$ contains acknowledged *ORDER*s only. Since there is no pre-condition for acknowledgement of $ORDER_c(start\_o_c)$, if this *ORDER* is acknowledged, A_i would have been updated accordingly (See line I5.6 in figure 5.3) and $A_i \geq start\_o_c$. However, if $ORDER_c(start\_o_c)$ is not acknowledged, the only reason can be the receipt of a fail-signal from P_c, in which case $ORDER_c(start\_o_c)$ would have already been discarded from the *Order_Pool* automatically.

5.12.2 Construction of $PRE\text{-}START_{c+}(c)$

Let P_{c+} be the *eligible*() FS process. Both p_{c+} and p'_{c+} prepare *PRE-START* independently as was done in Install. For the sake of simplicity, we will only describe it for p_{c+} here. p_{c+} computes the source number s as described in section 5.8. Let $s = c$ and *Status_Pool*(c) be the non-empty pool containing $STATUS_i(c)$ messages from at least a quorum Q_c. First step to construct $PRE\text{-}START_{c+}(c)$ is to compute CC_{mx} and PC_{mx} for *TransferHistory*, as defined in subsection 5.1.1, and then construct $START_{c+}(c)$ and the corresponding *ProofBag*. CC_{mx} and PC_{mx} are computed in the following way.

CC_{mx}: The *max_committed* values received from distinct processes are ordered in the non-increasing (largest first) manner and CC_{mx} is set to be the $(f_c+1)^{th}$ value in the ordered list.

***Claim 2*:** $ORDER(CC_{mx})$ is certainly committed by some correct process

***Reason*:** Since at most f_c of the largest *max_committed* values can be from malicious processes, $(f_c+1)^{th}$ value is guaranteed to be committed by some correct process.

However, if p_{c+} receives $ORDER_c(start\_o_c)$ in any $STATUS_i(c)$ message and $start\_o_c > CC_{mx}$ then it sets $CC_{mx} = start\_o_c$. This is to re-use already computed *START* message as highlighted in subsections 5.1.1 and 5.8. By definition, $ORDER_c(start\_o_c)$ corresponds to $(f+1)$-*START*, which contains *TransferHistory*. Since, *TransferHistory* is a list of *ORDER*s in $[CC_{mx}, PC_{mx}]$, claim 2 will still hold.

PC_{mx}: The A_i values received from distinct processes are ordered in the non-decreasing manner (smallest first) and PC_{mx} is set to be the Q_c^{th} value in the ordered list.

***Claim 3*:** Any $ORDER(o)$, $o > PC_{mx}$ is certainly not committed by any correct process

***Reason*:** Since PC_{mx} is the Q_c^{th} smallest A_i value, it is the highest A_i value among a quorum, say q_1. Let us assume to the contrary that $ORDER(o)$, $o > PC_{mx}$, is committed by a correct process after receiving acknowledgements from quorum q_2. Since quorums intersect, at least one correct process in q_1 would have contributed to this commitment and will have $A_i \geq o$. Hence, PC_{mx} cannot be smaller than o as assumed. Thus, p_{c+} knows with certainty that no correct process could have committed any $ORDER(o)$, $o > PC_{mx}$, while it is taking over the coordinator role.

***Lemma 1*:** A correct coordinator-designate always has $CC_{mx} \leq o \leq PC_{mx}$.

Proof: CC_{mx} is the $(f_c+1)^{th}$ ranked *max_committed* value from the top i.e. less than or equal to *max_committed* value of at least one correct process – $ORDER(CC_{mx})$ is certainly committed by a correct process – certainly at least $(Q_c - f_c)$ correct processes sent $ACK(CC_{mx})$ – so, in any set of at least Q_c *STATUS* messages, a correct p_{c+} will have at least one correct process p_i whose $A_i \geq CC_{mx}$. If A_i is the largest of the smallest Q_c A-values, PC_{mx} will be equal to A_i, otherwise $PC_{mx} \geq A_i$. Hence $CC_{mx} \leq PC_{mx}$.

Algorithm executed by p_{c+} to construct $START_{c+}(c)$ message is given as *Construct_Start*(c) in figure 5.12. We use *size_of*(*Status_Pool*(c)) to denote the number of *STATUS* messages in *Status_Pool*(c).

```
Construct_Start(c)
{
1.1   #S = Qc - 1;
1.2   Repeat
      {
1.3   Wait for size_of(Status_pool(c)) > #S;
1.4   #S = size_of(Status_pool(c));
1.5   } until STATUS_Set_found(c) = true;
1.6   Repeat ∀ ORDER(o) ∈ Status_Pool(c), CCmx ≤ o ≤ PCmx
      // Construct TransferHistory and ProofBag
      {
1.7   Perform spuriousness check;
1.8   Perform conflict resolution using all received STATUS messages
      in Status_Pool(c)
      }
}
STATUS_Set_found(c)
{
2.1   Compute CCmx;
2.2   Compute PCmx;
2.3   if (∀ o: CCmx ≤ o ≤ PCmx, ∃ ORDER(o) ∈ Status_pool(c)) →
2.4         return true;
2.5   else  return false;
}
```

Figure 5.12. *START* Construction Procedure

p_{c+} starts its attempt to construct $START_{c+}(c)$ after receiving $STATUS_i(c)$ messages from a quorum only. If it cannot find some $ORDER(o)$, $CC_{mx} \leq o \leq PC_{mx}$, it waits for more $STATUS_i(c)$ messages to arrive. Since values of CC_{mx} and PC_{mx} may vary with the arrival of new $STATUS_i(c)$ messages in *Status_Pool*(c), p_{c+} re-computes these bounds every time it attempts $START_{c+}(c)$ construction. Once p_{c+} finds the continuous stream of *ORDER* messages, it starts constructing *TransferHistory* and *ProofBag*. Proof for CC_{mx} forms the *Base* of *ProofBag*. Note that p_{c+} and p'_{c+} are synchronized for the messages they receive (see ISIQ in figure 3.3, chapter 3). Hence, this proof can simply comprise of the process ids of the senders of the *STATUS* messages used for CC_{mx} computation. Algorithmic details behind lines 1.7 and 1.8 in figure 5.12 are the same as mentioned in subsection 5.7.1 i.e., p_{c+} adds every $ORDER(o)$, $CC_{mx} \leq o \leq PC_{mx}$, in *TransferHistory* and corresponding proof in *ProofBag*. *commit_count* is also used to help construction of *TransferHistory* in the same way.

Exchange of $PRE\text{-}START_{c+}(c)$ is followed by reconstruction of $START_{c+}(c)$ in the way described in subsection 5.7.2.

5.13 Correctness proof for Install-II

This section presents the proof for Install-II that it satisfies the safety and liveness requirements.

***Background*:** At any moment, let the ordered list $\{X_r: 1 \leq r \leq R$ and $(Q_c - f_c) \leq R \leq n_c\}$ denote the *max_committed* values of *all* correct processes arranged in the non-increasing order (largest first). That is, $X_r = max\_committed_i$ of some unique and correct p_i and $X_1 \geq X_2 \geq \ldots \geq X_R$.

Claim 4: for all k, $1 \leq k \leq (Q_c - f_c)$ and for any correct p_i, $LW_i \leq X_k$.

Reason: By claim 1, there are at least $(Q_c - f_c)$ correct processes whose *max_committed* values are not smaller than LW_i. The minimum of the *max_committed* values held by these correct processes must be no larger than $X_{(Qc - fc)}$. Therefore, $LW_i \leq X_{(Qc - fc)}$. That is, $X_1 \geq X_2 \geq \ldots \geq X_{(Qc - fc)} \geq LW_i$. Since, $Q_c \geq (2f_c+1)$ and $(Q_c - f_c) \geq (f_c+1)$. Hence, $X_{(fc+1)} \geq X_{(Qc - fc)} \geq LW_i$.

Note that even if correct p_i computes LW_i using 'stale' values of $max\_committed_j$, claim 4 holds as *max_committed* values of correct processes, hence $X_1, X_2, \ldots, X_{(Qc - fc)}$, cannot decrease with passage of time.

Let $\Lambda\omega_{mx}$ and A_{mx} respectively denote the largest of *LW* and *A* values used by correct processes when they prepared their *STATUS* messages.

Lemma 2. Once a correct coordinator-designate p_{c+} receives *STATUS* messages from all correct processes, it is guaranteed to compute $CC_{mx} \geq \Lambda\omega_{mx}$ and $PC_{mx} \leq A_{mx}$.

Proof: Since p_{c+} has *max_committed* values from all the correct processes, CC_{mx} it computes will be one of $X_1, X_2, \ldots, X_{(fc+1)}$, irrespective of the *max_committed*-values reported by the faulty ones i.e., CC_{mx} cannot be smaller than $X_{(fc+1)}$, $CC_{mx} \geq X_{(fc+1)}$. By claim 4, LW_i computed by any correct p_i cannot be larger than X_k, $1 \leq k \leq (Q_c - f_c)$ i.e., $X_k \geq LW_i$ or simply $X_{(fc+1)} \geq LW_i$. Since $\Lambda\omega_{mx}$ is computed by a correct process, $X_{(fc+1)} \geq \Lambda\omega_{mx}$ and hence, $CC_{mx} \geq \Lambda\omega_{mx}$.

PC_{mx} is computed as the Q_c^{th} smallest *A* value in the *Status_pool*, there can be f_c or more processes whose $A_i \geq PC_{mx}$. Even if all faulty ones report A_i larger than A_{mx}, PC_{mx} cannot be larger than A_{mx}; so $PC_{mx} \leq A_{mx}$.

Theorem 1 (*Liveness*): Once a correct coordinator-designate p_{c+} receives *STATUS* messages from all correct processes, it is guaranteed to have an *ORDER*(o) for every o, $CC_{mx} \leq o \leq PC_{mx}$.

***Proof*:** By lemma 2, p_{c+} has $\Lambda\omega_{mx} \leq CC_{mx}$ and $PC_{mx} \leq A_{mx}$ and by lemma 1, $CC_{mx} \leq PC_{mx}$, which implies $\Lambda\omega_{mx} \leq CC_{mx} \leq PC_{mx} \leq A_{mx}$.

Consider the correct process with $A = A_{mx}$. By the definition of $\Lambda\omega_{mx}$, this process will have its $LW \leq \Lambda\omega_{mx}$; but, $\Lambda\omega_{mx} \leq A_{mx} = A$; so, it will have $minimum\{LW_i, A_i\} = LW \leq CC_{mx}$. Since $A_{mx} = A \geq PC_{mx}$, this correct process will have in its *STATUS* message, a conflict-free or conflict-resolvable *ORDER*(o) for every o, $CC_{mx} \leq o \leq PC_{mx}$. Since p_{c+} has received *STATUS* messages from all correct processes, it has at least one *ORDER*(o) for every o, $CC_{mx} \leq o \leq PC_{mx}$. Hence the theorem is proved.

Theorem 2 (*Safety*): When a correct coordinator-designate p_{c+} terminates *STATUS* message processing, none of the $ORDER_c(o, r)$ it selected for any o, $CC_{mx} \leq \ o \leq PC_{mx}$, will conflict with $ORDER_c(o, r')$, $r \neq r'$, committed by some correct process.

***Proof (by contradiction)*:** Let us assume that $ORDER_c(o, r)$ conflicts with $ORDER_c(o, r')$, $r \neq r'$, committed by some correct process. Since both order messages are doubly-signed and authentic, then both p_c and p'_c have failed and, by assumption IB, at least $2D$ time has elapsed subsequent to the first of these failures has been observed. Therefore if, $ORDER_c(o, r')$, is committed by some correct process, then p_{c+} will already have received *STATUS* messages from at least all correct processes with at least (Q_c - f_c) of them having included $ORDER_c(o, r')$, and only at most f_c processes having included $ORDER_c(o, r)$. For p_{c+} to have selected the planted $ORDER_c(o, r)$, it must have ignored at least one of the (Q_c - f_c) *received STATUS* messages containing $ORDER_c(o, r')$. But p_{c+} examines *all STATUS* messages received prior to termination for conflict resolution. Hence this is a contradiction and the assumption cannot be true.

5.14 Install vs. Install-II

In this section, we present a qualitative comparison of Install of section 5.4 and Install-II. We highlight the major differences in the execution steps involved and study the costs and benefits of using Install-II over Install. We will refer to Install as Install-I for uniformity. P_c and P_{c+} has the meaning of signalled and *eligible*() coordinator respectively, whereas p_i (or suffix i for that matter) is used to represent any process in the system.

Recall that *HistoryProof* and *AckHistory* in a *STATUS* message and *TransferHistory* in a *START* message are the main factors responsible for the large sizes of these messages in Install-I. Moreover, Install-II aims to reduce message size of

STATUS message by dropping *HistoryProof*. Hence, the following two benefits can be evidently gained by using Install-II in place of Install-I.

(i) Size of $STATUS_i(c)$ message reduces by Q_c *ACK* messages.

(ii) Performance improves as time needed to sign $STATUS_i(c)$ message decreases for smaller size message.

Depending on the size of individual *ACK* messages, the gain achieved by Install-II can be significant for higher values of Q_c. However it is important to see if there is any cost behind these advantages.

We analyse the effect of the optimization proposed in Install-II on the size of other components i.e., *AckHistory* and *TransferHistory*. We observe that the size of these lists is proportional to the extent to which the progress of A_i and $max\_committed_i$ in any process p_i is synchronized with that of any other process. For example, let us assume that p_i is receiving all *ORDER* messages instantly and is able to acknowledge them straight away while network connection for the remaining processes is slow which results in delays in acknowledgements. Since p_i cannot commit an $ORDER(o)$ without receiving $ACK(o)$ from at least a quorum, the difference ($A_i - max\_committed_i$) tends to be bigger. Hence, on receiving a fail-signal from P_c, p_i will generate large $AckHistory_i$. Similarly, *TransferHistory* will also be respectively large. On the other hand, if processes are synchronized naturally due to symmetric network connections, these lists are more likely to be smaller in size. Observing the influence of network connectivity on the working of Install-I, we classify the network environment into two categories and compare the two algorithms with respect to these.

Case A - Symmetric network conditions: This refers to the case where almost all processes are connected to each other via links of similar characteristics. Due to this homogenous nature, processes progress symmetrically and are naturally synchronized with respect to A_i and $max\_committed_i$ values.

Case B - Asymmetric network conditions: This is the case where network links between various processes are heterogeneous. Hence, A_i and $max\_committed_i$ values of any process p_i may be far apart from those of any other.

Let us first analyse the behaviour of Install-I and -II in first system configuration. As mentioned in case A, A_i and $max\_committed_i$ values of almost all processes will be quite close to each other, if not exactly same. For simplicity let us assume that these values are equal for all processes in the system. Obviously, for Install-I, $AckHistory_i$ produced by every process will be same with messages in the range

[$max\_committed_i$, A_i]. Largest of all $max\_committed_i$ (and A_i) will be same as any $max\_committed_i$ (and any A_i). This will lead to a *TransferHistory* containing messages in the range [$max\_committed_i$, A_i]. Install-II execution will also have the same effect. That is, since all $max\_committed_i$ values are equal, LW_i will be calculated to be same as $max\_committed_i$ and hence, $AckHistory_i$ and *TransferHistory* produced in Install-II will be same as that in Install-I. This shows that in a symmetric environment the two benefits listed for Install-II are simply retained.

Case B represents heterogeneous links between processes, possibly causing gaps between the A_i and $max\_committed_i$ values of various processes. For Install-I, the *AckHistory*s produced by various processes will include different number and range of messages within the respective $max\_committed_i$ and A_i values. However, the lower bound on order number in $AckHistory_i$ for Install-II is LW_i rather than $max\_committed_i$. Recall that LW_i is computed by p_i as the Q_c^{th} highest $max\_committed_j$ value received from every other process p_j. Hence, at least (Q_c - f_c) correct processes will have $LW_i \leq max\_committed_i$ and will produce larger $AckHistory_i$ than produced in Install-I. *TransferHistory* produced in Install-I will comprise of messages starting from the largest $max\_committed_i$ to the largest A_i found in a quorum. Whereas the same for Install-II will have messages from $(f_c+1)^{th}$ largest $max\_committed_i$ to the largest A_i found in a quorum. Hence, Install-II has a smaller lower bound for *TransferHistory* than Install-I which causes increase in the size of *TransferHistory*. This shows that Install-II achieves reduction of constant overhead size of *HistoryProof* by a possible increase in the size of *AckHistory* and *TransferHistory* when communication takes place in asymmetric environment. Moreover, lack of synchronization may leave P_{c+} encountering holes in the first Q_c *AckHistory*s received i.e., some of the messages in [CC_{mx}, PC_{mx}] may not be available. This will cause some delay as P_{c+} waits to receive messages from more than a quorum to fill in the holes (see lines 1.3 and 1.5 in figure 5.12). Hence, it can be concluded that Install-II may not necessarily be an optimal choice for highly heterogeneous environments.

5.15 Summary

This chapter presented the design and performance analysis of Install part of Protocol-I. Install part deals with the failures signalled by the coordinator FS process and transfers the system from current to next appropriate configuration with a new coordinator. The

main objective of Install is to help the eligible process to take over as the new coordinator in a safe and live manner.

We discussed in detail how the Byzantine state of FS process causes complications in the design of Install. In summary, the signalled coordinator process can become Byzantine faulty and start producing undetectably corrupt double-signed *ORDER* messages. Moreover, other faulty processes in the system can collude to present the corrupted messages to the eligible coordinator as if the *ORDER*s were received and acknowledged before the previous coordinator fail-signalled. Install achieves its objectives by tackling such situations. Moreover, it is also designed to deal with the situation where the eligible process fails during the execution of Install.

Description of the algorithm steps was followed by qualitative and quantitative analysis. Like Normal part, performance of Install of Protocol-I was compared against that of BFT. It was found that Install of Protocol-I incurs high latency as it involves five communication steps against two of BFT. Finally, we tried to optimize Install by reducing constant size overhead of one of the largest messages used. The optimized version, called Install-II, was also compared with Install. We found that Install-II succeeds in reducing the message size in symmetric network environments. However, asymmetric network environments may cause increment in other message components.

Chapter 6

Protocol-II

Protocol-II is a twin of Protocol-I in exploiting fail-signal facility to circumvent the FLP impossibility for solving consensus. The difference between the two protocols is the assumptions underlying the construction of FS processes. We take a new assumption set and design Protocol-II by carefully modifying Protocol-I to cater for the implications due to changes in assumptions.

Recall that the construction of FS process is based on two core assumptions presented in chapter 3. Assumption 1 states that the exact bounds on absolute communication delays between, and relative processing speeds of, any two nodes implementing an FS process are known. Assumption 2 restricts number of failures within an FS process to one. These two assumptions are central to Protocol-0 design. However, the design of Protocol-I relaxes assumption 2 to form 2A which allows both processes to fail, subject to the condition that these failures are sufficiently apart in time. Assumption 1 is kept the same but renamed as 1A. In Protocol-II however, we keep assumption 2 same as in Protocol-0 and adopt a less restricted version of assumption 1 instead.

The structure of this chapter is as follows. We first describe the new underlying assumptions. Then we examine the effects of these assumptions on the behaviour of FS process and discuss the consequences on protocol design. Finally, we show how Protocol-II is derived from Protocol-I by presenting the modifications needed.

6.1 Assumptions

Recall that assumption 1 of an FS process (defined in subsection 3.2.1) is about the ability to accurately estimate a differential delay bound within which one constituent process that has produced an output can expect its counter-part to do the same, if the counter-part is also operating in a timely manner and an output is expected as per the order protocol. Assumption 1 states that this estimate of timing bound within the FS process pair is *always* accurate. Hence, suspicions triggered by timeouts are never false. However, Protocol-II adopts a less restricted version of this assumption and assumes

that an accurate estimate of this bound cannot be guaranteed to hold *always* but only *eventually*. This means that the constituent processes p and p' can falsely suspect each other of untimely behaviour and generate fail-signal. Hence, fail-signal may not always be a true indication of occurrence of a fault. Possibility of these false indications are therefore admitted and dealt with in Protocol-II. We present the assumption set used in Protocol-II below. The relaxed assumption 1 is named as 1B and assumption 2 renamed as 2B (for consistency).

1B. There is a timing instance after which the timeouts used for mutual timeliness checks by order processes p_i and p'_i implementing FS process P_i will always be correct. This timing instance cannot be known a priori.

2B. At least one of the ordering processes p_i and p'_i does not fail.

Note that assumption 1B is weaker than assumption 1A (or 1): the latter assumes the unknown timing instance to be the system initialisation time; assumption 2B, on the other hand, is evidently stronger than 2A.

We start by considering the implications of the new assumption set on FS process behaviour and then on Protocol-I.

6.2 Implications of the New Assumption Set

6.2.1 Status of an FS Process

Recall that a constituent process of an FS process generates a fail-signal on detecting a value- or time-domain fault in its counterpart. The first implication of the new assumption set (due to 1B) is that constituent processes p_i and p'_i may find each other untimely even if both are non-faulty. However, we note that a process in this situation cannot disambiguate whether the counter-part is suffering a failure or the delay estimate turns out to be temporarily inaccurate. Thus, for safety reasons, it must assume the former and generate a fail-signal. Hence, both temporary inaccuracy in timeout estimation and occurrence of fault constitute the causes of fail-signal generation. Obviously, at this point the fail-signalled FS process cannot act as the coordinator; in other words, the constituent processes cannot work in active mode but must revert to passive mode and continue their mutual-checking. If they find each other timely again at a later stage then they can work in active mode again when needed. Therefore, if a fail-signal is generated due to an apparent time-domain failure, the signalled process should

be allowed to act as coordinator if and when the constituent processes subsequently find each other timely .

This implication is captured by the constituent processes p_i and p'_i of an FS process P_i maintaining a status variable $status_i$ which indicates the current operative status of P_i and can take values *up*, *down* or *temporarily_down*. $status_i$ is irreversibly set to *down* when a constituent process of P_i observes a value domain failure of its counterpart.

6.2.2 Change in *Acceptors*

Recalling the configurations and *Acceptors* defined for Protocol-I, we note that the signalled coordinator FS process is considered to have failed and is therefore not an acceptor for the next configuration. That is, when P_i fail-signals in Σ_i, system moves from Σ_i to $\Sigma_{i'}$, $i' > i$ and $P_i \notin Acceptors_{i'}$. With assumption 1B, a fail-signal cannot be attributed to a true failure. Hence, in the above scenario, P_i may be a process with both correct constituent processes, temporarily finding each other untimely. If this is the case, removal of P_i from $Acceptors_{i'}$ will violate the definition of *Acceptors*. This is because removal of an FS process in Protocol-I guaranteed removal of at least one faulty process from the *Acceptors* set which is no longer true for Protocol-II. Hence, the definition of $f_i = f - (i\text{-}1)$ and Q_i will also not be applicable. Moreover, in the worst case, this elimination can result in exclusion of all correct processes leaving behind f faulty processes and 1 correct non-FS process in the system which violates safety. Therefore, in Protocol-II, P_i must be allowed to participate in order assignment as an acceptor (working in passive mode) even after fail-signalling. This implication leads to a change in the definition of $Acceptors_i$, which is elaborated in a subsequent section.

6.2.3 Number of FS processes

The third implication (due again to assumption 1B) is that since every fail-signal may not be indicating a true failure, it can no longer be ascertained that f failures have occurred once f distinct FS processes have fail-signalled. Therefore, if we choose an unpaired process as the $(f+1)^{th}$ coordinator, it cannot be expected to be non-faulty. So, no unpaired process should be trusted for the coordinator role simply because f FS processes have fail-signalled. So, only an FS process can act as the coordinator and the system now requires (at least) $(f+1)$ process-pairs. We add another shadow process p'_{f+1}

to be paired with p_{f+1} to form FS process P_{f+1}. Hence, the system architecture becomes like that of Protocol-0 with the total number of processes $n = 3f+2$ and only paired processes being allowed to act as coordinators.

Note that having (f+1) FS processes does not mean that all (f+1) FS processes cannot fail-signal at the same time. This situation, if it happens, cannot prevail forever. There will be at least one FS process in which both constituent processes are non-faulty and see each other timely starting from some unknown time. So, there is an FS process whose operative status will eventually be always *up*. That is, eventually at least one 'perfect' FS process will emerge, where a perfect FS process is an FS process whose constituent processes are non-faulty and the timing estimates used within it are greater than or equal to the actual delays. The problem is to enable such a perfect process to become the coordinator *when* it emerges. Note also that in the worst case scenario, there may be only one perfect FS process and until that perfect FS process emerges *and* becomes the coordinator, no ordering will take place and the system remains in an unstable state.

6.2.4 Impact on Install

The last implication is due to assumption 2B. Since at most one constituent process of an FS process can fail, the FS process of Protocol-II reverts back to the 3-state model used in Protocol-0 (see figure 3.1). This means that the Byzantine state of an FS process no longer exists. Hence, the signalled coordinator cannot become Byzantine faulty and start producing planted *ORDER* messages which Protocol II does not have to deal with.

This implication results in a modest simplification of the most complex part of the Protocol-I i.e., the Install part. The resulting simplifications are two-fold; construction of *START* and reduction in number of communication phases of Install. Recall that the construction process of *START* message is the major source of complication which is due to its handling spurious and conflicting plants. This process is much simplified in Protocol-II as such planted messages cannot exist due to assumption 2B. However, unlike Protocol-0 and like Protocol-I, Protocol-II is designed to cater for the inconsistent failing state of an FS process. This is achieved by keeping the basic protocol operation same as Protocol-I with passive FS processes and unpaired processes participating as non-coordinators in quorums.

Secondly, the absence of Byzantine state makes phases III and IV of Install of Protocol-I redundant for Protocol-II (see figure 5.1). Recall that these phases deal with

collection of (f+1) signatures and construction of (f+1)-*START*. The purpose of these phases was to make *START* an incorruptible message. In Protocol-II we assume that one of the constituent processes in an FS process always remain correct. Hence, a double-signed message is incorruptible. Hence, Install of Protocol-II reduces to three phases of communications, illustrated in figure 6.1, where last phase is for commitment of *ORDER*(*start_o*) that encapsulates the double-signed *START* instead of (f+1)-*START*.

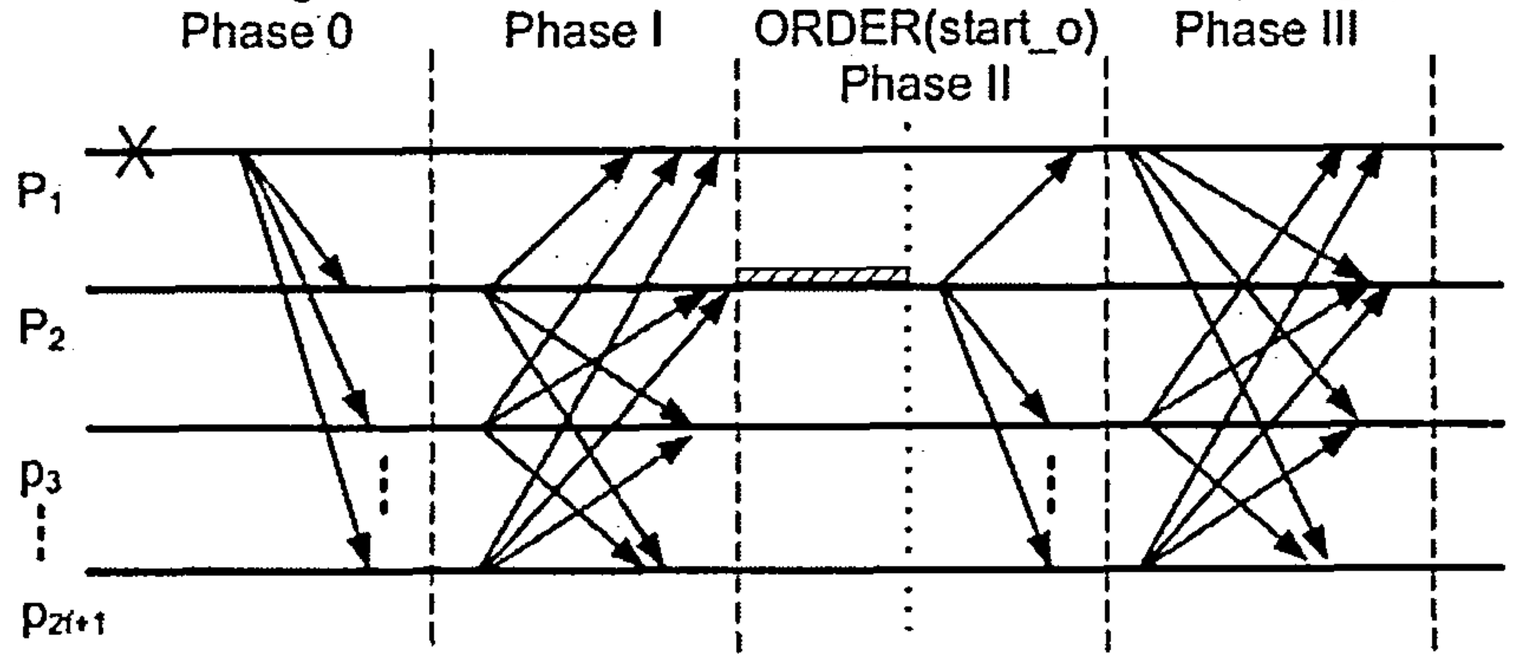


Figure 6.1. Three Communication Phases of Install for Protocol-II

6.3 System Architecture

The system architecture is similar to the one used in Protocol-0, as shown in figure 6.2. The system consists of (2f+1) nodes each of which, as before, hosts service process s_i and order process p_i, $1 \leq i \leq (2f+1)$. Additional (f+1) shadow nodes are added in the system. Each shadow node N'_i hosts shadow process p'_i and is paired with p_i to form FS process P_i executing Protocol-II. As before, process sets are defined as follows.

Π = Set of all FS Processes = $\{P_1, P_2, \ldots, P_{f+1}\}$

π = Set of all order processes co-located with service processes = $\{p_1, p_2, \ldots, p_{2f+1}\}$;

π' = Set of all shadow order processes (not co-located with service processes)
= $\{p'_1, p'_2, \ldots, p'_{f+1}\}$.

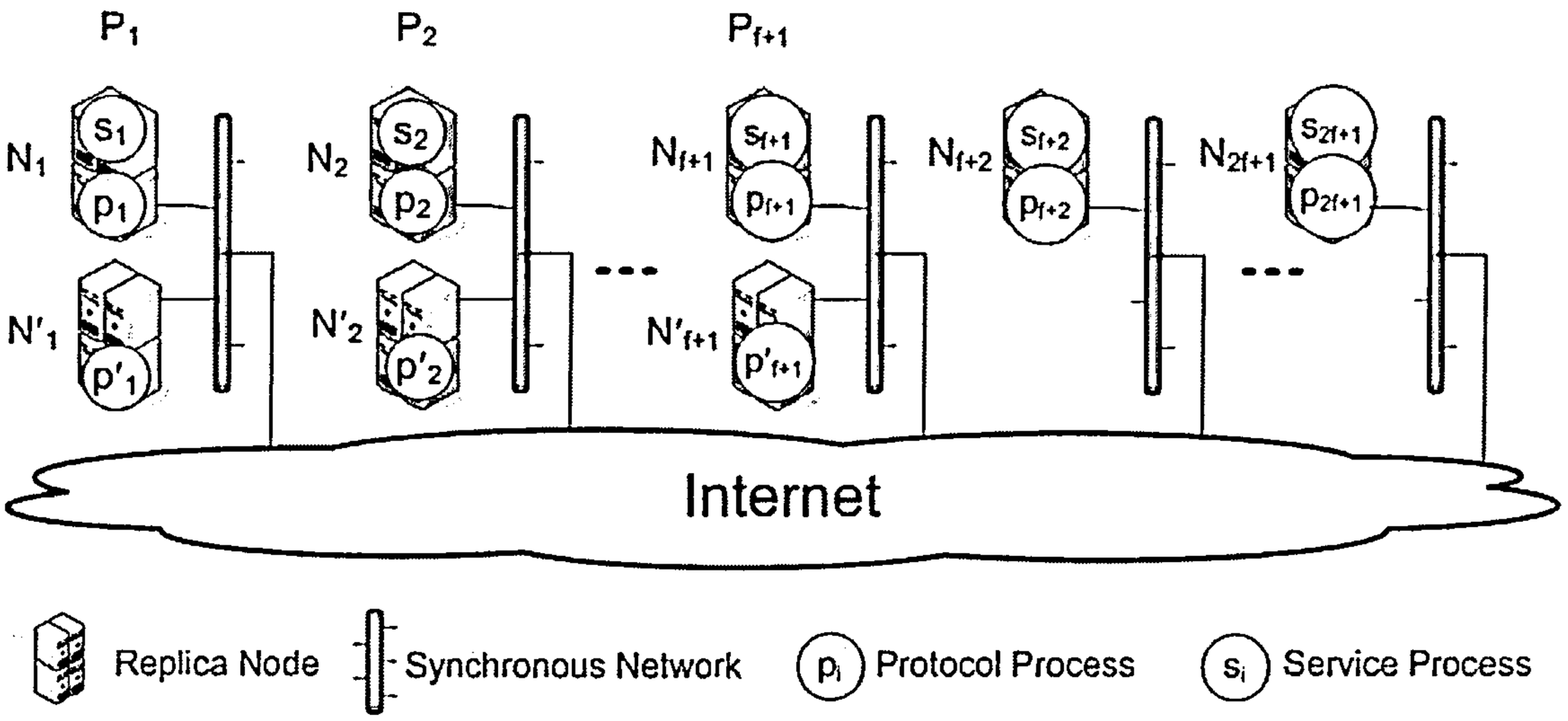


Figure 6.2. System Architecture

6.4 Protocol Design

Protocol-II only allows FS processes to play the coordinator role. Among all (*f*+1) FS processes, only the coordinator FS process works in active mode with remaining acting as passive FS processes. Like Protocol-I, all order processes including the unpaired ones participate in order assignment. As in previous protocols, the protocol is structured into two parts: Normal and Install.

At system start, P_1 is assigned the coordinator role. Normal part is executed by all processes to assign order numbers to clients' requests. Normal part remains exactly the same as Protocol-I (explained in subsection 4.4.2).

When P_1 fail-signals, execution at each process is switched to Install. Install part will attempt to install P_2 as the new coordinator. This may result in either of the following two scenarios.

(i) P_2 is installed as the new coordinator and Normal part execution resumes.

(ii) P_2 cannot be installed as it is down or temporarily down. In either case, installation attempt will be made for P_3.

Given assumption 1B, the possibility of finding all successive FS processes, P_2, …, P_{f+1}, to be down or temporarily down cannot be ruled out. Therefore, the attempt to install an FS process as coordinator must cycle back to P_1 underpinned by rational that a perfect FS process must emerge eventually. In the worst case, if all *f* FS processes have a faulty constituent process, this search for coordinator will continue until the only perfect FS process in the system emerges and is installed. This may take several cycles.

With the assumption about eventual accuracy of timing bounds, the perfect FS process will eventually stabilize with always *up* status and the cyclic attempt at Installation will end.

Thus, under Protocol-II, the system can be regarded to be moving through a series of potential *configurations*. The i^{th} configuration Σ_i, $i \geq 1$, refers to the system configuration with FS process P_c acting as the coordinator for the k^{th} time, $k \geq 1$. Note that the configuration number i is no longer the same as the rank c of the FS process playing the coordinator role – reflecting the possibility of an FS process P_c that once relinquished the coordinator role can again take it up in a later cycle if $status_c$ changes from *temporarily down* to *up* meanwhile. The relation between i and c is given by $c = [(i\text{-}1) \; mod \; (f\text{+}1)] + 1$, $i \geq 1$ and $1 \leq c \leq (f\text{+}1)$. Hence, Σ_i can be expressed as follows.

$$\Sigma_i = \{ p_1, p'_1, \ldots p_{c-1}, p'_{c-1}, P_c, p_{c+1}, p'_{c+1}, \ldots p_{f+1}, p'_{f+1}, p_{f+2}, \ldots, p_{2f+1} \}$$

Note that unlike Protocol-I, the number of configurations that the system can go through is no longer limited to $(f\text{+}1)$ but can be finitely large. That is, system will move from Σ_{f+1} to Σ_{f+2}, from Σ_{2f+2} to Σ_{2f+3} and so on. Moreover, configurations Σ_c, $\Sigma_{c+(f+1)}$, $\Sigma_{c+2(f+1)}$ etc., are identical with P_c acting as coordinator. For example, if $f = 4$, P_5 will be acting as the coordinator in Σ_5, Σ_{10}, Σ_{15} and so on. Also, if P_i becomes (permanently) down before entering into $\Sigma_{i+k(f+1)}$, $k \geq 0$, then attempts to realise $\Sigma_{i+k'(f+1)}$, $k' \geq k$, will not be successful. That is, if P_5 changes its status to *down* before entering Σ_{10}, configurations Σ_{10}, Σ_{15}, Σ_{20},...etc., will be absent in that protocol run.

Now we revisit the definition of other configuration parameters. Due to assumption 1B, FS process can oscillate between *up* and *temporarily_down* status during unstable periods. As perfect FS process cannot be identified, signalled processes should not be eliminated from *Acceptors* (as noted in subsection 6.2.2). Recall that in Protocol-I, only the FS process $P_{c'}$ that succeed the coordinator Process P_c in rank is allowed to participate as acceptor i.e., only for $c' > c$. In protocol-II however, the FS processes that precede the coordinator in the rank order can later become a coordinator and therefore must be retained as acceptors. So, we define $Acceptors_i$ to consist all processes in the system. $Acceptors_i$ simply becomes Σ_i and hence, $|Acceptors_i| = (3f\text{+}2)$, $i \geq 1$. This redefines other configuration parameters as follows:

$$n_i = |Acceptors_i| = (3f\text{+}2) = n$$

$$f_i = f$$

$$Q_i = \lfloor (n_i + f_i) / 2 \rfloor + 1 = (2f\text{+}2)$$

The core concern for design of Install for Protocol-II is to ensure that every process p_i safely moves from Σ_i, $i \geq 1$, to some appropriate $\Sigma_{i'}$, $i' \geq i+1$. Fortunately, Install part of Protocol-I is sufficient to guarantee this with a few changes as explained below. These changes are basically needed to incorporate the cyclic search of coordinator and the new relation between a configuration number and the number of the FS process acting as coordinator in that configuration.

6.4.1 Definition of *eligible*()

P_c is said to be eligible to act as the coordinator in Σ_i if $c = [(i\text{-}1)\ mod\ (f+1)] + 1$ and if FS processes eligible to act as coordinators for all Σ_g, $1 \leq g \leq i\text{-}1$, have indicated their unwillingness to act or to continue acting as the coordinator for their respective configurations. Next subsection describes how an eligible FS process can show its unwillingness to act as coordinator for a particular configuration.

6.4.2 Showing unwillingness for a configuration

We first note that in Protocol-0 and Protocol-I, fail-signal is a sure indication of a failure and the signalled FS process is considered as failed once and for all. Due to assumption 1B, this concept is no longer valid for Protocol-II. Hence fail-signal is not taken as a permanent failure here but just as unwillingness to accept coordinator-ship for a particular configuration. Secondly, we require a mechanism by which an eligible FS process can signal its unwillingness to act as the coordinator. This is achieved by tagging the fail-signal with the configuration number.

A fail-signal multicast by p_i for configuration Σ_j is denoted by $FS_i(j)$, where Σ_j is one of the designated configurations for which P_i is eligible to act as coordinator. It is assumed here that such fail-signal messages tagged with specific configuration numbers can be autonomously generated by a correct eligible process.

We note that $FS_i(j)$ cannot just be multicast at anytime an internal fault is detected because the fault may be temporary and P_i may become *up* again before the system is ready to enter into Σ_j. Therefore, $FS_i(j)$ is prepared and multicast at following two occasions.

1. P_i is already working as coordinator in Σ_j and detects an internal time- or value-domain fault.
2. $status_i$ = *down* or *temporarily_down* and *eligible*() = i.

In the first case, p_i detects a failure while working as coordinator in Σ_j and multicasts fail-signal to show unwillingness to continue playing the coordinator role. $status_i$ is updated to *down* or *temporarily_down* according to the detected fault being value - or time-domain respectively. Second case is when p_i is down but finds itself eligible for next configuration due to unwillingness of all preceding coordinators (see eligible definition in subsection 6.4.1 above). Hence, it shows its unwillingness to take over as the new coordinator by multicasting fail-signal.

6.4.3 Fail-Signal $FS_i(j)$

Recall that a fail-signal is generated by a constituent process without consultation with its counterpart. Moreover, every constituent process, say p_i, is provided with a fail-signal message singly signed by its counterpart p'_i at the time of system initialization. For Protocol-II, we assume the provision of multiple such singly-signed fail-signal messages each tagged with a configuration number j, for all configurations that P_i will be coordinating i.e., $j = i + k(f+1)$, $k \geq 0$. Hence, these single-signed messages are double-signed and multicast by a constituent process when the need arises.

Of course, the number of configurations that p_i will cycle through before getting stabilized at one cannot be known at system start. Hence, provision of an infinite number of single-signed fail-signal messages to p_i is not possible in practice. Hence, $FS_i(j)$ is redefined to have the following two fields.

1. Double-signed FS_i – This is the same as has been used for Protocol-0 and -I.
2. *Unwilling*(j) – This is an integer that holds the value of the configuration number j for which p_i does not want to play coordinator role.

Thus, only a single-signed fail-signal needs to be provided to p_i as before. When p_i needs to show its unwillingness to become or continue to act as coordinator for Σ_j, it double signs the fail-signal message to form double-signed FS_i. p_i then prepares $FS_i(j)$ to include FS_i and *Unwilling*(j), signs it and multicasts to all processes.

***Optimization Remark*:** In case of detection of a value-domain fault, p_i irreversibly changes $status_i$ to *down* and hence will never play coordinator role for any successive configuration. p_i indicates this permanent failure by using a boolean variable *perm*. *perm* is set to true and is added as a special field in $FS_i(j)$. This special $FS_i(j)$ is multicast as soon as $status_i$ is changed to *down*. j corresponds to the current configuration number if P_i was working as the coordinator at the time of detection,

otherwise it is the number of the next configuration for which P_i was supposed to be the coordinator. Any process receiving this special $FS_i(j)$ message, considers p_i to have failed permanently and hence will not wait for any further fail-signal messages from p_i to show unwillingness for any successive configuration. P_i keeps working as a passive FS process but will neither act as a coordinator nor will send any more fail-signal messages.

6.4.4 Other minor changes

1. $STATUS_i(j)$

   Every process p_k prepares a $STATUS_k(j)$ for every $FS_i(j)$ and multicasts to all. $STATUS_k(j)$ is prepared as in Protocol-I except that here j refers to a configuration number and not the coordinator FS process number. Hence, $STATUS_k(j)$ will contain messages that were sent in Σ_j, if any. Otherwise an empty $STATUS_k(j)$ is sent.

2. **Source Number *s***

   Say P_i is the eligible FS process with $status_i = up$. It prepares $START_i$ by identifying source number s and using the corresponding $Status\_Pool(s)$. However, as for j in $STATUS_i(j)$ above, s corresponds to the latest realized configuration number now. Recall that s was sent in *START* message so that all processes can use Q_s and $Acceptors_s$ to commit $ORDER_i(start\_o_i)$. Since now Q_s and $Acceptors_s$ are constants, *START* no longer needs to carry the source number. Recall that no conflict resolution or spuriousness check is needed and hence, construction of *START* becomes very simple.

3. $Signalled_i$

   Recall that $Signalled_i$ contains fail-signal messages of all FS processes. This set is multicast as a part of $START_i$ to prove failure of all predecessor coordinators of P_i and to justify P_i's attempt to install as new coordinator. The equivalent in Protocol-II is to send fail-signal messages corresponding to at least $(f+1)$ preceding configurations. This will cover each coordinator's unwillingness for at least one preceding configuration and will be sufficient to justify P_i's attempt. If P_i is attempting to install as coordinator of Σ_j, where $j = [i + l(f+1)]$, $l \geq 0$ then $Signalled_i$ will consist of fail-signal messages for configurations from $\min\{1, i + (l-1)(f+1)\}$ to $[(i-1) + l(f+1)]$. If a fail-signal for a configuration in this range is not received by P_i, this will be due to the corresponding coordinator being permanently down. P_i uses

the special fail-signal message sent by that process with *perm* = true as evidence (see optimization remark in subsection 6.4.3).

6.5 Summary

This chapter presented Protocol-II designed with different underlying assumptions than the earlier protocols presented in this thesis. Protocol-II shares the basic structure with Protocol-I and hence can be considered its twin. The new assumption set consists of a less restricted timing requirement but a stronger failure pattern assumption as compared to Protocol-I. That is, it allows the timing estimates to be inaccurate but only to hold eventually. Moreover, it only allows at most one failure within FS process.

It was shown that the two assumptions lead to two major changes. Firstly, since timing estimates can be inaccurate, fail-signal is no longer a sure indication of fault. Hence, although ineligible to play the role of coordinator, signalled process is allowed to participate as acceptor. This gives new meaning to fail-signal, which is now generated to show unwillingness for coordinator-ship of a given configuration and not to signal failure. Also, system needs to expand to have (f+1) FS processes now to let only the FS processes play the role of coordinator. The coordinator search may go in cycles through all (f+1) FS processes and in the worst case, all of them may be temporarily down at the same time.

Secondly, due to the presence of at least one correct process in FS process, FS process behaviour is represented with the original 3-state model without Byzantine state. This simplifies Install part of the protocol in two ways; first is that no spurious or conflicting plants need to be identified as signalled coordinator can never become Byzantine faulty and secondly number of phases of communication is reduced.

The new protocol design is presented in terms of amendments that need to be done in Protocol-I to adapt to the new assumptions. This mainly includes redefining configurations and *Acceptors* sets and dealing with the effects of these changes.

Chapter 7

Summary and Conclusions

This thesis introduced a new class of Byzantine fault-tolerant protocols that optimally use fail-signal processes to circumvent the FLP impossibility. It studied the design and analysis of three total-order protocols. Each of these protocols caters for different sets of assumptions that can possibly be made in the construction of fail-signal process. The three sets of assumptions cover a range of system contexts from near ideal to solely practical ones. The performance of one of the proposed protocols was extensively examined and compared against a canonical protocol in various network settings. The study gave encouraging results. This chapter summarizes the work presented in this thesis and presents some directions for future work.

7.1 Summary

The FLP impossibility needs to be circumvented for achieving a correct total order. This is typically done by using randomization or making weak synchrony assumptions. Randomized protocols [Ben83, EMR01, KS01] guarantee liveness in probabilistic terms to be a certainty with the passage of time. Every replica randomly chooses its estimate of the decided value from a set of initially proposed values. The protocol strives to get estimates of all replicas to converge to a single decision value.

Protocols of the second category make some assumptions about the bounds on communication delays between, and relative processing speeds of, replicas in the system. These can be further divided into two groups. First is a partitionable approach [BDM97, BBD97, ADK+92, EMS95] which assumes that estimates of these bounds are rarely violated. Replicas suspect other replicas to be failed when the associated bounds are violated. The suspecting replicas then reach agreement to exclude the suspected replicas from the group. Thus, when the bounds are violated, this approach can lead to formation of multiple groups or partitions each consisting of correct replicas that do not mutually suspect each other and do suspect all others. (Hence the term partitionable.)

The second is the non-partitionable approach [Lam98, CT96, CL99, DFK+96, KMM97], which allows violations of bounds but assumes that these will eventually

hold and also a pre-determined bound on the number of failures which is never violated. These two assumptions together eliminate the need to exclude the suspected ones. Most of the protocols in this class tend to be coordinator-based with one replica acting as the coordinator at a given time. The coordinator is empowered to enforce its decision on others. This has two performance implications: communication tends to 1-to-n or n-to-1, rather than n-to-n. Secondly, the failure monitoring is focused only on the coordinator; if the coordinator remains unsuspected, ordering is swift and the part of the protocol activated here is termed an the Normal part. If, on the other hand, the coordinator is suspected, replicas choose another peer as the coordinator often according to a pre-determined ranking of replicas. This change-over is the most expensive part of the protocol opearions which we refer to as the Install part. It is guaranteed to lead to Normal part, once the bound estimates used hold and false suspicions cease.

The protocols presented in this thesis belong to the non-partitionable class but use a different approach to circumvent FLP impossibility which essentially disallows false suspicions of coordinators. This approach involves using redundancy to construct a special process, called the fail-signal (FS) process, to perform the coordinator role. This FS process signals on failure and the failure characteristics are otherwise the same as that of a benign crash. That is, when FS process detects an internal fault, it stops working after signalling its stopping. Since the failures are no longer quiescent, FLP result ceases to apply. This approach is not new and has been explored by [MES03]. [MES03] replaced every process of a partitionable group communication protocol named Newtop by an FS process. The new system was called FS-Newtop. Since fail-signal is a sure indication of failure, no false suspicions exist and correct processes exclude from the group only the processes that have signaled. This prevents system from splitting into virtual partitions. However, FS-Newtop needs more than optimal number of replicas.

A significant benefit of using FS process abstraction for a total-order protocol design is that the solution does not have to rely on any synchrony assumption for either liveness or safety. However, synchrony assumptions are made for construction of FS process; otherwise a solution would not be possible [FLP85]. Thus, this approach limits the scope of synchrony assumption to (FS) process-level and not across system-level. Moreover, it transforms the Byzantine behaviour of constituent processes into crash of the FS process. But these benefits come at a cost which is in the form of restriction of the location of simultaneous faults. That is, the constituent processes of an FS process

cannot fail simultaneously. The aim of this thesis is to exploit fail-signal abstraction to the fullest extent and design Byzantine fault-tolerant protocols that are optimal in redundancy requirement and demonstrably efficient. In this thesis, we assume that a process pair constitutes an FS process.

We began by presenting the construction and characteristics of FS processes in chapter 3 in detail. An FS process was shown to be in one of three states: Working, Signalled and Failing. It was also shown that an FS process can exhibit two-facing behaviour in failing state which becomes problematic in the protocol design. The two core assumptions used in the construction of FS process are (1) bounds on communication and processing delays are known and (2) at most one constituent process can fail. Chapter 3 introduced a rudimentary Byzantine fault-tolerant total-order protocol named Protocol-0 which demonstrated the use of FS process while assuming that the failing state is never encountered. Finally, we analysed the effects of including failing state in the solution and proposed some modifications to handle these effects.

Chapter 4 extended the design of Protocol-0 and proposed an advanced protocol, Protocol-I. Protocol-I uses a collection of FS and non-FS processes and thus involves a major step forward over the earlier work of [MES03]. It also defines two mode of operation for FS process; Active and Passive. Moreover, it relaxes assumption 2 used in Protocol-0 to allow both constituent processes to fail as long as the failures are sufficiently apart in time. This adds the fourth state named Byzantine state to the FS process model. Protocol-I has two parts. Chapter 4 focussed on Normal part which is executed so long as the coordinator has not fail-signalled. We also presented a comprehensive performance study to evaluate the practicality of Protocol-I. BFT [CL99], a protocol well-known for its best performance in normal situations was used for comparison purposes. The experiments were performed in LAN and emulated WAN configurations with changing parameters like size of the system and cryptography techniques. The experiment results showed that Protocol-I outperforms BFT in all runs and the performance gap is more pronounced in slower WAN configurations.

Chapter 5 described the Install part of Protocol-I. Install part deals with all complications due to arbitrary behaviour of FS as well as non-FS processes in Byzantine state. It was shown that Install of Protocol-I involves more communication phases than that of BFT. This is the cost paid by Protocol-I for achieving high performance in Normal part. Performance study for Install showed higher latency trends of Protocol-I against BFT. However, the gap was found to be smaller for small number

of replicas and fast WAN configuration. Moreover, in attempt to reduce constant size overhead of a large message of Install, an optimized version, named Install-II, was proposed with correctness proof and critical analysis. Install-II was deemed to be useful in symmetric network environments.

Finally, Chapter 6 completed the family of protocols by presenting Protocol-II. Protocol-II relaxes the synchrony assumption used in the first two protocols and allows the estimates of timing bounds to be inaccurate. However, like most of other non-partitionable system protocols, for liveness guarantees, used timeouts are assumed to hold eventually. Protocol-II retains the failure assumption used in Protocol-0. We have described Protocol-II as a variant of Protocol-I by discussing only changes needed to deal with the effects of new assumptions.

7.2 Conclusions

We have developed a family of Byzantine fault-tolerant order protocols by carefully applying a technique long-known for building robust process abstractions [SS83]. These abstract processes, called fail-signal processes, only crash and signal their failure before crashing. Consequently, not only the order latency and the message overhead fall but also the protocol becomes easier to implement and no synchrony assumptions need to be made among the (un-paired) processes that do not have to cooperate to build the fail-signal abstraction. These benefits come at a cost: constituent processes cannot fail simultaneously. We meet this requirement by assuming that if both constituent processes fail, the failure occurrences are separated by a threshold interval (assumption 2A in Protocol-I) or that at least one process never fails (assumption 2 in Protocol-0 and 2B in Protocol-II). Results from performance study of Protocol-I showed that Normal part of Protocol-I performs better than BFT, whereas Install part is outperformed. However, since failures are assumed to be a rare event, the cost is not considered very high.

7.3 Future Work

It is known that public key cryptography is the major contributor to high latencies of Byzantine fault-tolerant protocols. Researchers have found a way around this by replacing it with more efficient symmetric encryption schemes, particularly Message Authentication Codes (MACs). However, this replacement is not very straightforward.

This is majorly because of the inability of MACs to authenticate the sender of the encrypted message by a third party. This will specifically affect Install part of our protocols, say Protocol-I, where *proof_of_commitment* in *STATUS* and (f+1) signatures in (f+1)-*START* are used to authenticate the corresponding signed messages. However, MACs can be adopted by carefully changing the authentication mechanism in the protocol at the expense of communication of extra messages as shown by [CL00]. Hence, it would be interesting to see the performance improvement in the proposed protocols after adapting to MACs and to compare it with the MAC version of BFT [CL00].

Both assumption 2A and 2B (or 2) rule out both the constituent nodes of an FS node to fail simultaneously, say, due to the same underlying cause e.g., a failure of common power supply or a natural disaster when both nodes are housed at the same location. They require exercising measures to ensure failure independence between the nodes and in particular eliminating any possible common failure modes through means such as diversity of node hardware and operating systems and housing the nodes at distinct locations. In this thesis, we assume that fail-independence can be sufficiently assured to make assumption 2A realistic in practice. However, we would like to conduct some research on the average width of window of vulnerability (2D time) and the time it takes to compromise a node. This will also include study on various defense mechanisms that can be utilized to prevent early compromise/failure of a constituent node. Moreover, we note that assumption 2B is a stronger assumption since it expects at least one node to remain non-faulty throughout the mission time. The larger is the latter, the less likely that it will hold. For longer operative periods, we require that the FS process be built using more than two processes when 2B is assumed; more precisely, each of the selected replica nodes to form FS node (see Figure 1.1) needs to be supplemented with ϕ, $\phi > 1$, shadow nodes and at most ϕ nodes can fail in a given FS node.

IT infrastructures have increasingly been targeted by malicious attacks and intrusions. To keep the maximum number of failures to the assumed bounded value during the life time of the system i.e. f-out of-n in the system and ϕ out of (ϕ+1) in an FS process, replicas should be allowed to participate after recovering from faults. In our proposed protocols this extension in the form of recovery can be both at system level and FS process level. That is, the signaled FS processes to be included back in the

system and faulty constituent processes back in the FS process after they have been repaired. This allowance will need several issues to be considered. For example, inclusion of the repaired processes should be viewed consistently by all processes in the system, the state of the repaired processes should be updated to the latest, new public keys of the repaired processes should be made known to all, some mechanism needs to be added to authenticate earlier messages signed by these processes before occurrence of failures but yet an adversary should not be allowed to send malicious messages as if they were sent earlier by exploiting old (leaked) public key etc.

After measuring the performance of the proposed protocol in failure-free and crash-like failure situations, we would like to see the effect on performance in various scenarios with faulty processes exhibiting arbitrary behaviour. This may effect the size of messages exchanged in Install/View-Change part and can also result in frequent suspicions in BFT. Moreover, we would like to see to what extent damage can be caused by violation of assumption 2A (2D time assumption), quantifying which is not trivial. Also, after experimenting in controlled environment, it would be interesting to see if the comparative figures are retained in real networks like PlanetLab [WWW1].

As highlighted earlier, the protocols presented in this thesis do not rely on any synchrony assumption for liveness on system-level. Hence, the performance is solely dependent on actual network conditions and true failures. This feature makes these algorithms ideal to be used to implement an e-service. This service can be used by various distributed applications needing to reach consensus on some action during their execution. Hence, it would be interesting to implement these protocols or their variants as a web service. Our paper [IE07] discussed this idea and proposed an earlier version of Protocol-0 as a Distributed Consensus Engine for large-scale self-organizing network applications.

References

[AD76] P. A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources", In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, San Francisco, CA, Oct. 1976.

[ADG+03] T. Anker, D. Dolev, G. Greenman, and I. Shnayderman. "Evaluating Total Order Algorithms in WAN", In *International Workshop on Large-Scale Group Communication*, 2003.

[ADK+92] Y. Amir, D. Dolev, S. Kramer and D. Malki, "Membership Algorithm for Multicast Communication Groups", *In Proceedings of 6th International Workshop on Distributed Algorithms*, pp 292-312, November 1992.

[AFM+04] E. Anceaume, A. Fernández, A. Mostéfaoui, G. Neiger, M. Raynal, "A Necessary and Sufficient Condition for Transforming Limited Accuracy Failure Detectors", *Journal of Computer and System Sciences (JCSS)*, 68(1): 123-133, 2004.

[AH90] J. Aspnes and M. Herlihy. "Fast Randomized Consensus Using Shared Memory", *Journal of Algorithms*, vol. 11, no. 3, pp. 441-460, 1990.

[ANB+07] M. Asplund, S. Nadjm-Tehrani, S. Beyer, P. Galdamez, "Measuring Availability in Optimistic Partition-tolerant Systems with Data Constraints", In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 656-665 , Edinburgh, UK, June 2007

[AS04] M. Alsaeed, N. A. Speirs, "A WAN Emulator for CORBA Applications", *Technical Report CS-TR-862*, University of Newcastle upon Tyne, October 2004.

[AS07] M. Alsaeed, N. A. Speirs, "A Wide Area Network Emulator for CORBA Applications", *In Proceedings of 10th International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pp. 359-364, Greece, May 2007.

[Asp03] J. Aspnes, "Randomized Protocols for Asynchronous Consensus", *Distributed Computing*, vol. 16, no. 2-3:165-175, September 2003.

[ASW97] N. Asokan, M. Schunter and M. Waidner, "Optimistic protocols for Fair Exchange", In *Proceedings of Fourth ACM Conference on Computer and Communications Security*, pp. 8-17, April 1997.

[AT96] M. K Aguilera and S. Toueg. "Randomization and failure detection: A hybrid approach to solve consensus". In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG),* LNCS1151, pp. 29-39, October 1996

[BBD97] O. Babaoglu, A. Bartoli, and G Dini, "Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems", *IEEE Transactions on Computers*, 46(6): pp.642-658, June 1997.

[BDG+95] O. Babaoglu, R. Davoli, L.A. Giachini and P. Sabattini, "The Inherent Cost of Strong-Partial View Synchronous Communication", In J.-M. Helary and M. Raynal, (ed) *Distributed Algorithms*, Lecture Notes in Computer Science, pp. 72-86, Springer-Verlag, 1995.

[BDM95] O. Babaoglu, R. Davoli, and A Montresor, "Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications", *Technical Report UBLCS-95-18*, Dept. of Computer Science, University of Bologna, Italy, Nov 1995.

[BDM97] O.Babaoglu, R. Davoli, and A Montresor, "Partitionable Group Membership: Specifications and Algorithms", *Technical Report UBLCS-97-1*, Dept. of Computer Science, University of Bologna, Italy, Jan 1997.

[Ben83] M. Ben-Or. "Another advantage of free choice: completely asynchronous agreement protocols". *In Proceedings of 2nd ACM Symposium on Principles of Distributed Computing*, pp. 27-30, 1983.

[BES+96] F.V. Brasileiro, P.D. Ezhilchelvan, and S.K. Shrivastava, N.A. Speirs, S. Tao, "Implementing Fail-Silent Nodes for Distributed Systems", *IEEE Transactions on Computers*, 45(11): pp 1226-1238, 1996.

[BGM+01] F. Brasileiro, F. Greve, A. Mostefaoui, M. Raynal, "Consensus in One Communication Step", *In Proceedings of the 6th International Conference on Parallel Computing Technologies*, Lecture Notes In Computer Science; Vol. 2127, 2001.

[BHR+99] R.Baldoni, J. Hélary, M. Raynal, L. Tanguy, "Consensus in Byzantine Asynchronous Systems", *Research Report 3655*, INRIA, France, 1999.

[Bra85] G. Bracha, "An "o(log n)" expected rounds randomized Byzantine generals protocol", *In Proceeding of 17th ACM Symposium on Theory of Computing*, ACM, New York, 1985.

[CB97] M.E. Crovella and A. Bestavros, "Self-similarity in World Wide Web traffic: Evidence and possible causes," *IEEE/ACM Transactions on networking*, 5(6): 835-846, December 1997.

[CD89] B. Chor, and C. Dwork, "Randomization in Byzantine agreement". *Advan. Comput. Res.* 5, 443-497, 1989

[CF99] F. Cristian and C. Fetzer. "The Timed Asynchronous Distributed System Model", *IEEE Transactions on Parallel and Distributed Systems,* pages 642-657, June 1999.

[CGS00] B. Charron-Bost, R. Guerraoui, and A. Schiper. "Synchronous system and perfect failure detector: Solvability and efficiency issues". In *International Conference on Dependable Systems and Networks*, (IEEE Computer Society), 2000

[CHT96] T.D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus", *Journal of the ACM* , vol. 43, pp. 685-722, July 1996

[Chu98] F. Chu, "Reducing Ω to ◊*W*", *Information Processing Letters*, 76(6): 293-298, 1998.

[CL00] M. Castro and B. Liskov, "Proactive Recovery in a Byzantine-Fault-Tolerant System", *In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, USA, October 2000.

[CL02] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery", *ACM Transactions on Computer Systems (TOCS)*, 20(4), November 2002.

[CL99] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *In Proceedings of the 3rd ACM Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 173-186, February 1999.

[CMS89] M. Chor, M. Merritt, and D.B. Shmoys, "Simple Constant-Time Consensus Protocols in Realistic Failure Models", *Journal of the ACM*, 36(3): 591–614, 1989.

[CS00] B. Charron-Bost, A. Schiper, "Uniform Consensus is Harder Than Consensus (extended abstract)", Technical Report LSR-REPORT-2000-014, Ecole Polytechnique Federale de Lausanne, Switzerland, 2000.

[CT91] T.D. Chandra and S. Toueg, "Unreliable failure detectors for asynchronous systems (Preliminary version)". *In Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pp. 325 – 340, Montreal, Quebec, Canada, 1991.

[CT96] T.D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems". *Journal of the ACM*, 43(2): 225-267, March 1996.

[DDS87] D. Dolev, C. Dwork and L. Stockmeyer, "On the minimal synchrony needed for distributed consensus", *Journal of the ACM*, 34(1): pp. 77-97, Jan 1987.

[DFK+96] D. Dolev. R. Friedman. I. Keidar and D. Malkhi. "Failure detectors in omission failure environments". *Technical Report 96-1605*, Department of Computer Science, Cornell University, September 1996.

[DGG05] A. Doudou, B. Garbinato, and R. Guerraoui, "Tolerating Arbitrary Failures with State Machine Replication", In *Dependable Computing Systems*, John Wiley & Sons, Inc, 2005.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. "Consensus in the presence of partial synchrony". *Journal of the ACM*, 35(2):288-323, April 1988.

[DM96] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communication", *Communications of the ACM*, 39(4):64-74, April 1996.

[DMS06] D. Dobre, M. Majuntke and N. Suri, "CoReFP: Contention-Resistant Fast Paxos for WANs", *Technical Report TR-TUD-DEEDS-11-01-2006*, Department of Computer science, Technische Universitat Darmstadt, 2006.

[DS97] A. Doudou and A. Schiper. "Muteness detectors for consensus with Byzantine processes". *Technical Report TR97-230*, Department of Computer Science, Ecole Polytechnic Federale de Lausanne, October 1997

[DSU04] X. Defago, A. Schiper and P. Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey", *ACM Computing Survey*, 36(4); 372–421, 2004.

[EMR01] P D Ezhilchelvan, A Mostefaoui, and M Raynal, "Randomized Multivalued Consensus", *In Proceedings of the Fourth International IEEE Symposium on Object oriented Real-time Computing (ISORC)*, pp. 195-201, Magdeburg, Germany, May 2001.

[EMS95] P.D. Ezhilchelvan, R.A. Macêdo, and S.K. Shrivastava, "Newtop: A Fault-Tolerant Group Communication Protocol", *In Proceedings of the 15th IEEE International Conference on Distributed Computing Systems (ICDCS '95)*, pp. 296-306, Vancouver, BC, Canada, 30 May - 2 June 1995.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process", *Journal of the ACM*, 32(2): 374-382, April 1985.

[FM03] C. Fraleigh and S. Moon, et al. "Packet-Level Traffic Measurements from the Sprint IP Backbone", *In IEEE Network*, 17(6), November 2003.

[FM97] Pesech Feldman and Silvio Micali. "An optimal probabilistic protocol for synchronous Byzantine agreement", *SIAM Journal on Computing*, 26(4):873-933, August 1997.

[FMR07] Roy Friedman, Achour Mostefaoui and Michel Raynal, "On the Respective Power of $\Diamond P$ and $\Diamond S$ to Solve One-Shot Agreement Problems", *IEEE Transactions on Parallel Distributed Systems*, 18(5): 589 – 597, May 2007.

[Gif79] D. K. Gifford, "Weighted voting for replicated data", In *Proc. of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, Dec. 1979.

[GS97] R. Guerraoui and A. Schiper. "Software-based replication for fault tolerance", *IEEE Computer*, 30(4), pp. 68–74, April 1997

[HN99] S. Haddad and F. Nguilla. "Combining Different Failure Detectors for Solving a Large-Scale Consensus Problem". In *14th ISCA-CATA*, Cancun, Mexico, April 1999

[IE06] Q. Inayat and P.D. Ezhilchelvan, "A Performance Study on the Signal-On-Fail Approach to Imposing Total Order in the Streets of Byzantium", *In Proceedings of the*

2006 International Conference on Dependable Systems and Networks, pp. 578-587, Philadelphia, PA, USA, 25-28 June 2006.

[IE07] Q. Inayat and P.D. Ezhilchelvan, "A Consensus Service for Applications in Large-Scale Self-Organizing Networks", *In Proceedings of Workshop on Dependable Application Support in Self-Organising Networks (DASSON) held with DSN 2007*, Edinburg, UK, June 2007.

[Jac88] V. Jacobson, "Congestion avoidance and control", *In Proceedings of SIGCOMM 88*, volume 18, pages 314-320, August 1988.

[JMM07] Flavio Junqueira and Keith Marzullo, "Classic Paxos vs. Fast Paxos: Caveat Emptor", In *Proceedings of the IEEE Workshop on Hot Topics in System Dependability (HotDep)*, Edinburg, UK, June 2007.

[KMM97] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. "Solving consensus in a Byzantine environment using an unreliable failure detector". In *Proceedings of the International Conference on Principles of Distributed Systems*, pages 61-75, December 1997.

[Kop97] Kopetz H. "Real-Time Systems: Design Principles for Distributed Embedded Applications". Kluwer Academic Publishers, 1997, ISBN 0-7923-9894-7.

[KR01] I. keidar and S. Rajsbaum, "On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial", *Technical Report MIT-LCS-TR-821*, MIT Laboratory for Computer Science, May 2001.

[KS01] K. Kursawe and V. Shoup, "Optimistic asynchronous atomic broadcast." *Cryptology ePrint Archive*, Report 2001/022, Mar. 2001

[KS05] K Kursawe and V Shoup, 'Optimistic Asynchronous Atomic Broadcast', In *Proceedings of International Colloqium on Automata, Languages and Programming (ICALP05)*, LNCS 3580, pp. 204-215, Springer, 2005.

[KS07] I. Keider, A. Shraer, "How to Choose a Timing Model?", *In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 389 - 398, Edinburgh, UK, June 2007

[Lam01] L. Lamport "Paxos Made Simple", *ACM SIGACT News (Distributed Computing Column)*, 32, 4: 18-25, December 2001.

[Lam06] L. Lamport, "Fast Paxos", *Distributed Computing*, 19(2): 79-103. October 2006.

[Lam78] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM* 21(7), pp. 558-565 (1978).

[Lam98] Leslie Lamport, "The part-time parliament", *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[LSP82] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, 4(3): 382-401, July 1982.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[MA06] J-P. Martin, L. Alvisi, "Fast Byzantine Consensus", *IEEE Transactions on Dependable and Secure Computing*, 3(3):202- 215, July-Sept. 2006.

[MES03] D. Mpoeleng, P.D. Ezhilchelvan and N.A. Speirs, "From Crash-tolerance to Authenticated Byzantine Tolerance: a Structured Approach, the costs and Benefits", In *Proceedings of International Conference on Dependable Systems and Networks (DSN2003)*, pp.227-236, June 2003

[MMA93] L.E. Moser, P.M. Melliar-Smith, And V. Agrawala, "Asynchronous fault-tolerant total ordering algorithms", *SIAM J. Comput. 22*, 4 (Aug.), 727–750, 1993.

[MNC+06] H. Moniz, N. F. Neves, M.l Correia and P. Veríssimo, "Randomized Intrusion-Tolerant Asynchronous Services", *Proceedings of the International Conference on Dependable Systems and Networks* (DSN), Philadelphia, USA, pages 568-577, June 2006.

[MR97] D. Malkhi and M. Reiter, "Unreliable Intrusion Detection in Distributed Computations", *In Proceedings of the 10th IEEE Computer Society Foundations Workshop*, pp. 116-124, Rockport (MA), June 1997.

[MR99a] A. Mostefaoui and M. Raynal, "Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Approach", In *Proceedings of 13th International Symposium on DIStributed Computing (DISC)*, pp. 49-63, 1999.

[MR99b] A. Mostefaoui, M. Raynal, "Unreliable Failure Detectors with Limited Scope Accuracy and an Application to Consensus", *In Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes In Computer Science; Vol. 1738, pp. 329-340, 1999.

[MRR+05] A. Mostefaoui, S. Rajsbaum, M. Raynal and C. Tavers, "From $\Diamond W$ to Ω : a Simple Bounded Quiescent Reliable Broadcast-based Transformation", *Research Report 1759*, INRIA, France, November 2005.

[MRR+06] Achour Mostéfaoui, S. Rajsbaum, M. Raynal, Corentin Travers, "From Failure Detectors with Limited Scope Accuracy to System-wide Leadership", *In Proceedings of 20th International Conference on Advanced Information Networking and Applications (AINA 2006)*, Vol. 1, pp. 81-86, 2006.

[PS03] F. Pedone and A Schiper, "Optimistic Atomic Broadcast: A Pragmatic View-point" in *Theoretical Computer Science (Elsevier)*, Vol. 291 (1), pp. 79-101, 2003.

[Rab83] M. O. Rabin. "Randomized Byzantine Generals", In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pp 403-409, November 1983

[RB91] A. Ricciardi and K. Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments," *ACM Symposium on Principles of Distributed Computing*, pp. 341-353, Montreal, Quebec, Canada, August 19-21, 1991.

[RKB07] R. Rodrigues, P. Kouznetsov, B. Bhattacharjee, "Large-Scale Byzantine Fault-Tolerance: Safe but Not Always Live", In *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep'07)*, June 2007.

[RL04] Rodrigo Rodrigues and Barbara Liskov, "Byzantine Fault Tolerance in Long-Lived Systems", In *2nd Bertinoro Workshop on Future Directions in Distributed Computing (FuDiCo II)*, (Bertinoro, Italy), June 2004

[Sch84] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors", *ACM Transactions on Computer Systems*, Vol. 2(2), pp. 145-154, May 1984.

[Sch93] F. B. Schneider. "Replication management using the state-machine approach", In *Distributed Systems* (edited by S. Mullender), ACM Press, pp. 169–197, 1993

[Sch97] André Schiper, "Early consensus in an asynchronous system with a weak failure detector", *Distributed Computing*, vol. 10, Issue 3, pp. 149–157, 1997

[SNL+06] P. Sousa, N. F. Neves, A. Lopes, and P. Verissimo." On the resilience of intrusion-tolerant distributed systems", DI/FCUL TR 06–14, Department of Informatics, Univ. of Lisbon, Sept 2006.

[SNV05] P. Sousa, N. F. Neves, and P. Verissimo. "How resilient are distributed *f* fault/intrusion-tolerant systems?", In *Proc. of the Int. Conf. on Dependable Systems and Networks*, pages 98–107, June 2005.

[SNV07] P. Sousa, N. F. Neves, P. Verissimo, "Hidden Problems of Asynchronous Proactive Recovery", In *Proceedings of Third Workshop on Hot Topics in System Dependability (HotDep'07)*, June 2007.

[SS83] R. Schlichting and F. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", *ACM Transactions on Computer Systems*, Vol. 1(3), pp. 222-238, August 1983

[TS90] A. Tully, S. K. Shrivastava, "Preventing state divergence in Replicated Distributed Programs", *In Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems (SRDS-9)*, Huntsville, Alabama, USA, pp. 104-113, October 1990

[Tsu92] G Tsudik, "Message Authentication Using one-way Hash Functions", *ACM Computer Communications Review*, 22(5), 1992.

[UHS+04] P. Urbán, N. Hayashibara, A. Schiper and T. Katayama "Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm", *In Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS'04)*, pp. 4-17, Ishikawa, Japan, October 2004.

[WWW1] Planetlab. http://www.planet-lab.org

[YMV+03] J.Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M Dahlin, "Separating Agreement from Execution for Byzantine Tolerant Services", *In proceedings of SOSP*, pp. 253-267, 2003.