The Primary-Backup Approach: Lower and Upper Bounds

Navin Budhiraja
Ph.D Thesis

93-1353
June 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

THE PRIMARY–BACKUP APPROACH: LOWER AND UPPER BOUNDS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Navin Budhiraja

August 1993

THE PRIMARY-BACKUP APPROACH:
LOWER AND UPPER BOUNDS

Navin Budhiraja, Ph.D.
Cornell University 1993

The most widely used approach to building replicated, fault-tolerant services is the primary-backup approach. In this approach, the state of the service is replicated across multiple servers, with one server designated as the primary, and the rest as backups. Clients send requests only to the primary. However, in case the primary fails, one of the backups takes over as the new primary. Ever since it was introduced in 1976 by Alsberg and Day, the primary-backup approach has become the basis for building many practical fault-tolerant services. However, despite the widespread use, the approach has not been studied systematically, and little is known of the fundamental costs and tradeoffs of using the approach under various kinds of failures. Thus there is a gap between theory and practice. In order to close this gap, this thesis analyzes the primary-backup approach, both from the theoretical perspective of specification, lower bounds and upper bounds, as well as from the practical viewpoint of performance tradeoffs in protocols. We identify three key cost metrics of primary-backup protocols—degree of replication, blocking time and failover time—and then show lower and upper bounds on these metrics

for a hierarchy of failure models. We then implement an important subclass of our primary–backup protocols, called 0–blocking protocols, and give performance figures. In addition to leading to the development of new, more efficient protocols, we believe that the work in this thesis has resulted in a better understanding of the properties of existing primary–backup protocols.

Biographical Sketch

Navin Budhiraja was born in Bhilai, Madhya Pradesh, India on January 17, 1966. Though he spent most of his childhood playing cricket in Bhilai, he also attended Higher Secondary School, and graduated in 1984. Subsequently, he joined the Computer Science Department at Indian Institute of Technology, Kanpur, from where he graduated with a bachelor of technology degree in 1988. In Fall 1988, he enrolled in the Computer Science Department at Cornell University, Ithaca, N.Y. as a graduate student. He received a masters of science degree in May 1991, and was awarded a doctor of philosophy degree in August 1993.

Acknowledgements

First of all, I would like to thank my advisor, Keith Marzullo, without whom this work would have not been possible. He provided me with direction and encouragement that has proved invaluable during the course of this work. I would also like to thank Fred Schneider and Sam Toueg, both of whom provided considerable assistance needed to bring this work to completion. Among other things, Fred's insight on building fault-tolerant services has been extremely helpful. And Sam, in addition to introducing me to the basics of Italian cuisine, also introduced me to research. I am also indebted to other members of my committee: Larry Blume and Devika Subramanium. The presentation in this thesis has benefited greatly from the comments of Keith, Fred, Devika and Sam.

Thanks also go to Prasad Jayanti for helping with the ideas behind Chapter 2. I have also benefited from various discussions with other members of the distributed systems group here at Cornell, specifically Tushar Chandra, Ajei Gopal, Mike Reiter, Pat Stephenson and Mark Wood.

The financial support provided by the Defence Advanced Research Projects Agency under Nasa Ames grant number NAG 2-593, Contract N00140-87-C-8904 is also acknowledged. In addition, this work has also been supported by an I.B.M. Graduate Fellowship.

Lastly, I am indebted to a beloved group of people without whom I would never have reached this stage in my life. These are my parents Chaman and Sneh Budhiraja, my sister Sabina and my wife Sonal. My family has been a source of pleasure, support and encouragement all through.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The most widely used technique for building fault–tolerant client–server systems is the *primary–backup* approach [AD76] shown in Figure 1.1. In this approach, any *service* is implemented using an ensemble of *servers*. The state of the service is replicated across the servers, with one server designated as the *primary* (server s_1 in the figure), and the remaining servers designated as *backups*. Clients send requests to the primary, which performs the requested service, and then responds to the clients. However, in order to keep the backups up-to-date with any changes that occur in the state of the service, the primary informs the backups of any state updates that occur. In case the primary fails, one of the backups (server s_2 in Figure 1.2) takes over as the primary, and clients send subsequent requests to this new primary.

Ever since it was introduced in 1976, the primary–backup approach has become the basis for building many practical and research systems. In particular, this approach has been used in the construction of a fault–tolerant version of UNIX[1]

[1]UNIX is a registered trademark of AT&T.

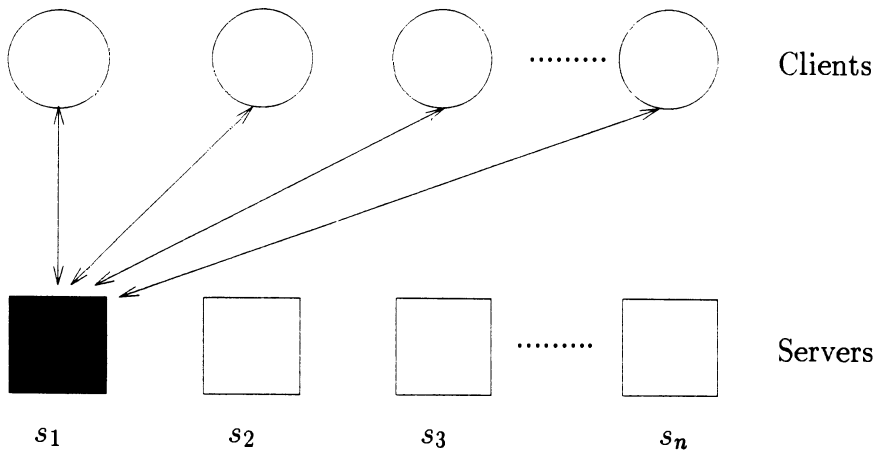

Figure 1.1: The primary–backup approach

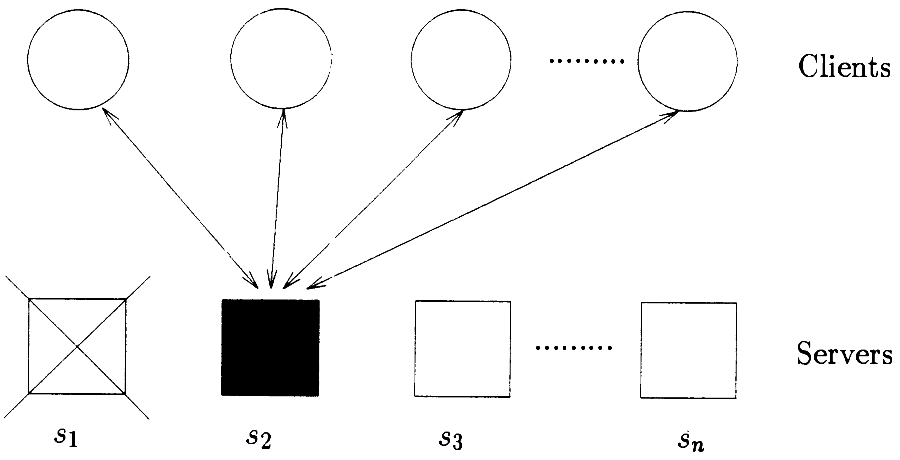

Figure 1.2: A backup takes over as the primary

[BBG+89], fault-tolerant parallel programs [BAA+92], highly available transaction processing systems [Bar81] and highly available file systems [MHS89,BEM91, LGG+91]. However, despite this widespread use in extant systems, the primary-backup approach has not attracted much systematic study, and little is known of the fundamental costs and tradeoffs of using this approach under various kinds of failures. For example, even though a variety of primary-backup protocols have been mentioned in the literature, it is not known whether these protocols are optimal in the number of servers they use, in the amount of time they take to respond to a client request, and so on. Thus, there is a sizable gap between theory and practice. In order to close this gap, this thesis studies the primary-backup approach, both from a theoretical viewpoint of specification, lower bounds and upper bounds, and from a practical viewpoint of performance tradeoffs in protocols. In addition to leading to a better understanding of existing primary-backup protocols by identifying which of these protocols are optimal, our work has also led to the development of new, optimal primary-backup protocols.

1.1 Contributions of the Thesis

Consider the following three key cost metrics of any primary-backup protocol.[2] The first is the **degree of replication** or the number of servers used to implement the protocol. For example, a protocol that uses $2f + 1$ servers to tolerate f server failures has a degree of replication of $2f + 1$. The second metric is the **blocking time** or the worst-case period between a request and its response in any failure-free execution of the protocol. For example, many existing primary-backup

[2]See Chapter 4 for formal definitions

protocols work as follows. In order to ensure that the states of the backups are up-to-date with the state of the primary, the primary sends to the backups any state update that occurs due to the receipt of a request, waits for the backups to acknowledge, and only then sends a response to a client. Therefore, assuming that all computation takes negligible time, the blocking time of these protocols equals the amount of time it takes to send a message and receive the corresponding acknowledgement. Finally, the last metric is the **failover time** or the worst-case period of time during which there is no primary. Any primary–backup protocol will have such a period because the current primary may fail, and consequently, a new primary will have to be elected.

Clearly, one is interested in knowing the values of these metrics for any primary–backup protocol. Not only do these metrics provide a basis for evaluating the protocol, they also provide a basis for comparing the protocol with other primary–backup protocols. However, one is also interested in answering a more fundamental question:

> Given that no more than f components can fail, what are the smallest possible values of the the degree of replication, the blocking time and the failover time?

An answer to the above question defines, in practical terms, what properties an optimal primary–backup protocol should have. Unfortunately, despite the widespread use of the primary–backup approach, the above question had not been addressed in the literature. We, therefore, do so in this thesis.

We answer this question by giving lower bounds for the three cost metrics that we identify, where a *lower bound* for a metric gives the smallest possible value of that metric for any possible primary–backup protocol. In turns out that the values of these lower bounds also depend on the underlying failure assumptions, and so we give the lower bounds for a commonly used hierarchy of failure models.

The existence of a lower bound, however, does not imply the existence of a protocol that achieves this bound. For example, degree of replication has a trivial lower bound of 0 (since the number of servers required to implement any primary–backup protocol cannot be negative). However, clearly no protocol can achieve this lower bound, because there has to be at least one server to send responses to the client requests. Therefore, in this thesis, we also demonstrate that most of our lower bounds are *tight* by showing that these lower bounds can be achieved in practice. We do this by giving *optimal* protocols for the various failure models *i.e.* protocols for which the degree of replication, the blocking time, and the failover time are equal to the values specified by the corresponding lower bounds.

Before we can derive lower bounds and upper bounds for the primary–backup approach, we need to precisely state what we mean by this approach *i.e.* give a specification for the primary–backup approach. The specification that we give in this thesis is not arbitrary, and we arrive at it by abstracting from existing protocols the "essence" of the approach. Not only is our specification weak enough to admit protocols that one would consider to be primary–backup, it is also strong enough to disallow protocols that one would not consider to be primary–backup. Since this is the first ever precise specification for primary–backup, the specification itself is an important contribution of this thesis because it provides a concrete point from which the primary–backup approach can be compared to other approaches for structuring fault–tolerant services.

The lower bound and upper bound results in this thesis have led to a better understanding of existing primary–backup protocols by identifying which of these protocols are optimal with respect to the metrics that we define. However, the contributions of this thesis are not entirely of a theoretical nature. Our theoretical work has led to the development of new, optimal primary–backup protocols in failure models where protocols were previously unknown. In order to compare the performance of these new protocols against existing protocols, and to better understand the applicability of our assumptions to practical systems, we also implement and analyze a subset of our protocols. The protocols that we implement form a very interesting subclass of the primary–backup protocols because these protocols are theoretically the most general class of protocols that can achieve the smallest possible response time.

1.2 Related Work

As we mentioned earlier, the primary–backup approach is a widely used technique for building replicated, fault–tolerant services. This approach, however, is not the only approach for building fault–tolerant services. Another important approach is the *state–machine* approach[Lam78,Sch90]. Like the primary–backup approach, the state–machine approach replicates the state of the service across all the servers implementing the service. However, unlike the primary-backup approach, any client request is *atomically broadcast* [CM84,CASD85,BJ87,BGT90] to each server (see Figure 1.3). Informally, atomic broadcast ensures that all (correct) servers receive the same set of requests, and they receive them in the same order. On receiving such a request, each server performs the requested service, updates

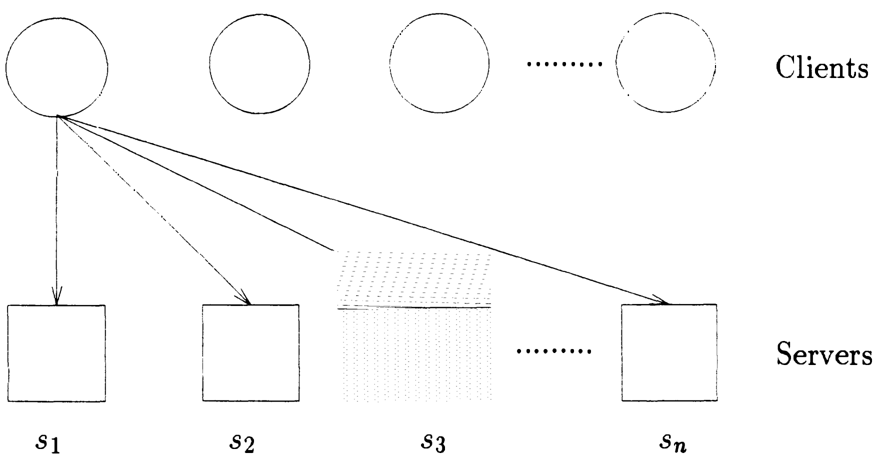

Figure 1.3: The state–machine approach

its local state if necessary, and then responds to the client.[3] Since a request is sent to each server, the failure of a server does not disrupt the service because the clients continue of get responses from the other servers (see Figure 1.4). The dashed arrow in Figure 1.4 indicates that the crashed server s_1 does not receive the request. Among others, this approach has been used in the design of process control applications [WLG$^+$78], distributed synchronization [Sch80] and highly available distributed services [LL86,MS88,Sie92].

The state–machine approach has attracted a great deal of theoretical study [Sch90], and a lot is known of the costs and tradeoffs of using this approach under various models of failures. For example, the fundamental primitives used in constructing state–machines is atomic–broadcast. Extensive amount of research has been done in order to understand the costs of atomic–broadcast for various models of failures [PSL80,DRS90,GSTC90,CDS90].

[3]In case of crash failures, if the atomic broadcast protocol prevents *inconsistency* [GT91], then the client accepts any response that it receives. For other kinds of failures, the client accepts the majority response.

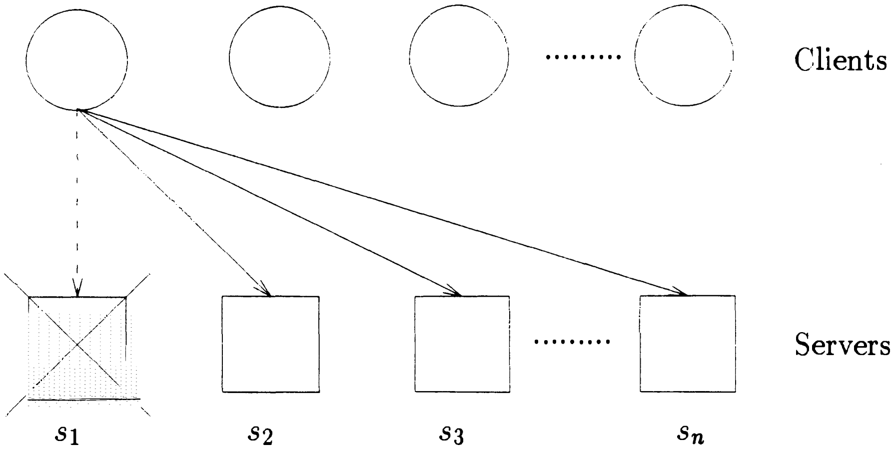

Figure 1.4: Server s_1 has failed

Furthermore, the state–machine approach is the preferred mode of replication in many time–critical applications. This is because failures of individual servers can be immediately masked from the clients—clients can continue to get responses from the other servers. In addition, this approach needs to place no restrictions on the behavior of the faulty servers and clients.

In contrast to the state–machine approach, with the primary–backup approach, a request may not generate any response. This may occur if the request was sent to a primary that had crashed. Even though it is possible to mask this failure by resending the request until the client receives a response, clients now may see a long delay before they receive this response. Furthermore, in contrast to the state–machine approach, the primary–backup approach only works for benign failures. However, the above two characteristics *i.e.* a longer delay in the unlikely case when failures occur, and a restriction on the kinds of failures that can be tolerated, do not preclude its use in many important applications, and the primary–backup approach is the one that is more extensively used in practice [Bar81,OL88,BBG+89,MHS89, SB89,CDD90,BEM91,LGG+91]. This is because it requires less processing (only at the primary, rather than at each server as in the state–machine approach), and is less costly (because it does not require the implementation of expensive atomic–broadcast protocols). Furthermore, the primary–backup approach can be used to build non–deterministic services, whereas the state-machine approach requires the servers to be deterministic.

The final section in this chapter describes the system model that we will use in showing our results, and then briefly discusses the structure of the rest of the thesis.

1.3 System Model

Two broad classes of primary–backup protocols have been considered in the literature: *asynchronous* and *synchronous.* In asynchronous protocols [OL88], no assumptions are made about the speeds of the processes, or the delays incurred in delivering messages from one process to another; in synchronous protocols [AD76, Bar81,BEM91], both process speeds and message delivery delays are known and bounded. An intermediate class of protocols in which the process speeds and the message delays are unbounded, but processes have clocks with bounded speeds have also been studied in the literature [MHS89,LGG+91]. However, for our purposes, we will classify these protocols along with the asynchronous protocols. In this thesis, we only concentrate on synchronous primary–backup protocols, because as we show in Chapter 2, it is impossible to guarantee both *safety* (the protocol gives correct responses) [Lam85] and *liveness* (the protocol eventually gives a response) [AS86] in asynchronous protocols.[4]

We consider a distributed system consisting of n_s servers and n_c clients. The servers have local clocks which are assumed to be perfectly synchronized with real time.[5] Clients and servers communicate through a completely connected, point-to-point, FIFO network. Furthermore, if processes (clients or servers) p_i and p_j are connected by a (non-faulty) link, then a message sent by p_i to p_j at time t arrives at p_j at some time $t' \in (t..t+\delta]$. We call δ the maximum message delivery delay.

For simplicity, we assume that every request elicits a response. We are only interested in identifying the costs inherent in primary–backup protocols, and so we also assume that it takes no time for a server to compute a response. However, our lower bounds are not invalidated for servers that require a substantial amount of time to compute a response. We consider the following hierarchy of failures:

Crash failures: A server may fail by halting prematurely. Until it halts, it behaves correctly [LF82].

Crash+Link failures: A server may crash or a link may lose messages (but links do not delay, duplicate or corrupt messages).

Receive-Omission failures: A server may fail not only by crashing, but also by omitting to receive some of the messages directed to it over a non-faulty link [PT86].

Send-Omission failures: A server may fail not only by crashing, but also by omitting to send some of the messages over a non-faulty link [Had84].

---

[4]The asynchronous protocols mentioned in the literature guarantee only safety. However, it is possible to construct safe and live implementations in an asynchronous system augmented with a *failure-detector* [BJ87,SS83]. We do not consider these in this thesis.

[5]Clearly, our lower bounds hold even if server clocks are only approximately synchronized [LMS84]. Furthermore, by using the techniques in [NT87], our protocols can be automatically extended to the case in which clocks are only approximately synchronized.

General-Omission failures: A server may exhibit send-omission or receive-omission failures [PT86].

Failures are counted by the number of failing components (either servers or links). A protocol tolerates f failures of a given model if it works correctly despite faulty behavior (as prescribed by that model) of up to f components.

The rest of the thesis is structured as follows. Chapter 2 shows that it is impossible to construct primary–backup protocols in asynchronous systems. Chapter 3 gives a precise specification of the primary–backup approach, Chapter 4 gives our lower bound results, and Chapters 5 and 6 give our upper bound results. We divide our upper bound results into two separate chapters because our protocols can be divided into two broad classes, and we address these two classes of protocols separately. Chapter 7 describes our implementation, and Chapter 8 gives a brief survey of existing primary–backup protocols and compares the costs of these protocols with the results in this thesis. We conclude in Chapter 9.

Chapter 2

Impossibility of Building Asynchronous, Fault-Tolerant Services

We had mentioned in Chapter 1 that this thesis only concentrates on synchronous primary–backup protocols because it is impossible to guarantee both safety and liveness in asynchronous protocols. In this chapter, we show this impossibility result.[1] In fact, the result we show is more general—we show that the impossibility holds for all protocols to building fault-tolerant services, not just primary–backup protocols.

Consider the following client–server system. A client c can send a request by calling a function $\texttt{REQUEST}(c, op, [arg_1 \ldots arg_n])$, where op is the request and $arg_1 \ldots arg_n$ are the (possibly empty) list of arguments to op. A response (if any)

[1] A similar result for a shared memory environment was shown by [Her91].

returned by this request should satisfy the following two properties:[2] *safety*—the response is "consistent" with the responses returned earlier; and *liveness*—if a correct client calls the function, then the function returns a response within a finite (but possibly unbounded) amount of time.

The implementation of `REQUEST` can depend on the way the client–server system is structured. For example, if the system is structured using the state machine approach, then `REQUEST` can atomically broadcast the request to all the servers that implement the system. If the system is structured using the the primary–backup approach, then `REQUEST` might need to send more than one copy of the request, perhaps to different primaries, in order to ensure that at least one of these copies eventually generates a response to the request. In either case, `REQUEST` can compute the final response from the replies received, and return this response to the client.

Clearly, we would like to construct such a `REQUEST` function for any practical client–server system, *i.e.* we would like it to satisfy the above safety and liveness properties. In fact, such systems ensuring both safety and liveness can be built in a synchronous environment with bounded number of failures [AD76,Bar81,BEM91, Sch90]. However, we show in this chapter that in an asynchronous environment with the possibility of failures, such a system cannot be built even for some simple safety properties.[3] Furthermore, our result holds even for partially asynchronous environments, *i.e.* environments in which process speeds are bounded but message delays are unbounded. We prove our result by showing that a safe and live implementation of a simple client–server system (called the *History system*) can

[2]More formal definitions appear later in the chapter.

[3]There are, however, safety properties for which implementations exist.

be used to solve the *consensus* problem [Fis83]. The result then follows from the impossibility of consensus in asynchronous environments[FLP85,DDS87].

The rest of the chapter is organized as follows. Section 2.1 gives a specification of the History system, and Section 2.2 shows the impossibility of constructing this system in an asynchronous environment with the possibility of failures.

2.1 Specification of the History system

Informally, a History system $\mathcal{H}$ behaves as follows: on receiving a client request, it just returns the sequence of requests that have been received until then. Clearly, the semantics of $\mathcal{H}$ are extremely useful as any other deterministic system can be implemented using $\mathcal{H}$—given an initial state and the sequence of requests that have been received, any response can be computed depending on the semantics of the system.

More precisely, in $\mathcal{H}$, a client c can initiate a request by calling the function REQUEST(c, op), where op is the next request (assume, for simplicity, that the argument list is empty). Consider the following (canonical) implementation of this History system using a *single non-faulty* server s (see Figures 2.1 and 2.2).

```
function REQUEST(c, op)
  send < c, op > to server s
    when received res from s :
      return res
```

Figure 2.1: The function REQUEST

In both the figures, the statement "**when** $G : S$" causes the program to block until the condition G is true, at which point statement S is immediately executed.

$sequence := \epsilon$

do forever
  when received $< c, op >$:
    $sequence := sequence \circ op$
    send $sequence$ to client c
od

Figure 2.2: Protocol executed by server s

For example, the condition "received res from s" causes the function to block until a response sent by s is received from the underlying message system. In addition, when this response is received, the value of res is set to the response received from s.

The above implementation guarantees that if a correct client initiates a request, then the client will eventually receive a response since s is non–faulty. We now require any History system $\mathcal{H}$ to "behave" like the one implemented on this non–faulty server s. More formally, let the *run* of any History system consist of two kinds of events— $\mathtt{send}(c, op, t)$ if client c initiates $\mathtt{REQUEST}(c, op)$ at time t, and $\mathtt{recv}(c, op, res, t)$ if client c receives a response res at time t to the above request. We assume that a client initiates a new request only after the response to the earlier request (if any) has been received. Let $\Omega_{\mathcal{H}_s}$ be the set of all infinite runs of an history system implemented on s, and let $\Omega_{\mathcal{H}}$ be the set of infinite runs of any other History system $\mathcal{H}$. Then our criterion for correctness of $\mathcal{H}$ is:

Consistency The History system $\mathcal{H}$ is equivalent to the one implemented on the non–faulty server s, *i.e.* $\Omega_{\mathcal{H}} \subseteq \Omega_{\mathcal{H}_s}$.

Notice that consistency captures both safety and liveness.

2.2 Impossibility Result

We now show that it is impossible to construct $\mathcal{H}$ in an asynchronous message passing environment even if at most one of the clients or the servers can fail by crashing. We assume that between any two processes, there is a reliable FIFO link.

In order to prove our impossibility result, we need to first define the *consensus* problem. Consider a system consisting of $n > 1$ processes, where each process might be *correct* or *faulty*. Each process p has an initial value v_p, and p may irrevocably *decide* on a value based on the initial values. Consensus must satisfy the following three properties:

Termination: Every correct process eventually decides some value.

Validity: If a process decides v, then v is the initial value of some process.

Agreement: No two correct processes decide differently.

It has been shown in [FLP85,DDS87] that there is no deterministic solution to consensus in an asynchronous environment that is subject to even a single crash failure. Suppose (for contradiction) now that $\mathcal{H}$ can be constructed using $n_s \geq 1$ servers. We show how to derive a consensus protocol using $\mathcal{H}$ for the $n = n_s + 2$ process system consisting of the n_s servers and 2 clients (say c_i and c_j) that tolerates a single crash failure. The impossibility of consensus now shows that such a system $\mathcal{H}$ cannot exist, proving our result.

The two client processes run the consensus protocol in Figure 2.3 (for some implementation of `REQUEST` that $\mathcal{H}$ provides) and each of the n_s servers, in addition of running the protocol corresponding to $\mathcal{H}$, run the consensus protocol in Figure 2.4.

$res :=$ REQUEST(c_i, v_i)
if $res = v_i$ **then**
  decide v_i
  send (decide, v_i) to all servers
else
  Let res be of the form $v \circ v_i$
  decide v
  send (decide, v) to all servers

Figure 2.3: Protocol run by client c_i with initial value v_i (symmetric for c_j)

when received (decide, v) :
  decide v

Figure 2.4: Consensus protocol executed by each server, in addition to executing the protocol corresponding to $\mathcal{H}$

A client first sends a request corresponding to its initial value to the service $\mathcal{H}$, and waits for a response. Since $\mathcal{H}$ returns the sequence of requests that have been received, client c_i (symmetric for c_j) can receive only two possible responses—v_i or $v_j \circ v_i$, where v_j is the initial value of c_j. In the first case, c_i decides on v_i and in the other case it decides on v_j (intuitively, the decision value corresponds to the initial value of the client who "first" sent its request). Client c_i then sends its decision value to the servers, and the servers decide on the value they receive.

We now informally argue that the above protocol satisfies Termination, Validity and Agreement and thus solves consensus. The formal proof appears in the next section.

Since, there can be only a single failure, at least one client (say c_i) is correct and therefore c_i eventually decides. It will send its decision value to all the correct servers, and so all the correct servers also decide. Thus, Termination holds. Validity holds trivially since each process can only decide on v_i or v_j. Agreement can be violated only if the the two clients send different decision values to the servers. However, this cannot happen because if client c_i receives the response v_i, then c_j can only receive the response $v_i \circ v_j$, and vice versa.

2.2.1 Formal proof of correctness

We now formally show that the protocol in Figures 2.3 and 2.4 satisfies Termination, Validity and Agreement and thus solves consensus.

Lemma 2.2.1 *The only responses that client c_i (symmetric for c_j) can receive are v_i and $v_j \circ v_i$.*

Proof: Follows from Consistency. □

Theorem 1 *The protocol in Figures 2.3 and 2.4 satisfies* **Termination**.

Proof: From Consistency, a correct client eventually gets a response to `REQUEST`. Therefore, from Lemma 2.2.1 and Figure 2.3, a correct client decides.

There must exist a correct client (say c_i) since there can be at most one failure. Since c_i decides, it must send $(\texttt{decide}, -)$ to all the correct servers, who will, therefore, also eventually decide (from Figure 2.4). Thus, Termination holds. □

Theorem 2 *The protocol in Figures 2.3 and 2.4 satisfies* **Validity**.

Proof: Follows from Lemma 2.2.1 and Figures 2.3 and 2.4. □

Theorem 3 *The protocol in Figures 2.3 and 2.4 satisfies* **Agreement**.

Proof: We first show that the two clients cannot decide differently. If $v_i = v_j$, then the result holds from Lemma 2.2.1 and Figure 2.3. Therefore, assume that $v_i \neq v_j$.

If client c_i (symmetric for c_j) decides v_i, then from Lemma 2.2.1 and Figure 2.3, it must have received the response v_i. From Consistency and Lemma 2.2.1, c_j can only receive the response $v_i \circ v_j$, and therefore, can only decide on v_i. On the other hand, if c_i decides v_j, then it must have received the response $v_j \circ v_i$. In this case, c_j can only receive the response v_j, and therefore, can only decide on v_j.

We next show that the correct servers decide as any correct client (say c_i). If c_i decides on v ($v \in \{v_i, v_j\}$), c_i must send $(\texttt{decide}, v)$ to all the servers. By the above argument, c_j can only decide (if it ever decides) on v, and therefore cannot send $(\texttt{decide}, v')$, $v' \neq v$ to the servers. Agreement now follows. □

Thus, we have the following impossibility result.

Theorem 4 *There is no implementation of the History system that tolerates even a single crash failure in an asynchronous environment.*

Chapter 3

Specification of the Primary–Backup Approach

As we mentioned in Chapter 1, before we can discuss our lower bound and upper bound results, we need to give a precise specification of the primary–backup approach. Our specification consists of four properties—Pb1–Pb4. Section 3.1 gives a formal statement of the properties Pb1–Pb3, but only an informal (and more intuitive) statement of Pb4. The description in Section 3.1 is sufficient to describe our lower bounds, and we need a formal description of Pb4 (given in Section 3.2) only when we discuss upper bounds. The reader may, therefore, choose to omit Section 3.2 in the first reading, and come back to it later before reading Chapter 5.

3.1 Specification

The lower bound and upper bounds results presented later in this thesis apply to any protocol that satisfies the following four properties. The first property states that no more than one server can be the primary at any time.

Pb1: There exists a local predicate $Prmy_s$ on the state of each server s. At any time, there is at most one server s whose state satisfies $Prmy_s$.

For brevity, whenever we say that "s is the primary (at time t)" we mean that the state of s satisfies $Prmy_s$ (at time t). Note that we do not specify any constraints on $Prmy_s$ except that each server can locally determine whether it is the primary.

The second property states that a client sends requests only to the server it believes to be the primary.

Pb2: Each client i maintains a server identity $Dest_i$ such that to make a request, client i sends a message (only) to $Dest_i$.

Pb2 therefore implies that the request can be received by at most one server. Of course, if client i gets notified that $Dest_i$ has failed, then it can update $Dest_i$ and direct subsequent requests to another server—the server pointed to by the new value of $Dest_i$.

For the next property, we model the communications network by assuming that client requests are enqueued in a message queue of a server. This property then states that only the primary can enqueue client requests.

Pb3: If a client request arrives at a server that is not the primary, then that request is not enqueued (and is therefore not processed by the server).

It may appear that Pb3 and Pb1 eliminate the need for Pb2. However, this is not the case because the primary can change while the client is sending the (multiple) copies of the request, resulting in the request getting enqueued (and therefore received) at more than one site. Some of our lower bounds will not hold in this case.

Properties Pb1–Pb3 specify a protocol for client interactions with a service, but not the semantics of the service. For example, the properties do not rule out a primary that ignores all requests. We eliminate such trivial implementations as follows.

A request sent to a primary–backup service can be lost if it is sent to a faulty primary. Periods during which requests are lost, however, are bounded by the time required for a backup to take over as the new primary. Such behavior is an instance of what we call *bofo* (*bounded outage finitely often*). Informally, we say that an *outage* occurs if some client sends a request but does not receive a response. A *(k, Δ)–bofo server* is one for which all outages can be grouped into at most k periods, each period having duration of at most Δ.[1] The final property of the primary–backup protocols is that they implement a (k, Δ)–bofo server for some values of k and Δ.

Pb4: There exist fixed and bounded values k and Δ such that the service behaves like a single (k, Δ)–bofo server.

A more formal statement of Pb4 appears in Section 3.2. Given Pb4, the `REQUEST` function described in Chapter 2 can be implemented by retrying a request until a response is received. Since k and Δ are both bounded, such a response will eventually be received.

3.2 Formal statement of Pb4

We now give a more formal statement of Pb4. We model execution of any client–server system by a *run*, which is a sequence of timestamped events involving clients,

[1]Therefore, as well as being finite, the number of such periods of service outage is also bounded (by k).

servers, and the message queues. These events include sending messages, enqueuing messages, receiving messages, and modeling computation at processes. Furthermore, define the *response time* of a system as follows: if a client sends a request and receives the corresponding response, then the interval between sending the request and receiving the response is never larger than the response time.

Property Pb4 requires that a primary–backup service behaves like a (k, Δ)–bofo server for some values of k and Δ. In order to do this, we will define "behaves like" in terms of indistinguishability as seen by clients—to the clients, the responses from the service appear as if they were sent by some (k, Δ)–bofo server. One consequence of this definition of indistinguishability is that it requires the clients to see a response time of 2δ (if a response is received) because this would be the response time of a bofo server in a system with message delivery time of δ.[2] However, the response time of a primary–backup service can be greater than 2δ because the primary may need to communicate with the backups before it can respond to a client. Thus, it would seem that the clients would be able to distinguish the service from a bofo server. The way we account for this is by adding another parameter, denoted ρ, to the definition of a bofo server, and requiring the primary–backup service to have like a (ρ, k, Δ)–bofo server. Informally, this means that to the clients, the service behaves like a bofo server running is a system with message delivery time of ρ. Clients expect the service to have a response time of 2ρ, and by defining ρ to be large enough, we can make the response time of the primary–backup service equal to 2ρ. In fact, our results can be optimized so that the clients expect the service to have a response time of $\rho + C + \delta$, where C is some constant that equals the

[2] For simplicity, we assume in this paper that any bofo server running in a system with message delivery time of δ has a response time of 2δ. However, our results can be generalized to servers whose response time is some other function of δ.

blocking time of the primary–backup protocol. This would, however, require that we model the bofo server to be running in an asymmetric system—the message delay from the client to the bofo server is ρ, and from the bofo server to the client is $C + \delta$. In order to simplify the presentation, we do not address this optimization.

We now define a (ρ, k, Δ)–bofo server more precisely. An outage of a bofo server P occurs at time t iff a client sends a request to P at time $t - \rho$ that is not enqueued by P by time t, or if P attempted to send a response at time t but omitted to do so. P is a (ρ, k, Δ)–bofo server iff in all runs of this server, the message delivery time is ρ, and all outages of P can be grouped in at most k intervals, with each interval having a duration no longer than Δ.

Let Ω_P be the set of runs of any (ρ, k, Δ)–bofo server P for some values of ρ, k and Δ, and Ω_{PB} be the set of runs of a primary–backup service PB. Let $\sigma_P \in \Omega_P$ and $\sigma_{PB} \in \Omega_{PB}$ be any two runs in the respective sets. We say that for some client i, $\sigma_P \approx_i \sigma_{PB}$ iff

1. client i sends a request m at time t in σ_P iff client i sends the same request m at time t in σ_{PB} and

2. client i receives a response r at time t in σ_P iff client i receives the same response r at time t in σ_{PB}.

We can now formally state Pb4.

Pb4: Let Ω_P be the set of runs of any (ρ, k, Δ)–bofo server P for some values of ρ, k and Δ, and Ω_{PB} be the set of runs of the primary–backup service PB. Then $\exists P : \forall \sigma_{PB} : \sigma_{PB} \in \Omega_{PB} : \exists \sigma_P : \sigma_P \in \Omega_P : \forall i : \sigma_P \approx_i \sigma_{PB}$

Chapter 4

Lower Bounds

In this chapter, we give the lower bounds for the degree of replication, blocking time and failover time for the primary–backup protocol specification in Chapter 3.

Recall that a *run* of a client–server system is a sequence of timestamped events involving clients, servers, and message queues. We say that two runs σ_1 and σ_2 of the system are *indistinguishable* to a process p if the same sequence of events (with the same timestamps) occur at p in both σ_1 and σ_2. We assume that servers are deterministic: if two runs σ_1 and σ_2 are indistinguishable to server s and s has the same initial state in both runs, then at any time t the state of s at t in σ_1 is the same as the state of s at t in σ_2. We make this assumption in order to simplify the discussion—our results hold for non-deterministic servers as well.

We also assume that the clients can send any request at any time. If we impose restrictions on the behavior of the clients, then we can derive protocols that violate the lower bounds in this thesis.

Define $\prec$ to be the *potential causality* relation [Lam78] on server events e_1 and e_2. Thus $\prec$ is the transitive closure of the following relation $\leadsto$: $e_1 \leadsto e_2$ iff both

e_1 and e_2 occur at the same server s and e_1 occurs before e_2, or e_1 is a send event and e_2 is the corresponding receive event.

We assume that clients can send two kinds of requests—*update* and *non–update.* Informally, an update request m changes the state of the service in such a way that the responses to subsequent requests depend on m. On the other hand, if m is a non–update request, then responses to subsequent requests need not depend on m. More formally, let m be a request with associated response r, and $e(m)$ and $e(r)$ be the events in the run associated with the receipt of m and the sending of r respectively. If m is an update request, then for all request/response pairs m'/r', where m' was sent after r, $e(m) \prec e(r')$. We will assume that update requests exist.

4.1 Bounds on Replication

This section gives the lower bounds on the degree of replication *i.e.* the relationship between n_s and f. The first bound is obvious.

Theorem 5 *Any primary–backup protocol tolerating f crash failures or send–omission failures requires $n_s \geq f + 1$.*

The following lemma is used in the rest of the theorems in this section.

Lemma 4.1.1 *Consider any protocol that satisfies* Pb4. *Suppose two disjoint and nonempty sets of servers A and B can be found that meet the following three properties:*

1. *There exists a run σ_a containing $R > 2k\lceil\Delta/d\rceil$ requests where d is the minimum time between the sending of any two client requests ($d > 0$). Further-*

more, in this run the servers in A do not crash and all other servers crash at time 0.

2. *There exists a run σ_b containing R requests. Furthermore, in this run the servers in B do not crash and all other servers crash at time 0.*

3. *There exists a run σ_{ab} containing R requests. Furthermore, the servers in A and B do not crash, σ_{ab} is indistinguishable from σ_a to all servers in A, and σ_{ab} is indistinguishable from σ_b to all servers in B.*

Then, at least one of the above runs violates Pb2.

Proof: Suppose for contradiction that the lemma is false and runs σ_a, σ_b and σ_{ab} all satisfy Pb2.

By Pb4, in σ_a at least $R - k\lceil\Delta/d\rceil$ of the requests must have been received by servers in A. This is because the number of requests that cannot have responses must fall into at most k intervals of length at most Δ, and each interval of Δ can contain at most $\lceil\Delta/d\rceil$ requests.

Similarly, in σ_b, at least $R - k\lceil\Delta/d\rceil$ of the requests must have been received by servers in B. Finally, since σ_{ab} is indistinguishable from σ_a to servers in A, they must execute the same number of receive events in both runs. The same holds for the servers in B. By Pb2, each request is sent to at most one server and so at least $2(R - k\lceil\Delta/d\rceil)$ requests must have been sent in σ_{ab}. Since only R requests were sent, we must have $R \geq 2(R - k\lceil\Delta/d\rceil)$, or $R \leq 2k\lceil\Delta/d\rceil$, which contradicts the assumption that $R > 2k\lceil\Delta/d\rceil$. □

Theorems 6 and 7 depend on two parameters of primary–backup protocols. Let Γ be the maximum time that can elapse between any two successive requests from non-faulty clients. Let D bound the time it takes for a client to learn the identity of a new server, so if some server s becomes the primary at time t_0 and remains the primary through time $t \geq t_0 + D$ when a correct client c_i sends a request, then $Dest_i = s$ at time t. We assume that if Γ is bounded then $D < \Gamma$, which implies that the service must be able to detect the failure of a primary and disseminate the new primary's identity to the clients without using messages from the clients. The condition $D < \Gamma$ is necessary in order to prove our lower bounds, because as we show in Section 4.4.1, one can construct a protocol that violates our lower bound if $D \geq \Gamma$.

With both send-omission failures and crash+link failures, messages may fail to reach their destinations. The following theorem shows that crash+link failures are more expensive to tolerate, because they require more replication.

Theorem 6 *Suppose there is at most one link between any two servers. Then any primary–backup protocol tolerating f crash+link failures and having $D < \Gamma$ requires $n_s \geq f + 2$.*

Proof: For contradiction, assume the existence of a protocol P with $n_s < f + 2$. We will show that P has three runs σ_a, σ_b and σ_{ab} that satisfy the conditions of Lemma 4.1.1. From the lemma, at least one of these runs violates Pb2, which implies that P cannot be a primary–backup protocol.

Let A be a set containing the one server s_a and let B be the set of remaining servers. Since $|A| = 1$ and $|B| = n - 1 \leq f$, A and B can become disconnected by link failures.

We first construct the run σ_{ab} in which no server crashes, postulating that the links between the servers in A and B are faulty and do not deliver any messages.

As required by Lemma 4.1.1, clients send a total of $R > 2k\lceil\Delta/d\rceil$ requests. Let $0 < d \leq \Gamma - D$ be the minimum interval between any two such requests. We postulate that a request will be sent at time t iff no request has been sent during the interval $[t - d..t)$ and one of the following rules hold.

1. A server s is the primary during the interval $[t - D..t]$. This request arrives immediately and is enqueued (at s, by Pb3 and the definition of D).

2. There is no primary at time t. This request arrives immediately and by Pb3 will never be enqueued at any server.

3. A server s is the primary at time t but another server s' is the primary immediately after time t. If this request is sent to s, then it arrives after t, and if it is sent to any other server, then it arrives immediately. In both cases, it arrives at a server that is not the primary, and so will not be enqueued (again by Pb3).

Note that, by construction, the maximum interval between any two client requests is $D + d$. This interval occurs when a server s becomes the primary just before d after a client message is sent, and s remains the primary for at least D. Hence, the client will be able to send R requests within time $R(D + d)$. This completes the construction of σ_{ab}.

We now construct σ_a and σ_b, recalling that in σ_a all of the servers except s_a crash at time 0, and in σ_b server s_a crashes at time 0. The clients send the same requests and at the same times in σ_a and in σ_b as in σ_{ab}. Furthermore, we construct σ_a and σ_b in such a way that these requests will arrive at the servers according to the same rules used in constructing σ_{ab}.

Since s_a does not receive any messages from servers in B in either σ_{ab} or σ_a, these two runs are indistinguishable to s_a as long as it receives the same client requests at the same times in both runs. We show that this is the case by showing that the same client requests are enqueued, at the same time, at s_a in both the runs.

Assume that this is true for the first $i-1$ client requests that are enqueued and assume that the i^{th} client request m is enqueued at time t. Since the i^{th} request is enqueued only at time t, the two runs are indistinguishable to s_a until time t.

We first show that if m is enqueued at s_a in σ_{ab}, then it is enqueued at the same time in σ_a as well. Since m was enqueued at s_a, m must have been sent by rule 1. By rule 1, s_a must have been the primary through $[t - D..t]$ in σ_{ab} and therefore, by indistinguishability, in σ_a as well. By the definition of D, m would have been enqueued at s_a at time t in σ_a as well.

Assume now that m was enqueued at s_a at time t in σ_a and m was sent at time $t' \leq t$. Suppose m was sent by rule 2. Then, by construction, there was no primary at time t' in σ_{ab}, and therefore, by indistinguishability, in σ_a as well. This implies (by Pb3) that m couldn't have been enqueued at s_a, a contradiction. Now suppose that m was sent by rule 3. First assume that $s = s_a$, where s is the server as defined in rule 3. Then by the construction in rule 3, m arrives immediately after t' in σ_a. In σ_{ab}, s_a was not the primary immediately after t', and so by indistinguishability, in σ_a as well. This implies that m couldn't have been enqueued at s_a, a contradiction. So $s \neq s_a$. Then, by construction in rule 3, m arrives immediately. Since s_a was not the primary at time t' in σ_{ab} (s was the primary), by indistinguishability, it is not the primary in σ_a as well. This again implies that m couldn't have been enqueued at s_a, a contradiction.

Therefore, the only possible rule by which m could have been sent is rule 1. Since m was enqueued at s_a, s_a is the primary at time t in σ_a (by Pb3). By indistinguishability, s_a is the primary in σ_{ab} as well. However, since m was sent by rule 1, s_a must have been the primary through $[t - D..t]$ in σ_{ab}. By the definition of D, m would have been enqueued at s_a at time t' in σ_{ab} as well and we are done.

We can similarly prove that σ_b and σ_{ab} are indistinguishable to the servers in B. Thus, by Lemma 4.1.1 P cannot be a primary–backup protocol. □

The next theorem states that additional replication is required in order to tolerate receive-omission failures. The proof is similar to that of Theorem 6, and so we will omit the details which are common to the two proofs.

Theorem 7 *Any primary–backup protocol tolerating f receive-omission failures and having $D < \Gamma$ requires $n_s > \lfloor \frac{3f}{2} \rfloor$.*

Proof: For contradiction, assume the existence of a protocol P with $n_s \leq \lfloor \frac{3f}{2} \rfloor$. We will show that P has three runs σ_a, σ_b and σ_{ab} that satisfy the conditions of Lemma 4.1.1. From the lemma, at least one of these runs violates Pb2, which implies that P cannot be a primary–backup protocol.

Let A and B be two disjoint sets containing up to $\lfloor \frac{f}{2} \rfloor$ servers each, and C be a set containing the rest of the servers.

We first construct the run σ_{ab} in which no server crashes, postulating that the servers in A and B are faulty by receive–omission (we can do this since $|A \cup B| \leq f$). The servers in A do not receive any messages from servers in $B \cup C$ and servers in B do not receive any messages from servers in $A \cup C$. As required by Lemma 4.1.1, clients send a total of $R > 2k\lceil \Delta/d \rceil$ requests in σ_{ab}, and these requests are sent according to the same three rules as in Theorem 6. This finishes the construction of σ_{ab}.

We now construct σ_a and σ_b, recalling that in σ_a all of the servers except those in A crash at time 0, and in σ_b all of the servers except those in B crash at time 0. Again, the construction of these two runs is exactly the same as in Theorem 6, and so we omit the details.

Since $s_a \in A$ does not receive any messages from servers in $B \cup C$ in either σ_{ab} or σ_a, these two runs are indistinguishable to s_a as long as it receives the same client requests at the same times in both runs. We can show that this is the case by exactly the same argument as in Theorem 6. Similarly for $s_b \in B$. Thus, by Lemma 4.1.1 P cannot be a primary–backup protocol and we are done. □

The next lower bound holds independent of the relation between D and Γ.

Theorem 8 *Any primary–backup protocol tolerating f general-omission failures requires $n_s > 2f$.*

Proof: Assume for contradiction that there is a protocol for $n_s \leq 2f$. Partition the servers into two disjoint sets A and B of size at most f each. We will construct two runs σ_1 and σ_2. In each run, one set of servers will be faulty and the other set will be non-faulty.

σ_1: The servers in A are faulty and fail to communicate with all servers in B, but behave correctly otherwise. Clients send update requests until the first response is sent (this must happen, by Pb4). Assume that the first response r to an update request m is sent at time t. Say that this response is sent by server s.

σ_2: The same as σ_1 up to time t, but if s is in B, then in σ_2 it is the servers in B that are faulty and fail to communicate with all servers in A rather than the servers in A that are faulty. In either case, r is sent by a faulty server. Furthermore, no server can distinguish σ_1 from σ_2 through time t and therefore, the first response r is sent at time t in σ_2 as well.

Let all of the faulty servers in σ_2 crash immediately after r is sent and have clients continue to send requests until another response r' is sent. This response must have been sent by a non-faulty server which implies that $\neg(e(m) \prec e(r'))$. However this violates the fact that m is an update request. □

4.2 Bounds on Blocking

This section gives the lower bounds on blocking times. Informally, a *blocking* primary-backup protocol is one in which the primary must, after receiving a request m, either receive a message from another server or simply wait an interval before it can respond to m. More formally:

Definition 4.2.1 C**–blocking protocol:** *Consider a failure-free run of a primary-backup protocol. Let the time that a request is received be* t_m*, and the time that the corresponding response is sent be* t_r*. We say that this protocol is* C–blocking *if* $t_r - t_m \leq C$*.*

For example, any primary-backup protocol in which the primary sends information about a request to the backups and waits for acknowledgement before sending the response to the client will be at least 2δ–blocking.

We now give the lower bounds on C for the various models of failures. However, before we do so, we prove the following lemma.

Lemma 4.2.1 *Consider a primary–backup system in which $D < \Gamma$ and the n_s servers can be divided into two disjoint sent S and S'. Assume that two runs σ and σ' exist with the following properties:*

- *The two runs are identical until time t_1.*
- *In σ, the servers in S' crash at time t_1.*
- *In σ', the servers in S do not get any messages from servers in S' that were sent after time t_1.*
- *Let t_2 be some time greater than t_1. In the interval $[t_1..t_2]$, clients send the same requests (at the same time) in both σ and σ'. Furthermore, these requests are sent to the same servers in both the runs, and they arrive at the same times in both the runs.*
- *After time t_2, the clients send requests in σ until a response occurs. These requests ares sent at intervals of at least d where $0 < d \leq \Gamma - D$ as follows. A request is sent at time t if no request has been sent in $[t - d..t)$ and one of the following rules hold.*
    1. *A server $s \in S$ is the primary during the interval $[t - D..t]$. This request arrives immediately and is enqueued (at s, by* Pb3 *and the definition of D).*
    2. *There is no primary in S at time t. This request arrives immediately and by* Pb3 *will never be enqueued at any server in B.*

3. A server $s \in S$ is the primary at time t but another server $s' \in S$ is the primary immediately after time t. If the request is sent to s, then it arrives after t, and if it is sent to any other server it arrives immediately. In both cases, it arrives at a server that is not the primary, and so will not be enqueued (again, by Pb3*).*

- *After time t_2, the clients send requests in σ' at the same times as they were sent in σ. Furthermore, these requests arrive according to the same three rules above.*

Then, the runs σ and σ' are indistinguishable to the servers in S.

Proof: Since $s_b \in S$ does not receive any messages from servers in S' that were sent after t_1 in either σ or σ', these two runs are indistinguishable to s_b as long as it receives the same client requests at the same times in both runs. We show that this is the case by showing that the same client requests are enqueued, at the same time, at s_b in both the runs.

Assume that this is true for the first $i-1$ client requests that are enqueued and assume that the i^{th} client request m is enqueued at time t. Since the i^{th} request is enqueued only at time t, the two runs are indistinguishable to s_b until time t.

We first show that if m is enqueued at s_b in σ, then it is enqueued at the same time in σ' as well. This is easily true for any request sent until time t_2, because these requests are sent to the same servers in both the runs. So we assume that m was sent after t_2. Since m was enqueued at s_b, m must have been sent by rule 1 in the lemma. By rule 1, s_b must have been the primary through $[t - D..t]$ in σ and therefore, by indistinguishability, in σ' as well. By the definition of D, m would have been enqueued at s_b at time t in σ' as well.

Assume now that m was enqueued at s_b at time t in σ' and m was sent at time $t' \leq t$. Suppose m was sent by rule 2. Then, by construction, there was no primary in S at time t' in σ, and therefore, by indistinguishability, in σ' as well. This implies (by Pb3) that m couldn't have been enqueued at s_b, a contradiction. Now suppose that m was sent by rule 3. First assume that $s = s_b$, where s is the server as defined in rule 3. Then by the construction in rule 3, m arrives immediately after t' in σ'. In σ, s_b was not the primary immediately after t', and so by indistinguishability, in σ' as well. This implies that m couldn't have been enqueued at s_b, a contradiction. So $s \neq s_b$. Then, by construction in rule 3, m arrives immediately. Since s_b was not the primary at time t' in σ (s was the primary), by indistinguishability, it is not the primary in σ' as well. This again implies that m couldn't have been enqueued at s_b, a contradiction.

Therefore, the only possible rule by which m could have been sent is rule 1. Since m was enqueued at s_b, s_b is the primary at time t in σ' (by Pb3). By indistinguishability, s_b is the primary in σ as well. However, since m was sent by rule 1, s_b must have been the primary through $[t - D..t]$ in σ. By the definition of D, m would have been enqueued at s_b at time t' in σ as well and we are done. $\Box$

Crash failures and crash+link failures have the trivial lower bound of $C = 0$. We, therefore, start with the lower bound for receive–omission failures.

Theorem 9 *Any primary–backup protocol tolerating f receive-omission failures with $f > 1$, $n_s \leq 2f$ and $D < \Gamma$ is C–blocking for some $C \geq 2\delta$.*

Proof: For contradiction, suppose there is a primary–backup protocol for $n_s \leq 2f$ and $f > 1$ that is C–blocking where $C < 2\delta$. Partition the servers into two sets

A and B where $|A| = f$ and $|B| = n - f \leq f$. We construct three runs. In all three runs, assume that all server messages take δ to arrive.

σ_1: There are no failures and all client requests take δ to arrive. Moreover, clients send update requests until some update request m evokes a response r. Let m be received at time t_m by server $s_a \in A$ and r be sent at time t_r by a server $s_{a'} \in A$ ($s_{a'}$ could be the same as s_a). Notice that since the protocol is C–blocking where $C < 2\delta$, $t_r - t_m < 2\delta$. Also since, by construction, all requests take δ to arrive, all client requests sent after time $t_m + \delta$ will be received after time t_r.

σ_2: Identical to σ_1 until s_a receives m at time t_m. At this point in σ_2, all servers in A are assumed to crash and clients are assumed to send no request during the interval $[t_m + \delta..t_r]$. Finally, after time t_r clients are assumed to repeatedly send requests according to the three rules in Lemma 4.2.1 with $B = S$ and $A = S'$.

As required by Lemma 4.2.1, there will be a response (say r') in σ_2, and by construction it must be from a request sent by rule 1.

σ_3: The same as σ_2, except that the servers in A *do not* crash at time t_m. Instead, the servers in B commit receive failures on all messages sent after t_m by servers in A. Clients send requests at the same times as in σ_2 which arrive using the same rules as σ_2.

Now, consider these three runs. By construction, the runs are identical up to time t_m. Since all server messages take δ to arrive, clients cannot distinguish σ_1 and σ_3 through $t_m + \delta$, and so clients send the same requests to the same servers in both σ_1 and σ_3 through $t_m + \delta$. Similarly, since all server messages take δ to arrive, the servers in B cannot distinguish between σ_1 and σ_3 through $t_m + \delta$. Therefore, since $t_r - t_m < 2\delta$, s_a (the server that received request m at time t_m in σ_1) and

$s_{a'}$ (the server that sent response r at time t_r in σ_1) cannot distinguish between σ_1 and σ_3 through time t_r, and so $s_{a'}$ sends response r in σ_3 as well. We will now show that servers in B cannot distinguish σ_2 and σ_3, and so response r' also occurs in σ_3. However, $\neg(e(m) \prec e(r'))$ which will violate the assumption that m is an update request.

However, σ_2 and σ_3 can be shown indistinguishable to the servers in B by using Lemma 4.2.1 with $S = B$, $S' = A$, $t_1 = t_m$, $t_2 = t_r$, $\sigma = \sigma_2$ and $\sigma' = \sigma_3$. All the conditions of the lemma can be seen to be true, and we are done. □

In run σ_3 of the above proof, a correct primary (s_a in set A) becomes the backup, while a faulty server from set B becomes the primary in s_a's place. We show below that it is always possible to construct such a run.

Theorem 10 *Any primary–backup protocol tolerating f receive-omission failures with $n_s \leq 2f$ and $D < \Gamma$ has a run in which a correct primary cedes to a faulty backup.*

Proof: Partition the servers into two sets A and B where $|A| = f$ and $|B| = n - f \leq f$. We construct three runs. In all three runs, assume that all server messages take δ to arrive.

σ_1: There are no failures and all client requests take δ to arrive. Moreover, clients send requests until some request m evokes a response r. Let m be received at time t_m by server $s_a \in A$.

σ_2: Identical to σ_1 until s_a receives m at time t_m. At this point in σ_2, all servers in A are assumed to crash. After time t_m, clients are assumed to repeatedly send requests at intervals of at least d where $0 < d \leq \Gamma - D$ and these requests are sent according to the three rules in Lemma 4.2.1 with $S = B$ and $S' = A$.

As required by the lemma, there will be a response (say r') to some request m' in σ_2. Assume that m' was received by some server $s_b \in B$.

σ_3: The same as σ_2, except that the servers in A *do not* crash at time t_m. Instead, the servers in B commit receive failures on all messages sent after t_m by servers in A. Clients send requests at the same times as in σ_2 which arrive using the same rules as σ_2.

Now, consider these three runs. By construction, the runs are identical up to time t_m. Furthermore it can be seen that all the conditions in Lemma 4.2.1 are satisfied for $S = B$, $S' = A$, $t_1 = t_m$, $t_2 = t_m$, $\sigma = \sigma_2$ and $\sigma' = \sigma_3$. Therefore, σ_2 and σ_3 are indistinguishable to s_b and so response r' occurs in σ_3 as well. However, this implies that in σ_3 there was a primary s_a in A before time t_m, and there was a primary $s_b \in B$ after t_m. Since s_a is correct and s_b is faulty, are done. $\square$

The above theorem shows a disconcerting property: there does not exist a primary–backup protocol that tolerates receive-omission failures with $n_s \leq 2f$ in which a primary cedes only when it fails. Moreover, this lower bound is tight—in Chapter 5, we give a receive-omission primary–backup protocol with $n_s = 2f + 1$ in which a primary cedes only when it fails.

And, if $f = 1$, then the following theorem holds: its proof is similar to the proof of Theorem 9, except that $s_a = s_{a'}$.

Theorem 11 *Any primary–backup protocol tolerating receive-omission failures with $f = 1$ and $n_s \leq 2f$ and having $D < \Gamma$ is C–blocking for some $C \geq \delta$.*

Proof: For contradiction, suppose there is a primary–backup protocol for $n_s = 2$ and $f = 1$ that is C–blocking where $C < \delta$. Partition the servers into two sets A and B where $|A| = 1$ and $|B| = 1$. We construct three runs. In all three runs, assume that all server messages take δ to arrive. Clearly, in all these runs, if a server p receives a request and a server q sends the corresponding response, then $p = q$ as otherwise the assumption $C < \delta$ will be violated (the message from p to q will take δ to arrive).

σ_1: There are no failures and all client requests take δ to arrive. Moreover, clients send update requests until some update request m evokes a response r. Let m be received at time t_m by server $s_a \in A$ and r be sent at time t_r by s_a. Notice that since the protocol is C–blocking where $C < \delta$, $t_r - t_m < \delta$. Also since, by construction, all requests take δ to arrive, all client requests sent after time t_m will be received after time t_r.

σ_2: Identical to σ_1 until s_a receives m at time t_m. At this point in σ_2, the server in A (which is s_a) is assumed to crash and clients are assumed to send no request during the interval $[t_m..t_r]$. Finally, after time t_r clients are assumed to repeatedly send requests according to the three rules in Lemma 4.2.1 with $B = S$ and $A = S'$.

As required by Lemma 4.2.1, there will be a response (say r') in σ_2, and by construction it must be from a request sent by rule 1.

σ_3: The same as σ_2, except that the servers in A *do not* crash at time t_m. Instead, the server in B commits receive failures on all messages sent after t_m by the server in A. Clients send requests at the same times as in σ_2 which arrive using the same rules as σ_2.

Now, consider these three runs. By construction, the runs are identical up to time t_m. Since all messages sent until time t_r take δ to arrive and $t_r - t_m < \delta$, s_a (the server that received request m at time t_m in σ_1) cannot distinguish between σ_1 and σ_3 through time t_r, and so s_a sends response r in σ_3 as well. We will now again show that the server in B cannot distinguish σ_2 and σ_3, and so response r' also occurs in σ_3. However, $\neg(e(m) \prec e(r'))$ which will violate the assumption that m is an update request.

However, σ_2 and σ_3 can be shown indistinguishable to the server in B by using Lemma 4.2.1 with $S = B$, $S' = A$, $t_1 = t_m$, $t_2 = t_r$, $\sigma = \sigma_2$ and $\sigma' = \sigma_3$. All the conditions of the lemma can be seen to be true, and we are done. □

Primary–backup protocols tolerating send-omission or general–omission failures exhibit the same blocking properties as those tolerating receive-omission failures, except that the restriction $D < \Gamma$ is no longer necessary. Here we prove just the results for send–omission failures. The results for general–omission failures then follow.

Theorem 12 *Any primary–backup protocol tolerating f send-omission failures with $f > 1$ is C–blocking for some $C \geq 2\delta$.*

Proof: For contradiction, suppose there is a primary–backup protocol that is C–blocking where $C < 2\delta$. We consider the following two runs in which all server messages take δ to arrive.

σ_1: There are no failures and all client requests take δ to arrive. Moreover, clients send update requests until some update request m evokes a response r. Let m be received at time t_m by server s_a and r be sent at time t_r by a server then one can construct protocols that violate our lower bounds. The execution of one such protocol is given in Section 4.4.2.

Theorem 14 *Any primary–backup protocol tolerating f crash failures must have a failover time of at least $f\delta$.*

Proof: The proof is by induction on f.

Base case $f = 0$: trivially true since a failover time cannot be smaller than zero.

Induction case $f > 0$: Suppose the theorem holds for at most $f - 1$ failures, but (for a proof by contradiction) there is a protocol P for which the theorem is false when there are f failures. From the induction hypothesis, there is a run σ with at most $f - 1$ failures and an interval $[t_0..t_1]$ at least $(f - 1)\delta$ during which there is no primary. Let s_1 be the server that becomes the primary at t_1. Consider the two runs σ_1 and σ_2 that extend σ as follows:

σ_1: Assume s_1 crashes at time t_1. By assumption, there exists a new primary (say s_2) at time $t_2 < t_1 + \delta$. Since s_1 crashes at time t_1, s_2 does not receive any messages from s_1 that were sent after time t_1.

σ_2: Assume that s_1 is correct, there are no other crashes at or after t_1 and all messages sent by s_1 after time t_1 take δ to arrive.

Since s_2 cannot distinguish σ_1 from σ_2 through time t_2, s_2 becomes the primary at time t_2 in σ_2. By Pb5, however, s_1 remains the primary at time t_2 in σ_2. This violates Pb1, and so P is not a primary–backup protocol. □

Failover times for all other failure models have a larger lower bound:

Theorem 15 *Any primary–backup protocol tolerating f crash+link failures has a failover time of at least $2f\delta$.*

Proof: The proof is by induction on f.

Base case $f = 0$: trivially true.

Induction case $f > 0$: Suppose the theorem holds for at most $f - 1$ failures, but (for a proof by contradiction) there is a protocol P for which the theorem is false when there are f failures.

From the induction hypothesis, there is a run σ with at most $f - 1$ failures and an interval $[t_0..t_1]$ at least $2(f-1)\delta$ during which there is no primary. Let s_1 be the server that becomes the primary at t_1. Consider the three runs σ_1, σ_2 and, σ_3 that extend σ as follows:

σ_1: Assume that s_1 crashes at time t_1 and all messages sent after t_1 take δ to arrive. Furthermore, the crash of s_1 is the only failure at or after t_1. By assumption, there exists a new primary (say s_2) at time $t_2 < t_1 + 2\delta$. Since s_1 crashes at time t_1, s_2 does not receive any messages from s_1 that were sent after time t_1. Furthermore, since all messages take δ to arrive, any message that was sent at or after after $t_1 + \delta$ can be received by s_2 only after time t_2.

σ_2: Assume that s_1 is correct, there are no other failures at or after t_1, and all messages sent after time t_1 take δ to arrive. Since there are no failures at or after time t_1, by Pb5 p_1 continues to be the primary through time t_2.

σ_3: The same as σ_2 except that the link between s_1 and s_2 is faulty and does not deliver any message sent by s_1 to s_2 after time t_1.

By construction, s_2 cannot distinguish σ_1 from σ_3 through time t_2, and so s_2 becomes the primary at time t_2 in σ_3. Similarly, s_1 cannot distinguish σ_2 from σ_3 through time t_2 and so s_1 remains the primary until time t_2 in σ_3. This violates Pb1, and so P is not a primary–backup protocol. □

The proofs of Theorems 16 and 17 are similar to Theorem 15.

Theorem 16 *Any primary–backup protocol tolerating f receive-omission failures has a failover time of at least $2f\delta$.*

Proof: The proof is by induction on f.

Base case $f = 0$: trivially true.

Induction case $f > 0$: Suppose the theorem holds for at most $f - 1$ failures, but (for a proof by contradiction) there is a protocol P for which the theorem is false when there are f failures.

From the induction hypothesis, there is a run σ with at most $f-1$ failures and an interval $[t_0..t_1]$ at least $2(f-1)\delta$ during which there is no primary. Let s_1 be the server that becomes the primary at t_1. Consider the three runs σ_1, σ_2 and, σ_3 that extend σ as follows:

σ_1: Assume that s_1 crashes at time t_1 and all messages sent after t_1 take δ to arrive. Furthermore, the crash of s_1 is the only failure at or after t_1. By assumption, there exists a new primary (say s_2) at time $t_2 < t_1 + 2\delta$. Since s_1 crashes at time t_1, s_2 does not receive any messages from s_1 that were sent after time t_1. Furthermore, since all messages take δ to arrive, any message that was sent at or after $t_1 + \delta$ can be received by s_2 only after time t_2.

σ_2: Assume that s_1 is correct, there are no other failures at or after t_1, and all messages sent after time t_1 take δ to arrive. Since there are no failures at or after time t_1, by Pb5 s_1 continues to be the primary through time t_2.

σ_3: The same as σ_2 except that s_2 is faulty and does not receive any message sent by s_1 after time t_1.

By construction, s_2 cannot distinguish σ_1 from σ_3 through time t_2, and so s_2 becomes the primary at time t_2 in σ_3. Similarly, s_1 cannot distinguish σ_2 from σ_3 through time t_2 and so s_1 remains the primary until time t_2 in σ_3. This violates Pb1, and so P is not a primary–backup protocol. □

Theorem 17 *Any primary–backup protocol tolerating f send-omission failures has a failover time of at least $2f\delta$.*

Proof: The proof is by induction on f.

Base case $f = 0$: trivially true.

Induction case $f > 0$: Suppose the theorem holds for at most $f - 1$ failures, but (for a proof by contradiction) there is a protocol P for which the theorem is false when there are f failures.

From the induction hypothesis, there is a run σ with at most $f - 1$ failures and an interval $[t_0..t_1]$ at least $2(f-1)\delta$ during which there is no primary. Let s_1 be the server that becomes the primary at t_1. Consider the three runs σ_1, σ_2 and, σ_3 that extend σ as follows:

σ_1: Assume that s_1 crashes at time t_1 and all messages sent after t_1 take δ to arrive. Furthermore, the crash of s_1 is the only failure at or after t_1. By assumption, there exists a new primary (say s_2) at time $t_2 < t_1 + 2\delta$. Since s_1 crashes at time t_1, s_2 does not receive any messages from p_1 that were sent after time t_1. Furthermore, since all messages take δ to arrive, any message that was sent at or after $t_1 + \delta$ can be received by s_2 only after time t_2.

σ_2: Assume that s_1 is correct, there are no other failures at or after t_1, and all messages sent after time t_1 take δ to arrive. Since there are no failures at or after time t_1, by Pb5 s_1 continues to be the primary through time t_2.

σ_3: The same as σ_2 except that s_1 is faulty and does not send any messages to s_2 after time t_1.

By construction, s_2 cannot distinguish σ_1 from σ_3 through time t_2, and so s_2 becomes the primary at time t_2 in σ_3. Similarly, s_1 cannot distinguish σ_2 from σ_3 through time t_2 and so s_1 remains the primary until time t_2 in σ_3. This violates Pb1, and so P is not a primary–backup protocol. □

Clearly from Theorem 16 or Theorem 17, the lower bound on failover time for general–omission failures is also $2f\delta$.

4.4 Summary

Table 4.1 summarizes all of our lower bound results.

4.4.1 Protocol having $D \geq \Gamma$

We now describe a protocol that has $D \geq \Gamma$, and violates our crash+link lower bound for degree of replication. For this protocol, $f = 1$ and $n_s = 2$, rather than $n_s = 3$ as required by Theorem 6 (refer to Figure 4.1).

There are two servers s_1 and s_2, with s_1 being the initial primary. When s_1 receives a request from a client c, it computes the response, updates its state,

Table 4.1: Summary of lower bound results

Failure model	Degree of replication	Blocking time	Failover time
crash	$n_s > f$	0	$f\delta$
crash+link	$n_s > f + 1$ †	0	$2f\delta$
receive omission	$n_s > \lfloor \frac{3f}{2} \rfloor$ †	δ if $n_s = 2f$ and $f = 1$ † 2δ if $n_s \leq 2f$ and $f > 1$ † 0 if $n_s > 2f$	$2f\delta$
send omission	$n_s > f$	δ if $f = 1$ 2δ if $f > 1$	$2f\delta$
general omission	$n_s > 2f$	δ if $f = 1$ 2δ if $f > 1$	$2f\delta$

† $D < \Gamma$ assumed.

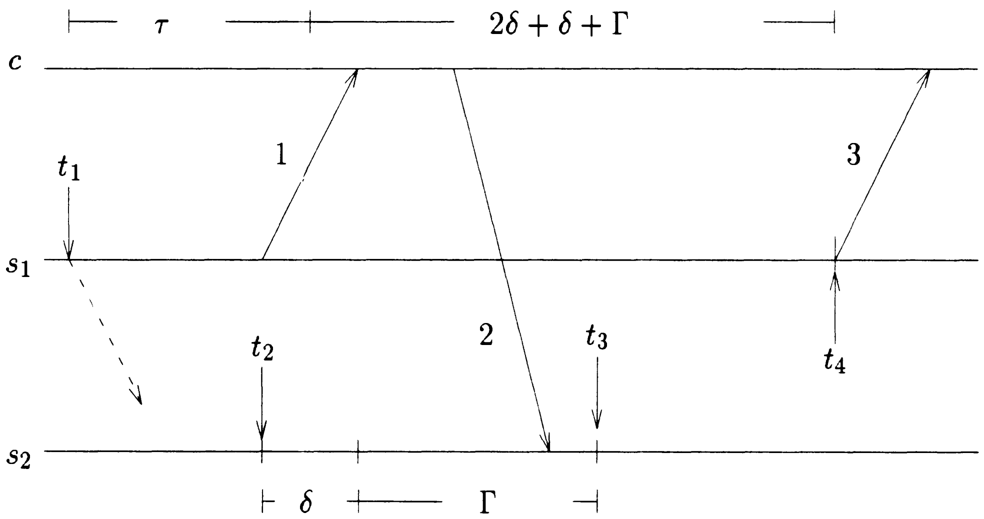

Figure 4.1: Protocol having $D \geq \Gamma$

sends a state update message to s_2, and then sends the response to the client. It is possible that due to link failures, s_2 does not get a state update message. However, as we will see, in this case s_2 will never process any requests from the client.

Server s_1 also sends periodic ("I am alive", $\ell\tau$) messages to s_2 at time $\ell\tau$ for $\ell = 0, 1, 2, \ldots$ (dashed arrow in the figure). Assume, for this protocol that $\tau > 2\delta$. Server s_1 then waits for s_2 to acknowledge these "I am alive" messages. Furthermore, s_1 processes no client requests during the period it is waiting for s_2 to acknowledge these messages.

If s_1 does not get an acknowledgement to an ("I am alive",$\ell\tau$) message (in the figure, $t_1 = l\tau$) by time $t_2 = l\tau + 2\delta$, then s_1 is correct because $f = 1$ and either s_2 has crashed, or the link between s_1 and s_2 has failed. However, in case the link has failed, then s_2 can also assume that s_1 has failed. Therefore, in order to inform s_2 that s_1 has not crashed, s_1 uses the client requests to inform s_1 of this fact. Server s_1 first becomes the backup, and then sends a $(Dest = s_2)$ message to the client (message 1 in the figure), informing the client to send further requests to s_2. As we will see in the next paragraph, s_2 will receive client requests and conclude that s_1 has not crashed. At time $t_4 = (l+1)\tau + 2\delta + \delta + \Gamma$, after giving enough time to s_2 to find out that s_1 has not crashed, s_1 agains becomes the primary and sends a $(Dest = s_1)$ message to the client (message 3 in the figure). Since s_1 is correct, it now continues to be the primary.

We now describe what s_2 does in case it does not get a ("I am alive",$\ell\tau$) message by time $l\tau + \delta$. Server s_2 becomes the primary at time $t_2 = l\tau + 2\delta$. Clearly, s_1 will not get an acknowledgement to the ("I am alive",$\ell\tau$) message (as shown in the figure). However, it is possible that s_1 did not get an acknowledgement to the ("I am alive",$(\ell - 1)\tau$) message as well because the acknowledgement that s_2 sent got lost due to a link failure. In either case, it can be seen from the description in the previous paragraph that s_1 would have become the the backup by time $l\tau + 2\delta$, and cannot become the primary again before time $l\tau + 2\delta + \delta + \Gamma$. Thus if s_1

has not crashed, then Pb1 will not be violated as long as s_2 does the remain the primary beyond $l\tau + 2\delta + \delta + \Gamma$. We will show that this is the case. After becoming the primary at time $l\tau + 2\delta$, s_2 waits until time $t_3 = l\tau + 2\delta + \delta + \Gamma$ to receive a client request. Again, from the previous paragraph, s_2 will receive such a request (message 2 in the figure) if s_1 has not crashed because s_1 would have informed the client by time $l\tau + 2\delta$ to send future requests to s_2. If s_2 receives such a request, then it concludes that s_1 has not crashed, and immediately becomes the backup.

On the other hand, if s_2 does not receive such a request, then s_1 must have crashed. In this case, s_2 is correct, and so it continues to be the primary. It sends a $(Dest_i = s_2)$ message to the client so that further requests are sent to s_2. Thus $D = \delta + \Gamma + \delta$ *i.e.* $D > \Gamma$. Furthermore, since s_1 is faulty, the link between s_1 and s_2 must be correct, which implies that s_2 must have received all the state update messages that s_1 would have sent before it crashed. Thus, the state of s_2 is consistent with the state of the previous primary, and so it can process requests from the client.

4.4.2 Protocol violating Pb5

As we had mentioned earlier, Pb5 is necessary in order to prove our lower bounds. We show this by giving a crash failure protocol for $n_s = 3$ and $f = 2$ that does not satisfy Pb5 and has a failover time of less than $2f\delta$ (refer to Figure 4.2).

The three servers are s_1, s_2 and s_3 with s_1 being the initial primary. Furthermore, s_1 will remain the primary until it crashes. When s_1 receives a request from a client c, it computes the response, updates its state, sends state update messages to s_2 and s_3, and then sends the response to the client without waiting for acknowledgements from the backups. For crash failures, a backup can detect the failure of a primary by the lack of an expected message from the primary. For example, if the clocks of the servers are perfectly synchronized, then the primary can send a ("I am alive", $\ell\tau$) message to the backups at time $\ell\tau$ for $\ell = 0, 1, 2, \ldots$

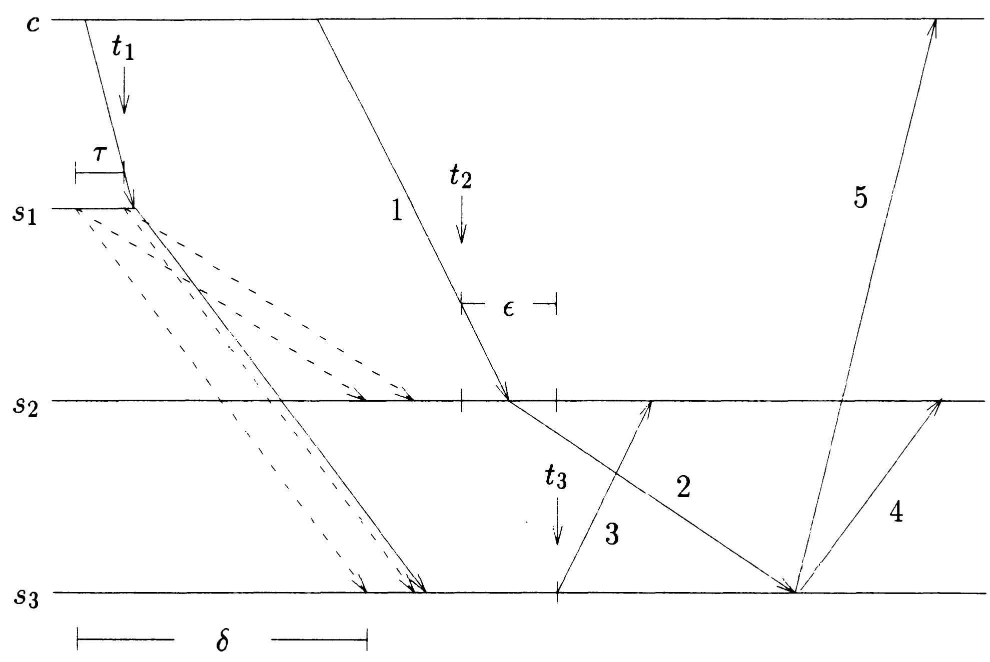

Figure 4.2: Protocol with $f = 2$ and Failover Time $< 2\delta$

If a backup does not receive this message by time $\ell\tau + \delta$, then the primary crashed at some time t_x in the range $(\ell - 1)\tau \leq t_x \leq \ell\tau$. The backups detect the failure of the primary by using this failure detection technique. The periodic "I am alive" messages are shown as dashed arrows in the figure.

If at some time $\ell\tau + \delta$, s_2 has not received an ("I am alive",$\ell\tau$) message from s_1, then s_2 knows that s_1 crashed at some time t_1: $(\ell - 1)\tau \leq t_1 \leq \ell\tau$ and so s_2 becomes the primary (time t_2 in the figure). Furthermore, if s_3 has not crashed, then it will also detect s_1's failure by not receiving an ("I am alive",$\ell'\tau$) message from s_1 where $\ell - 1 \leq \ell' \leq \ell + 1$. In the figure, s_1 has crashed just after sending the "I am alive" messages, receiving a client request, and sending the state update message to s_3 but not s_2.

The new primary s_2 remains the primary only through the interval $[\ell\tau + \delta..\ell\tau + \delta + \epsilon)$, for some $\epsilon > 0$, even if no more failures occur. This violates Pb5. During this interval, s_2 processes no requests that it receives from the clients. However, it remembers these requests (message 1 in the figure) and forwards them to be processed by s_3 (message 2 in the figure).

At time $(\ell' + 1)\tau + \delta + \epsilon$ (time t_3 in the figure), s_3 becomes the primary if it has not already crashed. Since $\ell\tau + \delta + \epsilon \leq (\ell' + 1)\tau + \delta + \epsilon$, s_2 is no longer the primary at this time as required by Pb1. s_3 first sends its state to s_2 (message 3 in the figure) in order to make the states of s_2 and s_3 equal. The new primary s_3 then starts processing client requests, including those forwarded by s_2. As before, it sends a state update message first to s_2 (message 4 in the figure) and then to the clients (message 5 in the figure).

Finally, after $t_2 + \epsilon$, s_2 updates its state on receiving the state and state update messages from s_3. Sometime later, if s_2 stops receiving "I am alive" messages from s_3, s_2 again becomes the primary. It first processes any requests that it had forwarded earlier to s_3 but s_3 did not process before crashing. Thereafter, s_2 simply processes new requests from the clients since there are no backups.

The longest period during which there will be no primary occurs when s_1 crashes at time $t_1 = (\ell - 1)\tau$, s_2 crashes at time $t_2 = \ell\tau + \delta$, and s_3 becomes the primary at time $t_3 = \ell\tau + \delta + \epsilon$, giving a failover time of $\tau + \delta + \epsilon$. If $\tau + \epsilon < \delta$ then the failover time is less than 2δ.

Chapter 5

Optimal Primary-Backup Protocols–Part I

In the following two chapters, we show which of our lower bounds derived in Chapter 4 are tight. The protocols (we call these δ–blocking protocols) showing that the lower bounds on blocking time for send–omission, receive–omission and general–omission failures are tight when $f = 1$ are quite different from the protocols that show the other lower bounds to be tight. We, therefore, address these δ–blocking protocols in the next chapter (Chapter 6) and concentrate on the remaining protocols here.

5.1 Protocols for the Clients and the Bofo Server

Property Pb4 requires that the primary–backup service behaves like some bofo server. Figure 5.1 gives a canonical server P, and our primary–backup protocols implements a (ρ, k, Δ)–bofo version of P for some values of ρ, k and Δ. Figure 5.2

gives the protocol that a client i uses to communicate with the bofo version of P. In both the figures, the statement "**when** $G : S$" causes the program to block until the condition G is true, at which point statement S is immediately executed. For example, the condition "received m from client c" causes a process to block until it gets a message m from the underlying message system. In addition, when this particular **when** terminates, the variable c contains the identifier of the client that sent the request.

```
initialize()
inform-clients("Dest = s")
do forever
  when received request from client c:
    response := Π(state, request)
    state = state ∘ response
    send response to client c
od

procedure initialize()
  state := ε

procedure inform-clients(ic)
  send (ic) to all clients
```

Figure 5.1: Protocol run by a canonical server P

5.2 The Primary–Backup Protocol Schema

We first make the simplying assumption that the links between the clients and the servers are non–faulty and there are no omission failures between the clients and the servers (*i.e.* only the links between servers can be faulty when assuming crash+link failures, and omission failures can occur only between servers when assuming omission failures). We show later how this assumption can be removed.

```
cobegin
  || do forever
       when received "Dest = s": Dest_i := s
     od
  || do forever
       ⋮
       if want to send request
          send request to Dest_i
          if not received response within 2ρ then
             recover() /* call some recovery procedure, which might retry */
          else
       ⋮
     od
coend
```

Figure 5.2: Protocol run by client i interacting with server s

Our primary–backup service consists of n_s servers $\{s_1, \ldots, s_{n_s}\}$, each of which runs the protocol in Figure 5.3. The protocol run by the clients is the same as the protocol for the bofo–server which is shown in Figure 5.2, with $\rho = n_c C + \delta$. We need this large value of ρ because, as we will see, our implementation of the primary–backup service has a response time larger than $C + 2\delta$ whenever $C > 0$. Also, note that this is just one possible value of ρ. For example, it is possible to define ρ such that it is independent of the number of clients, if we make the assumption that at any time, there are only a bounded number of requests in the message queue of the primary.

```
initialize(i)
cobegin
  || if i = 0 then primary(i) else backup(i)
  || delivery–process(i)
  || failure–detector(i)
coend
```

Figure 5.3: Protocol run by server s_i to emulate server P

The procedures **primary** and **backup** (shown in Figure 5.4) are the same for all the failure models. On the other hand, the implementation of the procedures **delivery–process**, **failure–detector**, **initialize** and **broadcast** (used in Figure 5.4), depend on the particular failure model being assumed. However, we ensure that these different implementations always satisfy a set of properties, listed below as B0—B11. We extracted these properties in order to make our proofs modular. In particular we prove that, independent of the failure model, the protocol in Figures 5.3 and 5.4 satisfies Pb1–Pb5, provided the four procedures satisfy B0—B11.

In the properties B0—B11, the values of the constants d, C and τ depend on the failure model. Intuitively, d corresponds to the amount of time that can elapse from the moment a message is broadcast to the moment it is dequeued by the receiver, C corresponds to the blocking time and τ corresponds to the periodicity of probing in the underlying failure detector.

When we say that a server "halts", we mean that either the server has crashed or has stopped executing the protocol by executing a **stop**. The array of booleans $Faulty_k$ indicates which servers s_k believes have halted: $Faulty_k[s_j]$ being true means that s_k believes s_j has halted. We define a broadcast by a server to be *successful* if the server does not halt during the corresponding execution of **broadcast**. Finally, whenever, we refer to "dequeue" in the following properties, we mean dequeueing from the queue $Rqueue_k$.

The properties can be subdivided according to the procedures they relate to:

```
procedure primary(j)
  inform-clients("Dest = s_j")
  broadcast((mylastlog, s_j, last(state_j)), j)
  /* broadcast(M, j) sends M to all servers, the sender being s_j */
  do forever
    when received request from client c:
      response := Π(state_j, request)
      state_j := state_j ∘ response
      broadcast((log, s_j, response), j)
      send response to client c
  od
  procedure backup(k)
    do forever
      ((tag, s_j, r), j) := Deq(Rqueue_k)
      /* The messages that are broadcast are enqueued in Rqueue_k
      /* Assume that dequeueing an empty queue
      does not return any sensible value of tag */

      /* synchronizing with the new primary */
      if tag = mylastlog then
        if r ∈ state_k then
          if r = last(state_k) then skip
          else state_k := state_k \ last(state_k)
        else state_k := state_k ∘ r

      /* logging response from primary */
      if tag = log then state_k := state_k ∘ r

      /* becoming the primary */
      if ∀i < k : Faulty_k[s_i] then primary(k)
    od
```

Figure 5.4: Procedures **primary** and **backup**

Properties of **broadcast** and **delivery–process**:

B0: If s_j dequeues b, then b must have been broadcasted before; and s_j dequeues b at most once.

B1: If s_j initiates a broadcast b' after it initiates broadcast b, then no server dequeues b' before it dequeues b.

B2: If s_j initiates a broadcast b at time t, then no server dequeues b after time $t + d$.

B3: No successful broadcast takes longer than C to complete. Furthermore,

- if $C = 0$ and s_j initiates a broadcast at time t and only halts (if it ever halts) after time t, then the broadcast is successful.
- if $C > 0$ and s_j initiates a broadcast at time t and does not halt by time $t + C$, then the broadcast is successful.

Properties of **failure–detector**:

B4: If $Faulty_j[s_k]$ becomes true, then it remains true, as long as s_j does not halt.

B5: The value of $Faulty_j[s_k]$ can only change at times $t = l\tau + d$ for some integer $l \geq 0$.

B6: If $Faulty_j[s_k] = true$ at time t then s_k has halted by time t.

B7: If s_k halts by time t, and s_j does not halt by time $t+\tau+d$, then $Faulty_j[s_k] = true$ by time $t + \tau + d$.

Properties of **broadcast** and **delivery–process** interacting with **failure–detector**:

B8: No correct server halts in procedures **initialize, broadcast, delivery–process** or **failure–detector.**

B9: If s_j initiates a successful broadcast at time t, then for all non–halted servers s_k, $k > j$, $Faulty_k[s_j] = false$ through time $\lceil \frac{t}{\tau} \rceil \tau + d$.

B10: If s_j initiates a successful broadcast b, then for every non–halted server s_k: $(Faulty_k[s_j] = true) \Rightarrow$ (s_k has dequeued b).

B11: If s_j initiates a broadcast b at time t and $s_k, k > j$ initiates a broadcast b', then either no server dequeues b after b', or $Faulty_k[s_j] = false$ through time $t + d$.

5.2.1 Proof of Correctness

We now show that the protocol in Figures 5.3 and 5.4 satisfies Pb1—Pb5 if the procedures **initialize**, **broadcast**, **delivery–process** and **failure–detector** satisfy B0—B11. We will use the notation $\uparrow k$ to denote the time s_k becomes the primary and $\downarrow k$ to denote the time s_k halts. Note that, from Figure 5.4, $\uparrow k$ is also the time when $\forall j < k : Faulty_k[s_j] = true$ first holds.

Lemma 5.2.1 *Let s_j receive a client request m at time t.*

- *if $C = 0$ and s_j only halts (if it ever halts) after time t, then s_j sends the response to m by time t.*
- *if $C > 0$ and s_j does not halt by time $t + C$, then s_j sends the response to m by time $t + C$.*

Proof: Follows from Figure 5.4 and B3. □

Lemma 5.2.2 *At any time, there can be at most one request from any client i in the queue of any server s_j.*

Proof: Assume, for contradiction, that t_2 is the first time that there are two requests (say m_1 and m_2) from some client i in the queue of s_j. Assume m_1 was sent at time t_1 and so it was enqueued by $t_1 + \delta$. Furthermore, since m_2 is the first duplicate request, there are at most $n_c - 1$ requests ahead of m_1 in s_j's queue. Therefore, from Figure 5.4 and B3, s_j will dequeue m_1 by time $t_1 + \delta + (n_c - 1)C$. Therefore, $t_2 \leq t_1 + \delta + (n_c - 1)C$. However, since $2\rho = 2(n_cC + \delta)$, from Figure 5.2, client i cannot send m_2 before time $t_1 + 2(n_cC + \delta)$ *i.e.* $t_2 \geq t_1 + 2(n_cC + \delta)$, which is a contradiction. □

Lemma 5.2.3 *If a request m is sent by a client at time t_m to server s_j, and s_j remains the primary through the interval $[t_m..t_m + \rho]$, then m is received by s_j by time $t_m + \rho$.*

Proof: Since the message delivery time is at most δ, m is enqueued at s_j by time $t_m + \delta$. From Lemma 5.2.2, there can be at most $n_c - 1$ requests ahead of m in s_j's queue. Therefore, from Figure 5.4 and B3, m is received by time $t_m + (n_c - 1)C + \delta$. Since $\rho = n_cC + \delta$, the lemma holds. □

Lemma 5.2.4 *If a client sends a request at time t and a response is sent, then the response will arrive at the client by time $t + 2\rho$.*

Proof: Follows from Figure 5.4, Lemma 5.2.3 and B3. □

Lemma 5.2.5 *No correct server ever halts.*

Proof: Follows from Figure 5.4 and B8. □

Lemma 5.2.6 *If s_j initiates a broadcast b and s_k, $k > j$ initiates broadcast b', then no server dequeues b after b'.*

Proof: If $Faulty_k[s_j] = true$ before time $t + d$, then the lemma follows trivially from B11. If $Faulty_k[s_j] = false$ through time $t + d$, then from Figure 5.4, s_k cannot become the primary before time $t + d$, and hence does not broadcast b' before $t + d$. The lemma follows from B0 and B2. □

Lemma 5.2.7 *If s_j initiates a successful broadcast b, then for each non-halted server s_k, $j < k$: s_k dequeues b or s_k never becomes the primary.*

Proof: Suppose, for contradiction, that b was initiated at time t', s_k does not dequeue b but becomes the primary at time $\uparrow k$. From Figure 5.4, for all $i < k, Faulty_k[s_i] = true$ at time $\uparrow k$. Therefore, from B4, B5 and Figure 5.4, $\uparrow k = l\tau + d$ for some $l \geq 0$. Furthermore, from B9, $\uparrow k \geq \lceil \frac{t'}{\tau} \rceil \tau + d$. Therefore, there exists a $t \geq t'$ such that $l = \lceil \frac{t}{\tau} \rceil$. However, this violates B10 for the time t because $Faulty_k[s_j] = true$ at time $\uparrow k = \lceil \frac{t}{\tau} \rceil \tau + d$ and b has not been dequeued by time $\uparrow k = \lceil \frac{t}{\tau} \rceil \tau + d$. □

Lemma 5.2.8 *If a response r is sent to the client, then there was a successful broadcast of $((\texttt{log}, s_j, r), j)$ for some s_j.*

Proof: Follows from Figure 5.4. □

Theorem 18 *The protocol in Figures 5.4 satisfies* Pb1.

Proof: Define the predicate $Prmy_{s_j}$ at time t to be: (s_j has not halted by time t) $\wedge$ ($\forall k < j : Faulty_j[s_k] = true$ at time t). Note that $Prmy_{s_j}$ can be locally determined by s_j. Consider $Prmy_{s_i}$ and $Prmy_{s_j}$, where $i < j$. Now, $Prmy_{s_j}$ $\Rightarrow$ ($Faulty_j[s_i] = true$) and from B6, s_i has halted. Thus $Prmy_{s_j} \Rightarrow \neg Prmy_{s_i}$. □

Theorem 19 *The protocol in Figures 5.4 satisfies* Pb2.

Proof: Follows from Figure 5.2. □

Theorem 20 *The protocol in Figures 5.4 satisfies* Pb3.

Proof: A client only sends to a server that has executed **inform-clients** and a server s_j executes **inform–clients** only after it becomes the primary. □

Lemma 5.2.9 *Let s_j be the first server that does a successful broadcast after $\downarrow i$. Let s_{i_k} be the k^{th} server (ties broken arbitrarily) to halt between $\downarrow i$ and $\uparrow j$ such that $i < i_k < j$. Then, $\downarrow i_k \leq \downarrow i + k(\tau + d + C)$.*

Proof: We prove the result by contradiction. Let s_{i_v} be the first server that violates the lemma *i.e.* $\forall u : 0 < u < v : \downarrow i_u \leq \downarrow i + u(\tau + d + C)$ and $\downarrow i_v > \downarrow i + v(\tau + d + C)$. Let $s_{i'}$ be a non–halted server with the smallest id at time $\downarrow i_{v-1}$ such that $i < i' < j$ (there is such a server, in particular the server that halts at time $\downarrow i_v$). By B7 and Figure 5.4, $s_{i'}$ must become the primary by time $t = \downarrow i_{v-1} + \tau + d \leq \downarrow i + (v-1)(\tau + d + C) + \tau + d = \downarrow i + v(\tau + d + C) - C < \downarrow i_v - C$. Since $i' \neq j$, $s_{i'}$ does not initiate a successful broadcast. Hence, by B3 $s_{i'}$ must halt by time $t + C < \downarrow i_v$, which is a contradiction since the v^{th} server does not halt before time $\downarrow i_v$. □

Lemma 5.2.10 *Let s_j be the first primary that initiates a successful broadcast after $\downarrow i$. Then $\uparrow j \leq \downarrow i + (j - i)(\tau + d) + (j - i - 1)C$.*

Proof: From Figure 5.4, all servers $s_h, h < j$ must have halted by time $\uparrow j$. Let $S = \{s_u : i < u < j \wedge s_u$ halts between $\downarrow i$ and $\uparrow j\}$ and let $|S| = k$. Let s_v be the last server to halt in S (with ties broken arbitrarily). From Lemma 5.2.9, s_v must halt by time $\downarrow v \leq \downarrow i + (j - i - 1)(\tau + d + C)$. Therefore, by B7, s_j must become the primary by time $\downarrow i + (j - i - 1)(\tau + d + C) + \tau + d$ or by time $\downarrow i + (j - i)(\tau + d) + (j - i - 1)C$. □

Lemma 5.2.11 *Let m be a request that is sent to s_j such that m does not have a response. Then m must have been sent at time $t \geq max(\uparrow j, \downarrow j - C - \rho)$ and if received, was received in the interval $[max(\uparrow j, \downarrow j - C), \downarrow j]$.*

Proof: Clearly, m couldn't have been sent before $\uparrow j$ because s_j executes **inform–clients** only after becoming the primary. Suppose that m was sent at time t, where $\uparrow j \leq t < \downarrow j - C - \rho$. By Lemma 5.2.3, m must have been received by time $t + \rho$ and by Lemma 5.2.1 responded to by time $t + \rho + C < \downarrow j$, a contradiction, since m does not have a response.

Similarly, suppose that m was received at time t, where $\uparrow j \leq t < \downarrow j - C$. Again, by Lemma 5.2.1 a response to m would be sent by time $t + \rho + C < \downarrow j$, a contradiction. $\square$

Let $x(\sigma[t])$ denote the final value of variable x in the run $\sigma[t]$ where $\sigma[t]$ is the prefix of σ through time t. When σ is clear from context, we will abbreviate $x(\sigma[t])$ as $x(t)$.

Define the following over finite sequences:

$a \sqsubseteq b \equiv a$ is a prefix of $b \wedge |b| - |a| \leq 1$.

$a \sqsubset b \equiv a$ is a prefix of $b \wedge |b| - |a| = 1$.

$\texttt{prefix}(b) = a$ if $|b| > 1$ and $a \sqsubset b$. $\texttt{prefix}(b)$ is undefined otherwise. Therefore, $\texttt{prefix}(b)$ is b with the last value removed.

Define the following over some run σ, possibly infinite.

$\texttt{sf}(s_i) \equiv (s_i$ initiated a successful broadcast$)$.

$\texttt{prevsf}(s_i) = s_h, h < i$ if $\texttt{sf}(s_h)$ and there does not exist a i', $h < i' < i$ such that $\texttt{sf}(s_{i'})$. $\texttt{prevsf}(s_i)$ is undefined otherwise. Intuitively, $\texttt{prevsf}(s_i)$ is the last server before s_i that initiated a successful broadcast.

$\texttt{Pred}(s_i) = \{s_h : h < i$ and s_h becomes the primary$\}$.

$\texttt{Succ}(s_i) = \{s_j : j > i$ and s_j becomes the primary$\}$.

Lemma 5.2.12 *Let* $\texttt{sf}(s_j)$ *and* $s_i = \texttt{prevsf}(s_j)$. *Then no broadcast initiated by* $s_h \in \texttt{Pred}(s_i)$ *is dequeued after* $\uparrow j$.

Proof: By Property B9 and Figure 5.4, $\uparrow j \geq \lceil\frac{\uparrow i}{\tau}\rceil\tau + d$. The lemma now follows from Property B2. $\square$

Lemma 5.2.13 *Let* $\texttt{sf}(s_j)$*,* $s_i = \texttt{prevsf}(s_j)$ *and* $s_k \in \texttt{Succ}(s_j)$*. Then at most one broadcast* $b = ((\texttt{log}, s_i, \texttt{last}(Final_i)), i)$ *by* s_i *can be dequeued by* s_k *at time* $t > \uparrow j$*.*

Proof: Let s_i initiate its last successful broadcast at time t_i. From B9 and Figure 5.4, $\uparrow j \geq \lceil\frac{t_i}{\tau}\rceil\tau + d$ and so from B2, no successful broadcasts by s_i can be dequeued after time $\uparrow j$. The first broadcast that s_i initiates is successful, so by B0, the only broadcasts by s_i that can be dequeued after $\uparrow j$ are of the form $((\texttt{log}, s_i, response)), i)$. From Figure 5.4, s_i can make at most one unsuccessful broadcast of the form $((\texttt{log}, s_i, \texttt{last}(Final_i)), i)$. The lemma now follows from B0. $\square$

Let $Init_i = state_i(\uparrow i)$ and $Final_i = state_i(\downarrow i)$. In the following lemma, if s_i is undefined, then let $\uparrow i = \downarrow i = 0$ and $Init_i = Final_i = \epsilon$.

Lemma 5.2.14 *Let* $\texttt{sf}(s_j)$*,* $s_i = \texttt{prevsf}(s_j)$ *and* $s_k \in \texttt{Succ}(s_j) \cup \{s_j\}$*. Then*

1. $state_k(\uparrow j) \sqsubseteq Final_i$.
2. *If there exists a broadcast initiated by* s_i *that is dequeued by* s_k *at time* $t > \uparrow j$*, then* $state_k(\uparrow j) \sqsubset Final_i$*.*
3. *If there exists a broadcast* b *initiated by* $s_h \in \texttt{Pred}(s_j) \setminus \{s_i\}$ *that is dequeued by* s_k *at time* $t > \uparrow j$*, then* $b = (\texttt{mylastlog}, s_h, \texttt{last}(Final_i))$ *or* $b = (\texttt{mylastlog}, s_h, \texttt{last}(\texttt{prefix}(Final_i)))$*.*

4. If $Init_j \sqsubset Final_i$, then the response $\texttt{last}(Final_i)$ *was not sent to the client.*

5. If $Final_i = Init_i$, then $Init_j = Init_i$

Proof: Let $n = |\{s_{i'} : s_{i'} \in \texttt{Pred}(s_j) \wedge \texttt{sf}(s_{i'})\}|$. We prove the result by induction on n.

Base Case: $n = 0$. In this case, there is no server s_i and so $Init_i = Final_i = \epsilon$. At time 0, all states are ϵ. Since $n = 0$, all servers in $\texttt{Pred}(s_j)$ must have halted during their first broadcast and so all broadcasts initiated before $\uparrow j$ must be of the form $((\texttt{mylastlog}, -, \epsilon), -)$. From Figure 5.4, this satisfies (1), (2), (3), (4), (5).

Induction Case: $n > 0$ and we assume that the lemma is true when s_i became the primary. Let b' be the last broadcast by $s_h \in \texttt{Pred}(s_i)$ to be dequeued after $\uparrow i$. If $n > 1$, then let $Final_{i-1} = Final_{\texttt{prevsf}(s_i)}$, else let $Final_{i-1} = \epsilon$.

Observation 1: We now show that if $s_l \in \texttt{Succ}(s_i)$ dequeues b', then $state_l \sqsubseteq Final_{i-1}$ after s_l dequeues b' and updates its state. If $n = 1$, then b' is of the form $((\texttt{mylastlog}, s_h, \epsilon), h)$ and the argument is similar to the Base case. So assume that $n > 1$. If $h = i - 1$, then the result follows from Figure 5.4, Lemma 5.2.13 and the induction hypothesis for (2). On the other hand, if $h \neq i - 1$, then the result follows from Figure 5.4 and the induction hypothesis of (1) and (3).

Consider now the first (successful) broadcast $b = ((\texttt{mylastlog}, s_i, r), i)$ that s_i makes at time $\uparrow i$. By Lemma 5.2.7 and B2, any server $s_l \in \texttt{Succ}(s_i)$ must dequeue b by time $\lceil \frac{\uparrow i}{\tau} \rceil \tau + d$.

Observation 2: We next show that $state_l = Init_i$ after s_l dequeues b and updates its state. By Lemma 5.2.6, b cannot be dequeued before b'. However, from

Observation 1, $state_l \sqsubseteq Final_{i-1}$ after b' is dequeued. The result now follows from Figure 5.4 and the induction hypothesis for (1).

Let $Final_i = Init_i \circ r_1 \ldots r_N$ (the sequence $r_1 \ldots r_N$ could be empty). Let $M = ((\mathtt{log}, s_i, r_1), i) \ldots ((\mathtt{log}, s_i, r_N), i)$. Let the sequence $r = r_1 \ldots r_s$ be those requests that have been sent to clients, and let $R = ((\mathtt{log}, s_i, r_1), i) \ldots ((\mathtt{log}, s_i, r_s), i)$. From Figure 5.4, $R \sqsubseteq M$ and from Lemma 5.2.8, all broadcasts R must have been successful.

Assume that $s_{i+1} \in \mathtt{Succ}(s_i)$ is any server that becomes the primary after s_i and before s_j. (the argument is similar if no such server exists). By assumption, s_{i+1} does not initiate a successful broadcast. Also, if the last broadcast in R was initiated at time t_R by s_i, then by Property B9, $\uparrow (i+1) \geq \lceil \frac{t_R}{\tau} \rceil \tau + d$. Therefore, again by Lemma 5.2.7, B0 B1 and B2, $b \circ R$ must have been dequeued by s_{i+1}, s_j and $s_l \in \mathtt{Succ}(s_j)$ by time $\uparrow (i+1)$. From Observation 2, dequeueing b makes $state_{i'} = Init_i$ where $i' \in \{i+1, j, l\}$. Therefore, $state_{i'}(\uparrow (i+1)) = Init_i \circ r_1 \ldots r_{N-1}$ or $state_{i'}(\uparrow (i+1)) = Init_i \circ r_1 \ldots r_N$. (2) now follows, because from Lemma 5.2.6, if the last broadcast by s_i is dequeued after $\uparrow j$, then any broadcast by $\mathtt{Succ}(s_i)$ can only be dequeued after $\uparrow j$.

When s_{i+1} becomes the primary, it can initiate only one broadcast (by assumption) of $b' = (\mathtt{mylastlog}, s_{i+1}, \mathtt{last}(state_{i+1}(\uparrow (i+1))))$ which now along with Lemma 5.2.12 satisfies (3). If $s_{i'}$, $i' \in \{j, l\}$ ever delivers b' at some time $t \leq \uparrow j$, we still have $state_{i'}(t) = Init_i \circ r_1 \ldots r_{N-1}$ or $state_j(t) = Init_i \circ r_1 \ldots r_N$ (from Figure 5.4). This satisfies (1) and (5).

(4) is satisfied for the following reason. From the argument in the previous paragraph, $Init_j \sqsubset Final_i$ only if either s_{i+1} or s_j did not dequeue $\mathtt{last}(Final_i)$,

which implies that the broadcast of $((\texttt{log}, s_i, \texttt{last}(Final_i)), i)$ must have been unsuccessful. The result now follows from Lemma 5.2.8. $\square$

In the following lemma, we will construct a run σ_P of the bofo server P and a set of corresponding failure intervals. For brevity, we will refer to the i^{th} failure interval that we construct in σ_P as $ff(\sigma_P, i)$. Recall that a message sent by p_i to p_j gets enqueued (if it ever gets enqueued) in the message queue $queue_{i,j}$ at process p_j.

Lemma 5.2.15 *Let σ be any run of the primary backup system in which $\texttt{sf}(s_j)$ holds. Then*

1. *there is a finite run σ_P of a $(\rho, j, \tau + d + C + \rho + \delta)$ bofo server P such that for all clients $k : \sigma_P \approx_k \sigma[\uparrow j]$*
2. *$\uparrow j \in ff(\sigma_P, j)$ and $|ff(\sigma_P, j)| \leq \tau + d + C$*
3. *$state_P(\sigma_P[\uparrow j]) = Init_j$*
4. *For all clients $k : queue_{k,P}(\sigma_P[\uparrow j]) = \phi$.*
5. *Response r is sent to client k in σ_P iff r is sent to client k in $\sigma[\uparrow j]$. Furthermore, if r is sent at time t in σ_P, then r is sent at time t' in $\sigma[\uparrow j]$, where $t \leq t' \leq t + C$.*

Proof: Let $n = |\{s_{i'} : s_{i'} \in \texttt{Pred}(s_j) \wedge \texttt{sf}(s_{i'})\}|$. We prove the result by induction on n.

Base Case: $n = 0$. s_j is the first primary that initiates a successful broadcast at time $\uparrow j$. If $j = 0$, then $\uparrow j = 0$ and the lemma trivially holds.

If $j \neq 0$, then by B3, s_0 must have halted by time C and so $\downarrow 0 \leq C$. Since $j \geq 1$, from Lemma 5.2.10, $\uparrow j \leq \downarrow 0 + j(\tau+d) + (j-1)C \leq C + j(\tau+d) + (j-1)C = j(\tau + d + C)$. Divide the time from 0 to $\uparrow j$ into j failure intervals, each of size at most $\tau + d + C$. Construct the run σ_P until $\uparrow j$ such that the same requests are sent in σ_P and $\sigma[\uparrow j]$. All requests sent in the interval $[0..\uparrow j - \rho]$ are never enqueued in σ_P and all requests sent in $(\uparrow j - \rho..\uparrow j]$ haven't yet been enqueued. Thus (4) is satisfied because all queues are empty and (5) is satisfied because no responses are sent until time $\uparrow j$ in either σ or σ_P. All outages corresponding to these requests will lie in one of the above defined failure intervals. Thus (1) and (2) are satisfied. Also, from the above construction and Figure 5.1, $state_P(\uparrow j) = \epsilon$. Since by Lemma 5.2.14, $Init_j = \epsilon$, we have satisfied (3) as well.

Induction Case: $n > 0$. By the induction hypothesis, the lemma holds through time $\uparrow i$ where $s_i = \mathbf{prevsf}(s_j)$. Therefore, there exists a finite run σ'_P corresponding to $\sigma[\uparrow i]$. We now extend the run σ'_P to σ_P.

In order to simplify notation, we divide the proof into two cases.

Case 1: $\downarrow i - \uparrow i \leq \delta + \rho$.

1. The i^{th} failure interval is extended by $[\uparrow i..\downarrow i]$. From Lemma 5.2.10, $\uparrow j - \downarrow i \leq (j-i)(\tau+d) + (j-i-1)C \leq (j-i)(\tau+d+C)$. Divide the time from $\downarrow i$ to $\uparrow j$ into $j - i$ failure intervals, each of size at most $\tau + d + C$.

2. Construct the run σ_P until $\uparrow j$ such that the same requests are sent in σ_P and $\sigma[\uparrow j]$. All client requests sent in $[max(0, \uparrow i - \rho)..\uparrow j - \rho]$ to $s_{i'}, i' < i$ that are not enqueued in σ'_P are never enqueued in σ_P, and all requests sent in $(\uparrow j - \rho..\uparrow j]$ to $s_{i'}, i' < i$ that are not enqueued in σ'_P haven't yet been enqueued in σ_P. We can do since all outages will lie in one of the failure

intervals constructed in step 1.

Define M such that $Final_i = Init_i \circ M$. From Lemma 5.2.14 $Init_j = Init_i \circ M$ or $Init_j = Init_i \circ R$, where $R \sqsubset M$. Let $\mathcal{M}$ be the requests corresponding to the responses in M.

We next construct σ_P so that the same responses are received by the clients in both σ and σ_P in the interval $[\uparrow i..\uparrow j]$. We do so as follows. Let $m \in \mathcal{M}$, and let r be a response to m such that r is sent to the client in σ. Construct σ_P so that m is received at the same times in both σ and σ_P. We can do this because from Lemma 5.2.3, m is received in σ within ρ of being sent. We can further ensure that response r is sent in σ_P as well, because $state_P(\sigma'_P[\uparrow i]) = Init_i$, for all clients k : $queue_{k,P}(\sigma'_P[\uparrow i]) = \phi = queue_{k,s_i}(\uparrow i)$ (both from the induction hypothesis) and, as argued above, no requests sent to $s_{i'}, i' < i$ will be enqueued after $\uparrow i$ in σ_P. Thus both s_i and P receive the same sequence of requests, at the same times, and so go through the same state changes. In σ_P, the response r is sent at the same time m is received (note from Figure 5.1 that this has to be true of all runs of P) whereas in $\sigma[\uparrow i]$, r might be sent as much as C time units later by s_i. This satisfies (5). However, we construct σ_P so that r is received at the same time by the client in σ and σ_P. We can do this because r is received in σ by the client within $C + \delta$ after m is received, and $\rho \geq C + \delta$. Finally, if a response sent before $\uparrow i$ is received by a client after $\uparrow i$ in σ, then that response is received in σ_P as well. This can be done from the induction hypothesis for (5).

We now finish the construction of σ_P. First assume that $M \neq \epsilon$. If $Init_j = Init_i \circ R$, then assume that no further requests are enqueued in σ_P after the request corresponding to `last`(R) is enqueued. We can do this because from Lemma 5.2.14, `last`(M) wasn't sent to the client. The corresponding outages will lie in the failure intervals defined in step 1. Thus we have $state_P(\sigma_P[\uparrow j]) = state_P(\sigma'_P[\uparrow i]) \circ R = Init_j$ satisfying (3). (1) and (2) also follow from the construction. If $Init_j = Init_i \circ M$, then again assume that no request is enqueued after the request corresponding to `last`(M) (say m'). If m' does not have a response in σ, then construct σ_P so that P receives m', changes state but omits to respond to m'. We can do this because this outage will fall in the failure interval j.

If $M = \epsilon$, then (1), (2), (3) and (5) can similarly be shown by assuming that no requests are enqueued in σ_P in the interval $[\uparrow i..\uparrow j]$.

3. Finally, we show that the queues are empty. The queues are empty at time $\uparrow i$. By the above construction of σ_P, no requests sent to $s_{i'}, i' < i$ are enqueued after $\uparrow i$ in σ_P. Furthermore, the requests sent after time $\uparrow i$ to $s_{i'}, i \leq i' < j$, other than m' in step 2, that are received in $\sigma[\uparrow j]$ are received in σ_P as well. Also, by construction in step 2, m' is not enqueued at P at time $\uparrow j$. Finally, again by construction, all requests sent in $[\uparrow i..\uparrow j]$ to $s_{i'}, i \leq i' < j$ that are not received in $\sigma[\uparrow j]$ are never enqueued or haven't yet been enqueued in σ_P. Thus $\forall k : queue_{k,P}(\sigma_P[\uparrow j]) = \phi$.

Case 2: $\downarrow i - \uparrow i > \delta + \rho$.

1. Extend the i^{th} failure interval by $[\uparrow i..\uparrow i + \delta + \rho]$. Let $k = max(\uparrow i + \delta, \downarrow i - \rho - C)$. Now since $\uparrow j - (k + \rho) \leq \uparrow j - \downarrow i + C \leq (j - i)(\tau + d + C)$ (from Lemma 5.2.10), divide the interval $[k + \rho..\uparrow j]$ into $j - i$ failure intervals, each of size at most $\tau + d + C$.

2. From the implementation of **inform–clients**, no request can be sent to server $s_{i'}$, $i' < i$ after time $\uparrow i + \delta$. Therefore, construct σ_P so that all requests sent in $[max(0, \uparrow i - \rho)..\uparrow i + \delta]$ to $s_{i'}, i' < i$ that haven't yet been enqueued by $\uparrow i$ are never enqueued in σ_P. The outages will lie in the failure interval i.

3. Similar to Case 1, we can show that the same responses are sent in σ and σ_P. The only difference is the following. If in step 1, $k = \downarrow i - \rho - C$, then by Lemma 5.2.11, the outage corresponding to the request m' in Case 1, will lie in one of the above $j - i$ intervals. On the other hand, if $k = \uparrow i + \delta$, then the outage will lie either in interval i, or one of the $j - i$ intervals.

4. Similar to Case 1, we can show that the queues are empty.

□

Lemma 5.2.16 *For any run σ_{PB} of the primary–backup system there exists a run σ_P of the $(\rho, f, \tau + d + C + \rho + \delta)$ bofo server P such that for all clients $k : \sigma_P \approx_k \sigma_{PB}$.*

Proof Outline: Consider the run σ_{PB} of the primary–backup system. Let i be such that $\mathtt{sf}(s_i)$ and no server s_j, $j > i$ ever makes a successful broadcast in σ_{PB}. Since by Lemma 5.2.5, the number of servers that halt are bounded by f, such an i has to exist and $i \leq f$. Therefore, from Lemma 5.2.15, there is a run σ'_P of a $(\rho, f, \tau + d + C + \rho + \delta)$ bofo server such that for all clients $k : \sigma'_P \approx_k \sigma_{PB}[\uparrow i]$. It is then easy to extend σ'_P to σ_P, by an argument similar to one in Lemma 5.2.15. □

Theorem 21 *The protocol in Figures 5.4 satisfies* Pb4.

Proof: Follows directly from Lemma 5.2.16 □

Theorem 22 *The protocol in Figures 5.4 satisfies* Pb5.

Proof: From Lemma 5.2.5, a correct server never halts. The result now follows because a correct primary always remains the primary. □

We now prove a result about the failover time of the system.

Lemma 5.2.17 *For any time t, if at most F servers have halted by time t, then for all $t_1, t_2 : t_1 \leq t_2 \leq t$ such that $t_2 - t_1 \geq F(d + \tau)$, there is a primary in the interval $[t_1..t_2]$.*

Proof: We prove the result by induction on F.

Base Case: $F = 0$. Trivial, since from Figure 5.4, s_0 will always be the primary.

Induction Case: $F \geq 1$. Assume that the lemma holds as long as at most $F - 1$ servers halt. Consider any run σ. If less than F servers halt in σ, then the lemma follows from the induction hypothesis. Therefore, assume that at least F servers halt in σ. Let the F^{th} server halt at time t' and let this server be s_x. If $t' > t$, then the lemma follows from the induction hypothesis. So assume that $t' \leq t$. Let $t'_2 = t_1 + (F - 1)(d + \tau)$. Note that $t'_2 + d + \tau \leq t_2$.

If $t'_2 < t'$, then by the induction hypothesis, there is a primary in the interval $[t_1..t'_2]$. Since $[t_1..t'_2]$ is a prefix of $[t_1..t_2]$, there is a primary in the interval $[t_1..t_2]$ as well.

On the other hand, if $t'_2 \geq t'$, then let s_j be a non–halted server with the smallest id at time t'. There will exist such a server because by Lemma 5.2.5,

$n_s > f \geq F$. Since $t' \leq t'_2$ and $t'_2 + d + \tau \leq t_2$, we must have that $t' + d + \tau \leq t_2$. Also, since at most F failures occur until time t, s_j cannot halt through time $t_2 \leq t$, which implies that s_j cannot halt through time $t' + d + \tau$. Thus, from B7, $\forall i : Faulty_j[s_i] = true$ by time $t' + \tau + d$ and from Figure 5.4, s_j will be the primary by time $t' + d + \tau \leq t'_2 + d + \tau = t_1 + F(d + \tau) \leq t_2$. Hence there is a primary in the interval $[t_1..t_2]$. □

5.2.2 Message losses between the clients and the servers

We now consider the case when there can be message lost between the clients and the servers due to both failures of the links between clients and the servers, and due to omission failures of the servers. The **primary** procedure in Figure 5.4 will not tolerate such failures. For example, a non–faulty primary might omit to receive all requests from a client, thereby violating Pb4. Similarly, **inform–clients** might omit to inform some of the clients. However, it is relatively easy to mask such failures when clients are non-faulty.[1] Assume that there is an upper bound (say G) between any two requests from a client and that requests carry sequence numbers. If the primary does not receive any requests from a client during an interval of length G or if the primary receives some request with a sequence number gap, then the primary halts. Similarly, the primary can detect that a response was lost by having clients acknowledge responses. If such an acknowledgement is not received, then again the primary halts. Properties Pb1–Pb5 can again be shown to be true if we make the above modification in Figures 5.2 and 5.4.

[1]The protocols described in this paper can also be extended to tolerate crash failures of clients. We do not discuss these extensions in this paper.

5.3 Implementation for the various failure models

In this section, we show how to implement the procedures **initialize**, **broadcast**, **delivery–process** and **failure–detector** for the various failure models. For each set, we also show these procedures satisfy B0—B11.

5.3.1 Crash Failures

The five procedures for crash failures are given in Figure 5.5. The notation $A \triangleq B$ defines the symbol A to be the same as the symbol B. Whenever we say that a server "delivered m", we mean that the procedure **deliver** has been called with parameter M having the value m. The procedure **Enq** adds an element to the head of a queue and the procedure **Deq** dequeues an element from the tail of the queue. Furthermore, in properties B0—B11, $d = \delta$ and $C = 0$.

Lemma 5.3.1 *The procedures in Figure 5.5 satisfies* B0–B11.

Proof:

B0: From Figure 5.5, any message that is enqueued in **deliver** must have been broadcast earlier. Furthermore, each message can be enqueued (and hence dequeued) at most once because each message can be received at most once.

B1: From FIFO properties of the communications links and the **Enq** and **Deq** operations on $Rqueue_k$.

B2: From the fact that message delivery takes at most δ and none of the procedures are blocking.

procedure initialize(k)
  $state_k := Rqueue_k := \epsilon$
  $\forall i : Faulty_k[s_i] := false$

procedure broadcast(M, k)
  send M **to** all servers

procedure deliver (M, k)
  Let M be of the form $(tag, -, -)$
  if $tag \in \{\texttt{log}, \texttt{mylastlog}\}$ **then** **Enq**($Rqueue_k, (M, k)$)

procedure delivery–process(k)
  do forever
    when received M: **deliver**(M, k)
  od

procedure failure–detector(k)
  $A^l_j \triangleq (\texttt{alive}, s_j, l\tau)$
  cobegin
    || **for** $l := 0$ to ∞
        when $current\text{-}time = l\tau$: **send** A^k_l **to** all servers
    || **for** $l := 0$ to ∞
        when $current\text{-}time = l\tau + d$:
          $\forall j$: **if** not delivered A^l_j **then** $Faulty_k[s_j] := true$
  coend

Figure 5.5: Procedures for crash failures

B3: From the implementation of **broadcast.**

B4: Follows from Figures 5.4 and 5.5, since they are set to false only in procedure **initialize.**

B5: Follows from Figures 5.4 and 5.5 and the second **when** statement in the **failure–detector.**

B6: Suppose s_k sets $Faulty_k[s_j] = true$ at $l\tau + d \leq t$. This implies that s_j did not send $(\texttt{alive}, s_j, l\tau)$ to s_k and therefore s_j must have halted by time $l\tau \leq t$.

B7: Let $(l-1)\tau \leq t < l\tau$. Therefore there was no $(\texttt{alive}, s_k, l\tau)$ message sent and so from Figure 5.5, s_j will set $Faulty_j[s_k] = true$ by time $l\tau + d = (l-1)\tau + \tau + d \leq t + \tau + d$.

B8: Follows from Figure 5.5 because the code does not contain any **stop** statement.

B9: Let $l\tau < t \leq (l+1)\tau$ for some $l \geq -1$, which implies that $\lceil \frac{t}{\tau} \rceil = l + 1$. By assumption, s_j initiates a successful broadcast at time t and so must have sent $(\texttt{alive}, s_j, l'\tau)$ for all $l' \leq l$ which in turn were delivered at all non–halted servers by $l'\tau + \delta$. Thus, from Figure 5.5 no server can set $Faulty_k[s_j] = true$ before time $(l+1)\tau + \delta = \lceil \frac{t}{\tau} \rceil \tau + \delta$.

B10: If b was initiated at time $t' \leq t$ then b must have been dequeued by all non–halted servers at time $t' + \delta \leq t + \delta \leq \lceil \frac{t}{\tau} \rceil \tau + \delta$.

B11: Let $l\tau < t \leq (l+1)\tau$ for some $l \geq -1$, which implies that $\lceil \frac{t}{\tau} \rceil = l + 1$. By assumption, s_j initiates a broadcast at time t and so must have sent $(\texttt{alive}, s_j, l'\tau)$ for all $l' \leq l$ which in turn were delivered at s_k by $l'\tau + \delta$. Thus, from Figure 5.5 s_k cannot set $Faulty_k[s_j] = true$ before time $(l+1)\tau + \delta \geq t + \delta$.

□

Theorem 23 *The lower bounds for crash failures shown in Table 4.1 are all tight.*

Proof: Properties B0—B11 are met for $n_s > f$ which is optimal. The procedure **broadcast** does not block and so $C = 0$ which is optimal. Finally, since Lemma 5.2.17 holds for all $\tau > 0$, the failover time is also optimal because $d = \delta$. □

5.3.2 Crash+link failures

The procedures in Section 5.3.1 do not satisfy all of B0—B11 when links are not assumed to be failure–free. For example, if s_j sends a message to s_k then the message might not reach s_k due to a link failure, leading to a violation of B6 or B10. We therefore replace the implementation of the procedures **initialize**, **broadcast**, **delivery–process** and **failure–detector** in Figure 5.5 with the implementation in Figure 5.6. We define $d = 2\delta$, $C = 0$ and assume that $n_s > f+1$. The procedures in Figure 5.5 use the routines **fifo–broadcast** and **fifo–deliver** in Figure 5.7. In the rest of the paper, whenever we say that a server "fifo–broadcasts m" ("fifo–delivers m"), then we mean that the procedure **fifo–broadcast** (**fifo–deliver**) was called with the parameter m. The procedures **fifo–broadcast** and **fifo–deliver** convert an intermittent link failures into a permanent link failure: if s_j fifo–broadcasts a message m to s_k, then for every subsequent message m' that s_j fifo–broadcasts to s_k, s_k will not fifo–deliver m' unless it first fifo–delivers m.

Lemma 5.3.2 *If s_j fifo-broadcasts m and then fifo-broadcasts m', then s_k cannot fifo-deliver m' before it fifo-delivers m.*

Proof: Follows from Figure 5.7. □

Lemma 5.3.3 *If s_j fifo-broadcasts m at time t and does not halt during the fifo-broadcast, and if s_k does not fifo-deliver m, then either s_k has halted by time $t+\delta$ or the link from s_j to s_k is faulty.*

Proof: We assume that s_k has not halted by time $t+\delta$ and show that the link from s_j to s_k must be faulty. Let m' be a message that s_j fifo–broadcasts at time

procedure initialize(k)
  $state_k$:= $Rqueue_k$:= $Dqueue_k$:= ϵ
  $\forall i : Faulty_k[s_i] := false$
  $last\text{-}sent_k := \forall j : expected_k[j] := 0$

procedure broadcast(M, k)
  $time :=current\text{--}time$
  fifo-broadcast(`init`, $M, s_k, time$)

procedure delivery-process(k)
  cobegin
    || **fifo-delivery-process**(k)
    || **do forever**
        $(tag, M, -, t)$:=**Deq**($Dqueue_k$)
        if tag = `init` **then fifo-broadcast** (`echo`, M, s_k, t)
        if tag = `echo` **and** not delivered M before **then deliver** (M, k)
      od
  coend

procedure failure–detector(k)
  $A^l_j \triangleq$ (`alive`, $s_j, l\tau$)
  cobegin
    || **for** $l := 0$ to ∞
        when $current\text{–}time = l\tau$: **fifo-broadcast**(`init`, $A^l_k, s_k, l\tau$)
    || **for** $i := 0$ to ∞
        when $current\text{–}time = l\tau + d$:
          $\forall j$: **if** not delivered A^l_j **then** $Faulty_k[s_j] := true$
  coend

Figure 5.6: Procedures for crash+link failures-i

$t' \leq t$ (m' could be m). Furthermore, all messages that were fifo–broadcast before m' (call this set $\mathcal{M}$) by s_j have been fifo–delivered by s_k by time $t' + \delta$, whereas m' is not fifo–delivered until time $t' + \delta$. If it is the case that m' is not received by s_k by time $t' + \delta$, then the link must be faulty, and we are done. On the other hand, if m' is received, then m' must be received after $\mathcal{M}$ because the links are FIFO. This, however, implies from the implementation of **fifo–delivery–process** that m' should have been fifo–delivered, a contradiction. □

---

procedure fifo-broadcast(tag, M, s_k, t)
  send $(tag, M, s_k, t, \textit{last-sent}_k)$ **to** all
  $\textit{last-sent}_k := \textit{last-sent}_k + 1$

procedure fifo-deliver (tag, M, s_j, t)
  Enq$(Dqueue_k, (tag, M, s_j, t))$

procedure fifo-delivery–process (k)
  do forever
    when received $(tag, M, s_j, t, last_j)$:
      if $(last_j \neq expected_k[j])$ **then skip**
      else
        $expected_k[j] := expected_k[j] + 1$
        fifo-deliver (tag, M, s_j, t)
  od

Figure 5.7: Procedures for crash+link failures-ii

Lemma 5.3.4 *If s_j fifo-broadcasts* $(\texttt{init}, M, s_j, t)$ *and does not halt during the fifo-broadcast, then all servers that have not halted by time $t + 2\delta$ fifo-deliver* $(\texttt{echo}, M, -, t)$ *by that time.*

Proof: Since s_j does not halt during the fifo–broadcast, s_j must have sent $(\texttt{init}, M, s_k, t)$ to all servers, and all servers s_i that fifo-delivered $(\texttt{init}, M, s_k, t)$

either sent $(\texttt{echo}, M, s_i, t)$ by time $t+\delta$ to all servers or halted. We show that any server s_k that does not halt by time $t+2\delta$ fifo-delivers at least one copy of $(\texttt{echo}, M, -, t)$ by that time. Consider the following $n_s - 1 \leq f+1$ communication paths: $< s_j, s_0, s_k >, \ldots, < s_j, s_{j-1}, s_k >, < s_j, s_{j+1}, s_k > \ldots, < s_j, s_{n_s-1}, s_k >$. Since there can be at most f failures, at least one communication path (say $< s_j, s_i, s_k >$) must remain intact *i.e.* s_i has not halted by time $t+2\delta$ and the link from s_j to s_i and the link from s_i to s_k are both non–faulty. The lemma now follows by applying Lemma 5.3.3 twice. □

Lemma 5.3.5 *The procedures in Figures 5.5 and 5.6 satisfy* B0–B11.

Proof:

B0: Any message that is dequeued must have been broadcast earlier. The fact that a message is dequeued at most once can be easily seen from the implementation of **delivery–process**.

B1: Suppose that s_j initiates the broadcast b' after it initiates the broadcast b. By Figure 5.7, s_j cannot fifo–broadcast $(\texttt{init}, b', s_j, -)$ before it fifo–broadcasts $(\texttt{init}, b, s_j, -)$. By Lemma 5.3.2, all servers s_i must fifo–deliver $(\texttt{init}, b, s_j, -)$ before it fifo–delivers $(\texttt{init}, b', s_j, -)$. Hence, by Figure 5.6, s_i does not fifo–broadcast $(\texttt{echo}, b', s_i, -)$ before it fifo–broadcasts $(\texttt{echo}, b, s_i, -)$. Therefore, by Lemma 5.3.2 all servers s_k must fifo–deliver $(\texttt{echo}, b, s_i, -)$ before it fifo–delivers $(\texttt{echo}, b', s_i, -)$. The lemma now follows from the FIFO properties of **Enq** and **Deq**.

B2: Suppose that a server s_k dequeues b. Therefore, some server s_i must have fifo–broadcasted $(\texttt{echo}, b, s_i, -)$ by time $t+\delta$ and s_k must have fifo–delivered this by time $t + 2\delta$ and therefore from Figure 5.6 delivered b by that time. The result now follows because s_k dequeues b as soon as it is delivered.

B3: From the implementation of **broadcast**.

B4: Follows from Figures 5.4 and 5.6, since they are set to false only in procedure **initialize**.

B5: Follows from Figures 5.4 and 5.6 and the second **when** statement in the **failure–detector**.

B6: Suppose $Faulty_j[s_k]$ is true for the first time at $t = l\tau + 2\delta$. This, therefore implies from Figure 5.6 that s_j did not deliver A_k^l and therefore did not fifo–deliver $(\texttt{echo}, A_k^l, -, l\tau)$ by time $l\tau + 2\delta$. By Lemma 5.3.4, s_k must have halted by time $l\tau$.

B7: Let $(l-1)\tau \leq t < l\tau$. Therefore there was no fifo–broadcast of $(\texttt{init}, A_k^l, s_k, l\tau)$ and so from Figure 5.6, s_j will set $Faulty_j[s_k] = true$ by time $l\tau + 2\delta = (l-1)\tau + \tau + 2\delta \leq t + \tau + 2\delta$.

B8: Follows from Figure 5.6 because the code does not contain any **stop** statement.

B9: Let $l\tau < t \leq (l+1)\tau$ for some $l \geq -1$, which implies that $\lceil \frac{t}{\tau} \rceil = l + 1$. s_j must have fifo-broadcasted $(\texttt{init}, A_j^{l'}, s_k, l'\tau)$ for all $l' \leq l'$. From Lemma 5.3.4, s_k will fifo-deliver $(\texttt{echo}, A_j^{l'}, s_k, l'\tau)$ by $l'\tau + 2\delta$ and so deliver $A_j^{l'}$. Therefore, s_k cannot set $Faulty_k[s_j]$ to $true$ before time $(l+1)\tau + 2\delta = \lceil \frac{t}{\tau} \rceil \tau + 2\delta$.

B10: From B9, $Faulty_k[s_j]$ cannot become true before time $t + 2\delta$. If on the other hand, $Faulty_k[s_j]$ does become true after this time, then using Lemma 5.3.4 and B2, s_k must have dequeued b by this time.

B11: Let $l\tau < t \leq (l+1)\tau$ for some $l \geq -1$, which implies that $\lceil \frac{t}{\tau} \rceil = l+1$. s_j must have fifo-broadcasted $(\texttt{init}, A_j^{l'}, s_k, l'\tau)$ for all $l' \leq l'$. From Lemma 5.3.4, s_k will fifo-deliver $(\texttt{echo}, A_j^{l'}, s_k, l'\tau)$ by $l'\tau + 2\delta$ and so deliver $A_j^{l'}$. Therefore, s_k cannot set $Faulty_k[s_j]$ to *true* before time $(l+1)\tau + 2\delta = \lceil \frac{t}{\tau} \rceil \tau + 2\delta$.

□

Theorem 24 *The lower bounds for crash+link failures shown in Table 4.1 are all tight.*

Proof: Properties B0—B11 are met for $n_s > f$ which is optimal. The procedure **broadcast** does not block and so $C = 0$ which is optimal. Finally, since Lemma 5.2.17 holds for all $\tau > 0$, the failover time is also optimal because $d = 2\delta$. □

5.3.3 Send–Omission failures

The procedures of Figure 5.5 do not tolerate send–omission failures. Procedures for send–omission failures are given in Figures 5.8 and 5.9, used in conjunction with **delivery–process** of Figure 5.6. These procedures use a technique similar to the translation technique described in [NT88]. However, we do not use the round model of computation, and our technique is adapted to work with a separate failure detector. Informally, this technique ensures that if a server omits to send a message, then it detects its own failure and halts. This is achieved by making each server s_i broadcast every τ a message saying that it is alive. If some server s_j does not receive this message within δ, then s_i must have committed a send–omission failure. Then s_j broadcasts this fact, and when s_i learns of this, it crashes itself. Thus, if a correct server observes the send–omission failure, then s_i will stop, effectively translating the send–omission failure into a crash failure.

For these procedures, we will define $d = 2\delta$ and $C = \tau + 2\delta$ and assume that $n_s > f$.

---

procedure initialize(k)
  $state_k := Rqueue_k := Dqueue_k := \epsilon$
  $current\text{-}primary{:=}0$
  $\forall i : Faulty_k[s_i] := false$
  $last\text{-}sent_k := \forall j : expected_j := 0$

procedure broadcast(M, k)
  $time := current\text{-}time$
  if $\lceil \frac{time}{\tau} \rceil \tau = time$ **then** $t := time + \tau$
  else $t := time$
  fifo-broadcast(init, $M, s_k, time$)
  wait until $current\text{-}time = \lceil \frac{t}{\tau} \rceil \tau + d$

procedure deliver (M, k)
  Let $M = (tag, s_j, -)$
  if $tag \in$ {log, mylastlog} **then**
    if $j <current\text{-}primary$ **then return**
    else
      $current\text{-}primary{:=}\, j$
      Enq($Rqueue_k, M$)

Figure 5.8: Procedures for send–omission failures-i

As can be seen, the procedure **broadcast** always blocks at least for time 2δ, even

procedure failure-detector(k)

$A^l_j \triangleq$ (alive, $s_j, l\tau$)

$F^l_j \triangleq$ (fault, $s_j, l\tau$)

cobegin

|| **for** $l := 0$ to ∞

when *current-time* $= l\tau$:

fifo-broadcast(init, $A^l_k, s_k, l\tau$)

|| **for** $l := 0$ to ∞

when *current-time* $= l\tau + \delta$:

$\forall j$: **if** not fifo–delivered (init, $A^l_j, s_j, l\tau$) **then**

fifo-broadcast (echo, $F^l_j, s_k, l\tau$)

|| **for** $l := 0$ to ∞

when *current-time* $= l\tau + 2\delta$:

if fifo–delivered (echo, $F^l_k, -, l\tau$) **then stop**

if $\exists j$: not delivered A^l_j **then** $Faulty_k[s_j] := true$

coend

Figure 5.9: Procedures for send–omission failures-ii

if there are no failures. However, this can be optimized so that the blocking occurs only for a round trip message delay (which can be less than 2δ) in the absence of failures. We do not show this optimization in this paper.

Lemmas 5.3.2 continues to hold for send–omission failures.

Lemma 5.3.6 *If a correct server* **fifo-broadcasts** m *at time* t*, then all servers that haven't halted by time* $t + \delta$ **fifo-deliver** m.

Proof: In this failure model, there are no receive–omission failures. The lemma now follows from Figure 5.7 because no non–halted server can omit to receive messages. □

Lemma 5.3.7 *The procedures in Figures 5.8 and 5.9 satisfy properties* B0—B11.

Proof:

B0: Same as crash+link failures.

B1: Suppose that s_j initiates the broadcast b' after it initiates the broadcast b. By Figure 5.7, s_j cannot fifo–broadcast $(\texttt{init}, b', s_j, -)$ before it fifo–broadcasts $(\texttt{init}, b, s_j, -)$. By Lemma 5.3.2, a server s_i must fifo–deliver $(\texttt{init}, b, s_j, -)$ before it fifo–delivers $(\texttt{init}, b', s_j, -)$. Hence, by Figure 5.6, s_i does not fifo-broadcast $(\texttt{echo}, b', s_i, -)$ before it fifo–broadcasts $(\texttt{echo}, b, s_i, -)$. Therefore, by Lemma 5.3.2, a server s_k must fifo–deliver $(\texttt{echo}, b, s_i, -)$ before it fifo-delivers $(\texttt{echo}, b', s_i, -)$. Now, if s_k enqueues b' in the queue $Rqueue_k$, it must have also enqueued b earlier because *current–primary* never decreases. The lemma now follows from the FIFO properties of **Enq** and **Deq**.

B2: Same as the proof for crash+link failures.

B3: Follows from the implementation of **broadcast** in Figure 5.8.

B4: Follows from Figures 5.4 and 5.9 because it can set to false only in **initialize**.

B5: Follows from Figures 5.4 and 5.9 from the third **when** statement in the **failure–detector**.

B6: By Lemma 5.3.6 all servers will deliver $(\texttt{alive}, s_j, l\tau)$ for all l if s_j is correct and so can never set $Faulty_k[s_j] = true$. So assume that s_j is faulty and a server s_k does not deliver $(\texttt{alive}, s_j, l\tau)$, where $l\tau + 2\delta \leq t$. This now implies (again from Lemma 5.3.6) that no correct server (say q) would have fifo-delivered $(\texttt{init}, (\texttt{alive}, s_j, l\tau), s_j, l\tau)$ and therefore, would have fifo-broadcasted $(\texttt{echo}, (\texttt{fault}, s_j, l\tau), q, l\tau)$. Server s_j would have received this message and therefore have halted by time $l\tau + 2\delta \leq t$.

B7: Same as the proof for crash+link failures.

B8: Let us assume, for contradiction, that s_k is the first correct server to halt at time $t = l\tau + 2\delta$ by executing a **stop** in Figure 5.9. s_k must have fifo-delivered $(\texttt{echo}, F_k^l, s_j, l\tau)$ from some server s_j at time $l\tau + 2\delta$. However, since s_k hadn't halted by time $l\tau$, if would have fifo-broadcasted $(\texttt{init}, A_k^l, s_k, l\tau)$ and from Lemma 5.3.6, s_j would have fifo–delivered this by time $l\tau + \delta$ and so couldn't have fifo-broadcasted $(\texttt{echo}, F_k^l, s_j, l\tau)$, a contradiction.

B9: Assume that the property is false for a successful broadcast b and s_k sets $Faulty_j[s_i] = true$ first at time $l\tau + 2\delta$. This implies that s_k did not deliver $(\texttt{alive}, s_j, l\tau)$, which implies that neither s_k nor any correct server would have fifo-delivered $(\texttt{init}, (\texttt{alive}, s_j, l\tau), s_j, l\tau)$ (by Lemma 5.3.6). Now by assumption, s_j must have fifo-broadcasted $(\texttt{init}, b, s_j, t)$ after fifo-broadcasting $(\texttt{init}, (\texttt{alive}, s_j, l\tau$

$), s_j, l\tau)$ and (by Lemma 5.3.2) neither s_k nor any correct server can fifo-deliver $(\texttt{init}, b, s_j, t)$. If $\lceil \frac{t}{\tau} \rceil \tau = t$ then let $t' = t + \tau$, else let $t' = t$. Since no correct server (say q) fifo-delivered $(\texttt{init}, b, s_j, t)$, q will not fifo-deliver $(\texttt{init}, (\texttt{alive}, s_j, \lceil \frac{t'}{\tau} \rceil \tau), s_j, \lceil \frac{t'}{\tau} \rceil \tau)$ (by Lemma 5.3.2). This, however, implies that s_j would have halted by time $\lceil \frac{t'}{\tau} \rceil \tau + 2\delta$ (Figure 5.9), a contradiction since b is a successful broadcast.

B10: Let b be initiated at time $t_b \leq t$. From Figure 5.4 and B9, no broadcast by s_l, $l > j$ can be initiated before time $t_b + 2\delta$ and so can only be delivered after $t_b + 2\delta$. Therefore, from the implementation of **deliver** in Figure 5.8, B10 holds if we can show that s_k fifo-delivers $(\texttt{echo}, b, -, t_b)$ by time $t_b + 2\delta$.

If s_j is correct, then this is trivially true (by Lemma 5.3.6) because all servers will fifo-deliver $(\mathtt{init}, b, s_j, t_b)$ by time $t_b + \delta$. So assume that s_j is faulty. If $\lceil \frac{t_b}{\tau} \rceil \tau = t_b$ then let $t' = t_b + \tau$, else let $t' = t_b$. Thus, s_j waited until time $\lceil \frac{t'}{\tau} \rceil \tau + 2\delta$ and did not halt at this time. This, however, implies that a correct server (say q), would have fifo-received $(\mathtt{init}, (\mathtt{alive}, s_j, \lceil \frac{t'}{\tau} \rceil \tau), s_j, \lceil \frac{t'}{\tau} \rceil \tau)$, which then implies that q would have fifo-delivered $(\mathtt{init}, b, s_j, t_b)$ and fifo-broadcasted $(\mathtt{echo}, b, q, t_b)$ by time $t_b + \delta$. Therefore, from Lemma 5.3.6, all non-halted servers would have fifo-delivered $(\mathtt{echo}, b, q, t_b)$ by time $t_b + 2\delta$.

B11: Follows trivially from the implementation of **deliver** in Figure 5.8.

□

Theorem 25 *The lower bounds for send–omission failures shown in Table 4.1 are all tight.*

Proof: Properties B0—B11 are met for $n_s > f$ which is optimal. From the procedure **broadcast**, $C = \tau + 2\delta$ for any $\tau > 0$, which is optimal. Finally, since Lemma 5.2.17 holds for all $\tau > 0$, the failover time is also optimal because $d = 2\delta$.

□

5.3.4 Receive–Omission failures

The protocol for tolerating receive–omission failures is obtained by taking the protocol for send–omission failures and replacing **broadcast** and **failure–detector** with the procedures in Figure 5.10. As with send–omission failures, these procedures were obtained using a technique similar to that described in [NT88]. Informally, a server s_i broadcasts every τ a message saying that it is alive. Any server that receives this message forwards it to all other servers. A server s_k keeps track of the number of forwarded "alive" messages in an array $witness_k[i]$. If s_k does not receive enough forwarded copies of the "alive" message within 2δ, then s_k must have committed at least one receive–omission failure. Server s_k detects this fact and crashes itself, effectively translating the receive–omission failure into a crash failure.

For these procedures, we define $d = 2\delta$, $C = 0$ and assume that $n_s > 2f$.

---

procedure broadcast(M, k)
  $time := current\text{–}time$
  fifo-broadcast$(\texttt{init}, M, s_k, time)$

procedure failure-detector(k)
  $A_j^l \triangleq (\texttt{alive}, s_j, l\tau)$
  $F_j^l \triangleq (\texttt{fault}, s_j, l\tau)$
  cobegin
    || **for** $l := 0$ to ∞
        when $current\text{–}time = l\tau$: **fifo-broadcast**$(\texttt{init}, A_k^l, s_k, l\tau)$
    || **for** $l := 0$ to ∞
        when $current\text{–}time = l\tau + \delta$:
          $\forall j$: **if** not fifo–delivered $(\texttt{init}, A_j^l, s_j, l\tau)$ **then**
            fifo-broadcast $(\texttt{echo}, F_j^l, s_k, l\tau)$
    || **for** $l := 0$ to ∞
        when $current\text{–}time = l\tau + d$:
          $witness_k[k] := \{s_j | \text{fifo–delivered } (\texttt{echo}, A_k^l, s_j, l\tau)\}$
          $\forall j \neq k : witness_k[j] := \{s_i | \text{fifo–delivered } (\texttt{echo}, A_j^l, s_i, l\tau)$ **or**
                fifo–delivered $(\texttt{echo}, F_j^l, s_i, l\tau)\}$
          if $\exists j : |witness_k[j]| < n_s - f$ **then stop**
          if $\exists j$: not delivered A_j^l **then** $Faulty_k[s_j] := true$
coend

Figure 5.10: Procedures for receive–omission failures

Lemma 5.3.6 no longer holds because a faulty server might omit to receive by receive–omission. We, therefore, strengthen it as follows.

Lemma 5.3.8 *If a server* **fifo-broadcasts** m *at time* t *and does not halt, then all correct servers* **fifo-deliver** m *by time* $t + \delta$.

Proof Outline: From Figure 5.7 and the facts that there are only receive–omission failures and the channels are FIFO, a correct server can skip messages only if it halts. However, B8 proved below shows that a correct server can never halt. □

Property B8 will be proved below by using only the following weak form of Lemma 5.3.8.

Lemma 5.3.9 *If a correct server* **fifo-broadcasts** m *at time* t *and does not halt, then all correct servers that have not halted by time* $t + \delta$ **fifo-deliver** m *by time* $t + \delta$.

Proof Outline: From Figure 5.7 and the fact that the channels are FIFO. □

Lemma 5.3.10 *The procedures in Figure 5.10 satisfy properties* B0—B11.

Proof:

B0: Same as send–omission failures.

B1: Same as the proof for send–omission failures.

B2: Same as the proof for crash+link failures.

B3: From the implementation of **broadcast** since the maximum time blocking can occur is $\tau + 2\delta$.

B4: Follows from Figures 5.4 and 5.10 because it can become false only in **initialize**.

B5: Follows from Figures 5.4 and 5.10 and the third **when** statement in the **failure–detector**.

B6: Assume that s_k has not halted by time t. We prove that $Faulty_j[s_k]$ is false at time t. Thus, for all $l : l\tau \leq t$, s_k has fifo-broadcast $(\texttt{init}, (\texttt{alive}, s_k, l\tau), -, -)$ at time $l\tau$. From Lemma 5.3.8, all correct servers fifo-deliver $(\texttt{init}, (\texttt{alive}, s_k, l\tau), -, -)$ by time $l\tau + \delta$. Thus no more than f servers fifo-broadcast $(\texttt{echo}, (\texttt{fault}, s_k, l\tau), -, -)$. Since $2f < n_s$, $f < n_s - f$ and so no server s_j fifo-delivers sufficient $(\texttt{echo}, (\texttt{fault}, s_k, l\tau), -, -)$ messages to set $Faulty_j[s_k] = true$ before time $t + 2\delta$.

B7: Same as the proof for crash+link failures.

B8: Let us assume, for contradiction, that s_k is the first correct server to halt at time $t = l\tau + 2\delta$ by executing a **stop** in Figure 5.10. Therefore, $witness_k[j] < n_s - f$ for some j.

Case $j = k$: Since s_k did not halt at time $l\tau$, it must have fifo-broadcasted $(\texttt{init}, A_k^l, -, l\tau)$ at time $l\tau$. Since no correct server has halted by time $l\tau + \delta$, by Lemma 5.3.9, all correct servers will fifo-deliver this message by time $l\tau + \delta$. Since there are $n_s - f$ correct servers, all these servers will fifo-broadcast $(\texttt{echo}, A_k^l, -, l\tau)$. Server s_k can execute a **stop** statement at time $l\tau + 2\delta$

only if it did not fifo-deliver $n_s - f$ copies of $(\texttt{echo}, A_k^l, -, l\tau)$. However, since $n_s - f$ copies of $(\texttt{echo}, A_k^l, -, l\tau)$ were sent and s_k is correct, from Figure 5.7, s_k can omit to fifo-deliver any one of these copies (say from server s_i) only if it had omitted to fifo-deliver an earlier message from s_i. However, this cannot happen since s_k is correct and the channels are FIFO. Thus s_k cannot execute the **stop**, a contradiction.

Case $j \neq k$. Similar to the previous case because each correct server would have fifo-broadcasted $(\texttt{echo}, A_j^l, -, l\tau)$ or $(\texttt{echo}, F_j^l, -, l\tau)$ by time $l\tau + \delta$, and s_k would have fifo-delivered this. Since there are at least $n_s - f$ correct servers, s_k cannot halt.

B9: Let $l\tau < t \leq (l+1)\tau$. We show that $Faulty_k[s_j] = false$ at time $l\tau + 2\delta$ and therefore can become true only at time $(l+1)\tau + 2\delta$.

Since $t > l\tau$, for all $l' : l' \leq l$, s_j has fifo-broadcast $(\texttt{init}, (\texttt{alive}, s_j, l'\tau), -, -)$ at time $l'\tau$. From Lemma 5.3.8, all correct servers fifo-deliver $(\texttt{init}, (\texttt{alive}, s_j, l'\tau), -, -)$ by time $l'\tau + \delta$. Thus no more than f servers fifo-broadcast $(\texttt{echo}, (\texttt{fault}, s_j, l\tau), -, -)$. Since $2f < n_s$, $f < n_s - f$ and so s_k cannot fifo-deliver sufficient $(\texttt{echo}, (\texttt{fault}, s_j, l'\tau), -, -)$ messages to set $Faulty_j[s_k] = true$ before time $(l+1)\tau + 2\delta$.

B10: Let b be initiated at time $t_b \leq t$ and let $l\tau \leq t_b < (l+1)\tau$. Assume first that $t_b > l\tau$. We show that s_k cannot omit to dequeue b. Again, as for send–omission failures, we need to prove that s_k would have fifo-delivered $(\texttt{echo}, b, -, t_b)$ by time $t_b + 2\delta$.

All correct servers must have fifo-delivered $(\texttt{init}, b, s_j, t_b)$ (by Lemma 5.3.8) and fifo-broadcasted $(\mathbf{echo}, b, -, t_b)$ before time $(l+1)\tau+\delta$. Since s_k did not halt by time $(l+1)\tau+2\delta$, it must have fifo-delivered $(\mathbf{echo}, (\mathtt{alive}, s_j, (l+1)\tau), -, -)$ or $(\mathbf{echo}, (\mathtt{fault}, s_j, (l+1)\tau), -, -)$ from at least $n_s - f$ servers. Since $n_s > 2f$, at least one of these servers (say s_i) is correct and, therefore s_i is one of the servers that would have fifo-broadcasted $(\mathbf{echo}, b, -, t_b)$ earlier. This, however, implies from Lemma 5.3.2 that s_k cannot omit to fifo-deliver $(\mathbf{echo}, b, -, t_b)$.

Assume now that $t_b = l\tau$. First assume that s_j had not fifo–broadcasted $(\mathtt{init}, (\mathtt{alive}, s_j, l\tau), -, -)$ before $(\mathtt{init}, b, s_j, t_b)$. All correct servers must have fifo-delivered $(\mathtt{init}, b, s_j, t_b)$ (by Lemma 5.3.8) and fifo-broadcasted $(\mathbf{echo}, b, -, t_b)$ by time $l\tau+\delta$. If s_k did not halt by time $l\tau+2\delta$, it must have fifo-delivered $(\mathbf{echo}, (\mathtt{alive}, s_j, l\tau), -, -)$ or $(\mathbf{echo}, (\mathtt{fault}, s_j, l\tau), -, -)$ from at least $n_s - f$ servers. Since $n_s > 2f$, at least one of these servers (say s_i) is correct and, therefore s_i is one of the servers that would have fifo-broadcasted $(\mathbf{echo}, b, -, t_b)$ earlier. This, however, implies from Lemma 5.3.2 that s_k cannot omit to fifo-deliver $(\mathbf{echo}, b, -, t_b)$.

Finally, assume that s_j had fifo–broadcasted $(\mathtt{init}, (\mathtt{alive}, s_j, l\tau), -, -)$ before $(\mathtt{init}, b, s_j, t_b)$. All correct servers must have fifo–broadcast $(\mathbf{echo}, b, -, t_b)$, and therefore, from Lemma 5.3.2, they must have fifo–broadcast $(\mathbf{echo}, (\mathtt{alive}, s_j, l'\tau), -, -)$ earlier for all $l' \leq l$. Since $n_s > 2f$, s_k cannot get $n_s - f$ copies of $(\mathbf{echo}, (, \mathtt{fault}, s_j, l'\tau), -, -)$ for any $l' \leq l$, and thus $Faulty_k[s_j]$ can become true earliest at time $(l+1)\tau+2\delta$. If s_k fifo-delivers $(\mathbf{echo}, b, -, t_b)$ by time $t_b+2\delta$, then we are one. On the other hand, if s_k omits to fifo-deliver $(\mathbf{echo}, b, -, t_b)$ by time $t_b+2\delta$ then it will never deliver b (by B2). For this case, we show that s_k would halt by time $(l+1)\tau + 2\delta$. If not, s_k must have fifo-delivered $(\mathtt{echo}, (\mathtt{alive}, s_j, (l+1)\tau), -, -)$ or $(\mathtt{echo}, (\mathtt{fault}, s_j, (l+1)\tau), -, -)$ from at least $n_s - f$ servers which was fifo-broadcasted after $(\mathtt{echo}, b, -, t_b)$ was fifo-broadcasted. However, since s_k omitted to fifo-deliver $(\mathtt{echo}, b, -, t_b)$ and $n_s > 2f$, it couldn't have fifo-delivered all these messages (by Lemma 5.3.2).

B11: As for send–omission failures.

□

Theorem 26 *The lower bound for blocking time when $n_s > 2f$, and the lower bound for failover over time in Table 4.1 for receive–omission failures are tight.*

Proof: The procedures have $n_s > 2f$ and $C = 0$, and so the corresponding lower bound on blocking time is tight. Also, from Lemma 5.2.17, the lower bound on failover time is tight because $d = 2\delta$. □

5.3.5 General–Omission failures

Our last failure model is general omission failures. The **broadcast** procedure is given in Figure 5.11. The remaining procedures are the same as receive–omission failures. For this case, we define $d = 2\delta$, $C = 2\delta$ and assume that $n_s > 2f$.

Lemma 5.3.11 *The procedures for general–omission failures satisfy properties* B0—B11.

Proof:

procedure broadcast(M, k)
  $time := current\text{–}time$
  fifo-broadcast(`init`, $M, s_k, time$)
  if by $time + d$ fifo–delivered (`echo`, $M, s_j, time$)
    for at least $n_s - f$ different j **then return**
  else stop

Figure 5.11: Procedures for general–omission failures

B0: Same as send–omission failures.

B1: Similar to send–omission failures.

B2: Similar to receive–omission failures.

B3: From the implementation of **broadcast**.

B4: Follows from Figures 5.4 and 5.10.

B5: Follows from Figures 5.4 and 5.10.

B6: Suppose that s_k sets $Faulty_k[s_j] = true$ at time $l\tau + 2\delta \leq t$ because it did not deliver (`alive`, $s_j, l\tau$). If s_k did not deliver (`alive`, $s_j, l\tau$), it must have fifo-delivered at least $n_s - f$ (`echo`, (`fault`, $s_j, l\tau$), $l\tau, -$). Since $n_s > 2f$, s_j could not have fifo-delivered $n_s - f$ (`echo`, (`alive`, $s_j, l\tau$), $-, -$) and would have, therefore, halted by time $l\tau + 2\delta$.

B7: As in send–omission failures.

B8: Same as Section 5.3.4 if s_k halts in Figure 5.10 because Lemma 5.3.9 continues to hold for general–omission failures. So assume that s_k halts in **broadcast** in Figure 5.11. However, again the argument is the same because at least $n_s - f$ correct servers would have fifo-broadcasted (`echo`, $M, -, time$) and s_k would have fifo-delivered these.

B9: Suppose that s_k sets $Faulty_k[s_j] = true$ at time $l\tau+2\delta$. Then s_k did not deliver ($\texttt{alive}, s_j, l\tau$) and must have fifo-delivered at least $n_s - f$ ($\texttt{echo}, (\texttt{fault}, s_j, l\tau)$ $, -, -$). If s_j fifo-broadcasted ($\texttt{init}, b, s_j, t$) after time $l\tau$, it must have previously fifo-broadcasted ($\texttt{init}, (\texttt{alive}, s_j, l\tau), s_j, l\tau$). Since b was successful, s_j must have fifo-delivered at least $n_s - f$ ($\texttt{echo}, b, s_i, t$) from servers $\{s_i\}$. Therefore, by Lemma 5.3.2, $\{s_i\}$ must have fifo-delivered ($\texttt{init}, (\texttt{alive}, s_j, l\tau), s_j,$ $l\tau$), and therefore could not have fifo–broadcasted ($\texttt{echo}, (\texttt{fault}, s_j, l\tau), s_i, -$). Now since $n_s > 2f$, s_k could not have fifo-delivered at least $n_s - f$ ($\texttt{echo}, ($ $\texttt{fault}, s_j, l\tau), -, -$), a contradiction.

B10: Let b be initiated at time $t_b \leq t$ and let $l\tau \leq t_b < (l+1)\tau$. Assume first that $t_b > l\tau$. We show that s_k cannot omit to dequeue b. Again, as for send–omission failures, we need to prove that s_k would have fifo-delivered ($\texttt{echo}, b, -, t_b$) by time $t_b + 2\delta$.

From Figure 5.11, at least $n_s - f$ servers must have fifo-delivered ($\texttt{init}, b, s_j, t_b$) and fifo-broadcasted ($\texttt{echo}, b, -, t_b$) before time $(l+1)\tau+\delta$. Since s_k did not halt by time $(l+1)\tau + 2\delta$, it must have fifo-delivered ($\texttt{echo}, (\texttt{alive}, s_j, (l+$ $1)\tau), -, -$) or ($\texttt{echo}, (\texttt{fault}, s_j, (l+1)\tau), -, -$) from at least $n_s - f$ servers. Since $n_s > 2f$, at least one of these servers (say s_i) is one of the servers that would have fifo-broadcasted ($\texttt{echo}, b, -, t_b$) earlier. This, however, implies from Lemma 5.3.2 that s_k cannot omit to fifo-deliver ($\texttt{echo}, b, -, t_b$).

Assume now that $t_b = l\tau$. First assume that s_j had not fifo–broadcasted ($\texttt{init}, (\texttt{alive}, s_j, l\tau), -, -$) before ($\texttt{init}, b, s_j, t_b$). At least $n_S - f$ servers must have fifo-delivered ($\texttt{init}, b, s_j, t_b$) and fifo-broadcasted ($\texttt{echo}, b, -, t_b$) by time $l\tau + \delta$. If s_k did not halt by time $l\tau + 2\delta$, it must have fifo-delivered $(\texttt{echo}, (\texttt{alive}, s_j, l\tau), -, -)$ or $(\texttt{echo}, (\texttt{fault}, s_j, l\tau), -, -)$ from at least $n_s - f$ servers. Since $n_s > 2f$, at least one of these servers (say s_i) is one of the servers that would have fifo-broadcasted $(\texttt{echo}, b, -, t_b)$ earlier. This, however, implies from Lemma 5.3.2 that s_k cannot omit to fifo-deliver $(\texttt{echo}, b, -, t_b)$.

Finally, assume that s_j had fifo–broadcasted $(\texttt{init}, (\texttt{alive}, s_j, l\tau), -, -)$ before $(\texttt{init}, b, s_j, t_b)$. At least $n_s - f$ servers must have fifo–broadcast $(\texttt{echo}, b, -, t_b)$, and therefore, from Lemma 5.3.2, these servers must have fifo–broadcast $(\texttt{echo}, (\texttt{alive}, s_j, l'\tau), -, -)$ earlier for all $l' \leq l$. Since $n_s > 2f$, s_k cannot get $n_s - f$ copies of $(\texttt{echo}, (, \texttt{fault}, s_j, l'\tau), -, -)$ for any $l' \leq l$, and thus $Faulty_k[s_j]$ can become true earliest at time $(l+1)\tau + 2\delta$. If s_k fifo-delivers $(\texttt{echo}, b, -, t_b)$ by time $t_b + 2\delta$, then we are one. On the other hand, if s_k omits to fifo-deliver $(\texttt{echo}, b, -, t_b)$ by time $t_b + 2\delta$ then it will never deliver b (by B2). For this case, we show that s_k would halt by time $(l+1)\tau + 2\delta$. If not, s_k must have fifo-delivered $(\texttt{echo}, (\texttt{alive}, s_j, (l+1)\tau), -, -)$ or $(\texttt{echo}, (\texttt{fault}, s_j, (l+1)\tau), -, -)$ from at least $n_s - f$ servers which was fifo-broadcasted after $(\texttt{echo}, b, -, t_b)$ was fifo-broadcasted. However, since s_k omitted to fifo-deliver $(\texttt{echo}, b, -, t_b)$ and $n_s > 2f$, it couldn't have fifo-delivered all these messages (by Lemma 5.3.2).

B11: As in send–omission failures.

□

Theorem 27 *The lower bounds in Table 4.1 for general–omission failures are all tight.*

Proof: Properties B0—B11 are met for $n_s > 2f$ which is optimal. From the procedure **broadcast**, $C = 2\delta$, which is optimal. Finally, since Lemma 5.2.17 holds for all $\tau > 0$, the failover time is also optimal because $d = 2\delta$. □

```
procedure primary()
  cobegin
    || inform-clients("Dest = s_0")
    || status := OK
       do forever
         when received request from client c
           response := Π(state_0, request)
           state_0 := state_0 ∘ response
           status :=broadcast((log, s_0, (response, c)), 0)
           if status = NOTOK then
             failure-mode-primary(0)
       od
  coend

procedure backup()
  do forever
    ((tag, s_0, (r, c)), -) := Deq(Rqueue_1)

    /* logging response from primary */
    if tag = log then
      state_1 := state_1 ∘ r
      send r to client c

    /* becoming the primary */
    if Faulty_1[s_0] then failure-mode-primary(1)
  od

procedure failure-mode-primary(k)
  cobegin
    || inform-clients("Dest = s_k")
    || do forever
         when received request from client c
           response := Π(state_k, request)
           state_k := state_k ∘ response
           send response to client c
       od
  coend
```

Figure 6.2: Procedures for send–omission failures($n_s = 2, f = 1$)-i

```
procedure initialize(k)
   state_k := Rqueue_k := Dqueue_k := ε
   ∀i : Faulty_k[s_i] := false

procedure broadcast(M, k)
   j := k̄
   time := current–time
   fifo-send(init, M, s_k, time)
   if by time + 2δ: fifo-delivered (echo, M, s_j, time)
      then return(OK)
   if ⌈time/τ⌉τ = time then t := time + τ
   else t := time
   when current–time= ⌈t/τ⌉τ + 2δ:
      return(NOTOK)

procedure deliver (M, k)
   Let M = (tag, –, –)
   if tag = log then Enq(Rqueue_k, (M, k))

procedure delivery-process(k)
   j := k̄
   cobegin
      || fifo-delivery-process(k)
      || do forever
            (tag, M, s_j, t) := Deq(Dqueue_k)
            if tag = init then
               fifo-send (echo, M, s_k, t)
               deliver (M, k)
         od
   coend
```

Figure 6.3: Procedures for send–omission failures($n_s = 2, f = 1$)-ii

other server. Lemma 5.3.2 continues to hold with **fifo–broadcast** replaced by **fifo–send**. As required by Corollary 6.0.1, this protocol is also "pass-the-buck". Again we have assumed that there are no omission failures between the clients and the servers. This assumption can be removed in a manner similar to Chapter 5.

---

procedure failure-detector(k)
  $A_j^l \triangleq$ (`alive`, $s_j, l\tau$)
  $F_j^l \triangleq$ (`fault`, $s_j, l\tau$)
  $j := \bar{k}$
  cobegin
    || **for** $l := 0$ to ∞
        when *current–time* $= l\tau$:
          fifo-send(`init`, $A_k^l, s_k, l\tau$)
    || **for** $l := 0$ to ∞
        when *current–time* $= l\tau + \delta$:
          if not fifo-delivered (`init`, $A_j^l, s_j, l\tau$) **then**
            fifo-send (`echo`, $F_j^l, s_k, l\tau$)
    || **for** $l := 0$ to ∞
        when *current–time* $= l\tau + 2\delta$:
          if fifo-delivered (`echo`, $F_k^l, s_j, l\tau$) **then stop**
          if not delivered A_j^l **then** $Faulty_k[s_j] := true$
  coend

Figure 6.4: Procedures for send–omission failures($n_s = 2, f = 1$)-iii

---

There are two servers—s_0 and s_1. Initially, s_0 executes the procedure **primary** and s_1 executes the procedure **backup**. Since the protocol is "pass the buck", initially the backup sends responses to the clients.

As long as there are no failures, it can be shown that the procedure **broadcast** (described later) continues to return the status of OK. However, if a status of $NOTOK$ is returned, then again it can be shown that the backup s_1 has failed either by crashing or by send–omission. In this case, since $f = 1$ and s_1 is faulty, s_0 must be correct and will never crash. Therefore, s_0 continues to be the primary by calling **failure–mode–primary**. Since s_0 is correct, it can be shown that s_1 will never become the primary. As a result, s_0 does not send information about state updates to s_1 in **failure–mode–primary**. Furthermore, now it is the primary (*i.e.* s_0) that sends responses to the clients.

On the other hand, if the backup s_1 is correct (notice that in this case **broadcast** can never return a status of $NOTOK$) and detects that the primary s_0 has crashed (by using the **failure–detector** in Figure 6.4), then s_1 becomes the primary by calling **failure–mode–primary**.

As mentioned above, the procedure **broadcast** is used to send state updates to s_1 as long as s_0 is executing the procedure **primary**. The **broadcast** procedure sends a message to s_1, and then waits for an acknowledgement. If an acknowledgement does not arrive, then a failure must have occured. If s_0 is faulty, then the description given earlier requires that a status of $NOTOK$ never be returned. This is ensured because of the following reason. If s_0 is faulty, then it must have omitted to send the message to s_1. By Lemma 5.3.2, s_1 will not fifo–deliver the next `alive` message that s_0 sends, and therefore, will send the `fault` message in Figure 6.4. As a result, it can now be shown that s_0 will halt (either by crashing or by fifo–delivering the above `fault` message in Figure 6.4) before it can return a status of $NOTOK$ in procedure **broadcast**.

On the other hand, if s_0 is correct, then **broadcast** will return a status of $NOTOK$ as required.

6.1.1 Proof of correctness

We now formally show that the protocol in Figures 6.1,6.2, 6.3 and 6.4 satisfies Pb1—Pb5.

Define:

- $Prmy_{s_0} \triangleq s_0$ has not halted
- $Prmy_{s_1} \triangleq Faulty_1[s_0] = true \wedge s_1$ has not halted.

Theorem 30 *The protocol in Figures 6.1,6.2, 6.3 and 6.4 satisfies* Pb1.

Proof: Suppose that $Prmy_{s_1}$ is true at some time t. Let $Prmy_{s_1}$ become true for the first time at $l\tau + 2\delta \leq t$ because $Faulty_1[s_0] = true$ at time $l\tau + 2\delta$ in Figure 6.4. s_1 must not have delivered A_0^l, which implies that s_0 is faulty (as there are only send omission failures). Since s_1 is correct, it would have fifo-sent $(\mathtt{echo}, F_0^l, s_1, -)$ at time $l\tau + \delta$, and s_0 would have halted by time $l\tau + 2\delta$. Thus $Prmy_{s_0}$ cannot be true at time $t \geq l\tau + 2\delta$. □

Theorem 31 *The protocol in Figures 6.1,6.2, 6.3 and 6.4 satisfies* Pb2

Proof: Follows from Figure 5.2. □

Theorem 32 *The protocol in Figures 6.1,6.2, 6.3 and 6.4 satisfies* Pb3

Proof: Easy to see since server s_j $j = 0, 1$ executes **inform–clients** only after it becomes the primary. □

Lemma 6.1.1 *Let* $k = \bar{j}$. *If* s_j *broadcasts* b *before* b', *then* s_k *cannot deliver* b' *before* b.

Proof: Suppose s_k delivered b'. Thus s_k fifo-delivered $(\texttt{init}, b', s_j, -)$. Since s_j fifo-broadcasted $(\texttt{init}, b, s_j, -)$ before $(\texttt{init}, b', s_j, -)$, by Lemma 5.3.2, s_k would have fifo-delivered $(\texttt{init}, b, s_j, -)$ before $(\texttt{init}, b', s_j, -)$, and hence delivered b before b'. □

Lemma 6.1.2 *No correct server ever halts.*

Proof: For contradiction, assume that s_j is the first correct server to halt at time t and let $k = \bar{j}$.

This implies that $t = l\tau + 2\delta$ because s_j must have halted in Figure 6.8 after it fifo-delivered $(\texttt{echo}, F_j^l, s_k, -)$. Since s_j is correct, it must have fifo-sent $(\texttt{init}, A_j^l, s_j, -)$ at time $l\tau$ and s_k must have fifo-delivered this by time $l\tau + \delta$ or s_k must have halted. However, this implies that s_k could not have fifo-sent $(\texttt{echo}, F_j^l, s_k, -)$, a contradiction. □

Lemma 6.1.3 *If s_j, $j = 0, 1$, calls* **failure–mode–primary**, *then s_j is correct.*

Proof: For contradiction, assume that s_j is faulty and calls **failure–mode–primary** at time $t = l\tau + 2\delta$.

Case 1: $j = 1$. From Theorem 30, s_0 must have halted and from Lemma 6.1.2 s_1 must be correct.

Case 2: $j = 0$. Assume for contradiction that s_0 is faulty and s_1 is correct. From Figure 6.3, s_0 did not fifo-deliver $(\texttt{echo}, M, s_1, -)$ for some M which, since s_1 is correct, implies that s_1 did not fifo-deliver $(\texttt{init}, M, s_0, -)$ which s_0 was supposed to send at time t' (say). If $\lceil \frac{t'}{\tau} \rceil \tau = t'$, then let $t'' = t' + \tau$, else let

$t'' = t'$. Therefore, from Figure 6.3, $\lceil \frac{t''}{\tau} \rceil \tau + 2\delta = t = l\tau + 2\delta$, or $\lceil \frac{t''}{\tau} \rceil = l$. Since s_1 did not fifo-deliver $(\mathtt{init}, M, s_0, -)$, from Lemma 5.3.2, it will not fifo-deliver $(\mathtt{init}, A_0^l, s_0, -)$, and so will fifo-send $(\mathtt{echo}, F_0^l, s_1, -)$. This, however, implies that s_0 would have fifo-delivered $(\mathtt{echo}, F_0^l, s_1, -)$ and halted by time $l\tau + 2\delta$. Thus, s_0 cannot call **failure–mode–primary**, a contradiction. □

Lemma 6.1.4 *If $Prmy_{s_0}$= false at time t, then $Prmy_{s_1}$= true by time $t + \tau + 2\delta$.*

Proof: If s_0 halts at time t, where $(l-1)\tau \le t < l\tau$, then by Lemma 6.1.2, s_0 is faulty and s_1 is correct and so s_1 will never halt. s_1 will not deliver A_0^l and become the primary by time $l\tau + 2\delta \le t + \tau + 2\delta$. □

Lemma 6.1.5 *Let s_1 become the primary. Then if s_0 sent sequence of broadcasts $b_0 \circ b_1 \ldots b_n$ then*

1. *s_1 must have delivered $b_0 \circ b_1 \ldots b_{n-1}$ by time $\uparrow 1$.*
2. *if $b_n = (\mathtt{log}, s_0, (r_n, -))$ and r_n was sent to the client, then s_1 must have delivered b_n by time $\uparrow 1$.*

Proof: If s_1 becomes the primary, then by Lemma 6.1.3, s_1 must be correct, and s_0 must have halted before calling **failure–mode–primary**.

Since s_0 must have broadcasted b_i, $i = 0 \ldots n-1$, and did not call **failure–mode–primary**, from Figures 6.2 and 6.3, s_0 must have fifo–delivered $(\mathtt{echo}, b_i, s_1, -)$, which implies that s_1 must have delivered b_i. (1) now follows from Lemma 6.1.1.

(2) is trivially true since only s_1 could have sent r_n to the client before becoming the primary. □

Lemma 6.1.6 *If s_1 becomes the primary and s_0 halt at time $\downarrow 0 \leq \uparrow 1$, then*

1. *$Init_1 \sqsubseteq Final_0$*
2. *If $Final_0 = M_0$ and $R_0 \neq \epsilon$, where $R_0 \sqsubset M_0$, then R_0 was sent to the client.*
3. *If $Init_1 \sqsubset Final_0$, then $last(Final_0)$ was not sent to the client.*

Proof: As before, s_0 must have halted before it could have called **failure–mode–primary**. Let $Final_0 = r_0 \circ r_1 \ldots \circ r_n$. If s_0 broadcasted $(\texttt{log}, s_0, (r_n, -))$ then it must broadcasted $(\texttt{log}, s_0, (r_i, -))$, $i = 0 \ldots n$, and therefore from Lemma 6.1.5, $(\texttt{log}, s_0, (r_i, -))$, $i = 0 \ldots n-1$, must have been delivered at s_1 and (1) is satisfied.

Now assume that s_0 does not broadcast $(\texttt{log}, s_0, (r_n, -))$ and therefore halts. Therefore, from Lemma 6.1.2, s_0 is faulty and so s_1 is correct. s_0 must have broadcasted $(\texttt{log}, s_0, (r_i, -))$, $i = 0 \ldots n-1$ and received $(\texttt{echo}, (\texttt{log}, s_0, (r_i, -)), s_1)$. (1) now follows from Lemma 6.1.1.

(2) can be seen to be true from Figure 6.2 because s_0 must have fifo-delivered $(\texttt{echo}, (\texttt{log}, s_0, r_i), s_1)$, $i = 0 \ldots n-1$ and since s_1 is correct, it must have sent r_i, $i = 0 \ldots n-1$, to the client.

(3) is trivially true because only s_1 could have sent the response to the client. □

Let $\rho = n_c(\tau + 2\delta) + \delta$.

Lemma 6.1.7 *If s_0 remains the primary through time t, then there can be at most one request from any client i in the queue of s_0 at time t.*

Proof: Assume, for contradiction, that $t_2 \leq t$ is the first time that there are two requests (say m_1 and m_2) from some client i in the queue of s_0. Assume m_1 was sent at time t_1 and so it was enqueued by $t_1 + \delta$. From Figure 5.2, $2\rho = 2(n_c(\tau + 2\delta) + \delta)$ and so client i cannot send m_2 before time $t_1 + 2(n_c(\tau + 2\delta) + \delta)$ *i.e.* $t \geq t_2 \geq t_1 + 2(n_c(\tau + 2\delta) + \delta)$. Since m_2 is the first duplicate request, there are at most $n_c - 1$ requests ahead of m_1 in s_0's queue Therefore, from Figures 6.2 and 6.3 , s_0 will dequeue m_1 by time $t_1 + \delta + (n_c - 1)(\tau + 2\delta) < t$. Therefore, $t_2 \leq t_1 + \delta + (n_c - 1)(\tau + 2\delta)$ which is a contradiction. □

Lemma 6.1.8 *If a request m is sent by the client at time t_m to server s_j, and s_j remains the primary through the interval $[t_m..t_m + \rho]$, then m is received by s_j by time $t_m + \rho$.*

Proof: Since the message delivery time is at most δ, m is enqueued at s_j by time $t_m + \delta$. If $j = 0$, then from Lemma 6.1.7, there can be at most $n_c - 1$ requests ahead of m in s_j's queue. Therefore, from Figures 6.2 and 6.3, m is received by time $t_m + (n_c - 1)(\tau + 2\delta) + \delta \leq t_m + \rho$.

On the other hand, if $j = 1$, then from Figure 6.2, m is received by time $t_m + \delta \leq t_m + \rho$. □

Lemma 6.1.9 *Let s_0 call* **failure–mode–primary** *at time $\downarrow 0$ and $state_0(\downarrow 0) = r_0 \circ r_1 \ldots r_n$. Let m_n be the request corresponding to r_n. Then m_n must have been received in the interval $[max(0, \downarrow 0 - \tau - 2\delta)..\downarrow 0]$.*

Proof: From Lemma 6.1.2, s_0 is correct and did not deliver $(\texttt{echo}, (\texttt{log}, s_0, (r_n, -)), s_1, -)$. Suppose for contradiction that m_n was received at time $t < \downarrow 0 - \tau - 2\delta$. Since s_0 did not deliver $(\texttt{echo}, (\texttt{log}, s_0, (r_n, -)), s_1, -)$, it must have called **failure–mode–primary** by time $t + \tau + 2\delta < \downarrow 0$, a contradiction. □

Lemma 6.1.10 *Let s_0 halt and $Final_0 = r_0 \circ r_1 \ldots r_n$. Let m_n be the request corresponding to r_n. If r_n was not sent to the client, then m_n must have been sent in the interval $[max(0, \downarrow 0 - \tau - 2\delta - \rho)..\downarrow 0]$ and received in the interval $[max(0, \downarrow 0 - \tau - 2\delta)..\downarrow 0]$.*

Proof: From Lemma 6.1.2, s_0 is faulty and could not have called **failure–mode–primary**. Suppose for contradiction that m_n was sent in the interval $[0..\downarrow 0 - \tau - 2\delta - \rho)$. From Lemma 6.1.8, m_n must have been received by time $t < \downarrow 0 - \tau - 2\delta$ If $\lceil \frac{t}{\tau} \rceil \tau = t$, then let $t' = t + \tau$, else let $t' = t$. Since s_0 has not halted at time $t + \tau + 2\delta < \downarrow 0$, s_0 could not have fifo-delivered $(\texttt{echo}, (\texttt{fault}, s_0, \lceil \frac{t'}{\tau} \rceil \tau), s_1, -)$. Since s_1 is correct, this implies that s_1 must have fifo-delivered $(\texttt{init}, (\texttt{alive}, s_0, \lceil \frac{t'}{\tau} \rceil \tau), s_0, -)$ and therefore also fifo-delivered $(\texttt{init}, (\texttt{log}, s_0, (r_n, -)), s_0, -)$ (by Lemma 5.3.2), which implies that r_n must have been sent to the client, a contradiction.

For the second part, if m_n is received at some time $t < \downarrow 0 - \tau - 2\delta$, then this is a contradiction as proved in the previous paragraph. Therefore m_n can be only received in the interval $[max(0, \downarrow 0 - \tau - 2\delta)..\downarrow 0]$. □

Lemma 6.1.11 *For any run σ_{PB} of the primary–backup system there exists a run*

σ_P of the $(\rho, 2, 2\tau + 5\delta + \rho)$ bofo server P such that $\forall j : \sigma_P \approx_j \sigma_{PB}$.

Proof: **Case 1:** $Prmy_{s_0}$ is always true in σ_{PB} and s_0 never calls **failure–mode–primary**. If s_0 makes infinitely many broadcasts in σ_{PB}, then from Figure 6.6, all requests have responses. Therefore, assume that all the requests m that s_1 responds to are responded to in σ_P as well. m is received at the same times in σ_{PB} and σ_P. However, m is enqueued at the same time it is received in σ_P (this has to be true of all runs of P). We can do this from Lemma 6.1.8. Furthermore, in σ_P the response r has to be sent at the same time the request m is received, whereas in σ_{PB}, r might be sent after δ time units by s_1. However, r has to be received at the same times by the clients in σ_{PB} and σ_P. We can do this because r is received by the client within 2δ after m is received, and $\rho \geq 2\delta$. However, if s_0 makes only finitely many broadcasts, then again from Figure 6.6, the last request might not have a response. Assume that the last request m_n is received by s_0 at time t. Let the first (and only) outage interval of σ_P be $[t..t]$ and assume in σ_P that the request m is received, but a response never sent.

Case 2: $Prmy_{s_0}$ is always true in σ_{PB}, but s_0 calls **failure–mode–primary** at time $\downarrow 0$. We construct σ_P until time $\downarrow 0$, and then show how to extend it. Let $Final_0 = r_0 \circ r_1 \ldots r_{n-1} \circ r_n$. Again, as in case 1, the same responses are sent in both σ_{PB} and σ_P. However, as can be seen from Figures 6.2 and 6.3, r_{n-1} and r_n might not have been sent to the client. Let m_{n-1} and m_n be the requests corresponding to r_{n-1} and r_n respectively, which were received at times t_{n-1} and t_n respectively. Let the two outage intervals of σ_P be $I_1 = [t_{n-1}..t_{n-1}]$ and $I_2 = [max(0, \downarrow 0 - \tau - 2\delta)..\downarrow 0]$. If r_{n-1} was not sent to the client, assume that in σ_P, m_{n-1} was received, but a response never sent (outage belongs to I_1).

Similarly for r_n since by Lemma 6.1.9, m_n must have been received in I_2.

σ_P can be easily extended beyond $\downarrow 0$ because all requests received after $\downarrow 0$ will have a response.

Case 3: s_0 halts at time $\downarrow 0$ in σ_{PB}. We first construct the run σ_P until time $\uparrow 1$, and then show how to extend it.

By Lemma 6.1.4, s_1 becomes the primary at time $\uparrow 1 \leq \downarrow 0 + \tau + 2\delta$. As in case 1, all requests sent in the interval $[0..\downarrow 0]$ that have responses by s_1 in σ_{PB} are responded to in σ_P as well. Let $Final_0 = r_0 \circ r_1 \ldots r_{n-1} \circ r_n$. By Lemma 6.1.6, r_n might not have been sent to the client. As in case 2, let m_n be the request corresponding to r_n which was received at time t_n by s_0. Let the only outage interval of σ_P be $I_1 = [max(0, \downarrow 0 - \tau - 2\delta)..\uparrow 1]$.

From Lemma 6.1.6, $Init_1 = R = r_0 \circ r_1 \ldots r_{n-1}$ or $Init_1 = M = r_0 \circ r_1 \ldots r_{n-1} \circ r_n$. If $Init_1 = R$, then r_n was not sent to the client and from Lemma 6.1.10, m_n was sent in the interval $[max(0, \downarrow 0 - \tau - 2\delta - \rho)..\downarrow 0]$. We now show that $Init_1 = state_P(\uparrow 1)$. From Lemma 6.1.6, $r_0 \circ r_1 \ldots r_{n-1}$ must have been sent to the client and from construction, these responses are sent in σ_P as well. Therefore, we are done if we can show that P receives no further requests until time $\uparrow 1$. Assume in σ_P that m_n is never enqueued (outage belongs to I_1) or hasn't yet been enqueued, depending on the time m_n is actually sent. Futhermore, all other requests that were sent in the interval $[0..\uparrow 1]$ that haven't been received in $\sigma_{PB}[\uparrow 1]$ must have been sent in the interval $[\downarrow 0 - \rho..\uparrow 1]$ (Lemma 6.1.8). Assume, therefore, that in σ_P these requests are never enqueued (outage belongs to I_1) or haven't yet been enqueued in σ_P. Therefore P receives no further requests and $Init_1 = state_P(\uparrow 1)$.

On the other hand, if $Init_1 = M$, then r_n must have been sent to the client and it can again be shown that $Init_1 = state_P(\uparrow 1)$.

We now extend σ_P. Extend I_1 as $I_1 = [max(0, \downarrow 0 - \tau - 2\delta)..\uparrow 1 + \delta + \rho]$. Therefore, $|I_1| \leq 2\tau + 5\delta + \rho$ (from Lemma 6.1.4). We first show that $\forall j : queue_{cj,P}(\uparrow 1) = \phi$. By the above construction of σ_P, all requests sent before $\uparrow 1$, other than m_n, that are received in $\sigma_{PB}[\uparrow 1]$ are received in σ_P as well. Also, m_n is not enqueued at time $\uparrow 1$. Finally, all requests sent before $\uparrow 1$ that are not received in $\sigma_{PB}[\uparrow 1]$ must must have been sent in $[\downarrow 0 - \rho..\uparrow 1]$ (by Lemma 6.1.8). Assume that these are never enqueued in σ_P (again outages lies in I_1) or haven't yet been enqueued. Thus, $\forall j : queue_{cj,P}(\uparrow 1) = \phi$. Furthermore, requests sent before $\uparrow 1$ to s_0 will never be enqueued after $\uparrow 1$ in σ_P

It is now easy to extend σ_P. Assume now that all requests that clients send in $[\uparrow 1..\uparrow 1 + \delta]$ to s_0 are never enqueued in σ_P. This outage again lies in I_1. However, any respose that s_1 sends in σ_{PB} after $\uparrow 1$ is sent in σ_P as well. We can do this since $Init_1 = state_P(\uparrow 1)$, $queue_{cj,P}(\uparrow 1) = \phi = queue_{cj,s_0}(\uparrow 1)$ and no request sent to s_0 is enqueued after $\downarrow 0$ in σ_P . Since, by Lemma 6.1.3, s_1 is correct and will never halt, we are done. $\square$

Theorem 33 *The protocol in Figures 6.1,6.2, 6.3 and 6.4 satisfies* Pb4.

Proof: Follows from Lemma 6.1.11. $\square$

Theorem 34 *The protocol in Figures 6.1,6.2, 6.3 and 6.4 satisfies* Pb5.

Proof: A server continues to be the primary until it halts. The result now follows from Lemma 6.1.2. $\square$

Theorem 35 *The protocol in Figures 6.1,6.2, 6.3 and 6.4 is δ–blocking.*

Proof: Follows from the description of the protocol. □

6.2 General–Omission Failures

In this section, we give the δ–blocking protocol tolerating general–omission failures. The protocol is shown in Figures 6.5,6.6, 6.7 and 6.8, and for this protocol $f = 1$ and $n_s = 2f+1 = 3$. The procedures **initialize, deliver** and **failure–mode–primary** are the same as those for send–omission failures. As required by Corollary 6.0.2, this protocol is "pass-the-buck".

---

initialize(i)
cobegin
  || **if** $i = 0$ **then primary**() **else if** $i = 1$ **then backup1**() **else backup2**()
  || **delivery–process**(i)
  || **failure–detector**(i)
coend

Figure 6.5: Protocol run by server s_i for general–omission failures($n_s = 3, f = 1$)

There are three servers—s_0, s_1 and s_2. As we will see below, in this protocol, only s_0 and s_2 can become primaries. Initially, s_0 executes the procedure **primary**, s_1 executes the procedure **backup1** and s_2 executes the procedure **backup2**. Since the protocol is "pass the buck", initially the backup s_1 sends responses to the clients. Furthermore, since s_1 can never become the primary, it does not need to keep its state up-to-date with the state of the primary.

As long as there are no failures, the procedure **broadcast** continues to return the status of OK. However, if a status of $NOTOK$ is returned, then it can be seen that either s_0 or s_1 must be faulty, which implies that s_2 is correct. The protocol now ensures that s_2 becomes the new primary by calling **failure–mode–primary**. This is done as follows. When **broadcast** returns a status of $NOTOK$, s_0 calls the procedure **inform–new–primary**, and sends a `takeover` message. If it receives an acknowledgement to this message from s_2, then it knows that s_1 will become the primary by calling **failure–mode–primary** in Figure 6.6. If it does not receive an acknowledgement, then it knows that it is faulty (since s_2 is correct as argued above), and halts itself. In this case, s_2 will again become the primary as it will detect that s_0 has halted in Figure 6.8. Once s_2 becomes the primary, it continues to be the primary since it is correct.

6.2.1 Proof of correctness

We now formally show that the protocol in Figures 6.5,6.6, 6.7 and 6.8 satisfies Pb1—Pb5.

Define:

- $Prmy_{s_0} \triangleq status = OK \wedge s_0$ has not halted
- $Prmy_{s_1} \triangleq false$
- $Prmy_{s_2} \triangleq (Faulty_2[s_0] = true \vee$ delivered $(\texttt{takeover}, s_0, -)) \wedge s_2$ has not halted.

Theorem 36 *The protocol in Figures 6.5,6.6, 6.7 and 6.8 satisfies* Pb1.

```
procedure primary() /* server s_0 */
  cobegin
    || inform-clients("Dest = s_0")
    || status := OK
       do forever
          when received request from client c
             response := Π(state_0, request)
             state_0 := state_0 ∘ response
             status :=broadcast((log, s_0, (response, c)), 0)
             if status = NOTOK then inform-new-primary()
       od
  coend

procedure backup1() /* server s_1 */
  do forever
    ((tag, s_0, (r, c)), −) := Deq(Rqueue_1)

    /* responding to client */
    if tag = log and fifo-delivered (init, (tag, s_0, (r, c)), s_0, −) then
       send r to client c
  od

procedure backup2() /* server s_2 */
  do forever
    ((tag, s_0, (r, c)), −) := Deq(Rqueue_2)

    /* logging response from primary */
    if tag = log then state_2 := state_2 ∘ r

    /* becoming the primary */
    if Faulty_2[s_0] or delivered (takeover, s_0, −) then
       failure-mode-primary(2)
  od
```

Figure 6.6: Procedures for general–omission failures($n_s = 3, f = 1$)-i

```
procedure inform–new–primary()
  time := current–time
  fifo-broadcast(init, (takeover, s0, time), s0, time)
  if not fifo-delivered (echo, (takeover, s0, time), s2, time)
  by time + 2δ then stop
  else loop forever

procedure broadcast(M, k)
  time := current–time
  fifo-broadcast(init, M, sk, time)
  if by time + 2δ not fifo-delivered (echo, M, sk+1, time) then
    return(NOTOK)
  else return(OK)

procedure delivery-process(k)
  cobegin
    || fifo-delivery-process(k)
    || do forever
         (tag, M, −, t) :=Deq(Dqueuek)
         if tag = init then fifo-broadcast (echo, M, sk, t)
         if tag = echo and not delivered M before then
           deliver (M, k)
       od
  coend
```

Figure 6.7: Procedures for general–omission failures($n_s = 3, f = 1$)-ii

procedure failure-detector(k)
  $\forall i, j : A^l_j \triangleq (\texttt{alive}, s_j, l\tau)$
  $\forall i, j : F^l_j \triangleq (\texttt{fault}, s_j, l\tau)$
  cobegin
    || **for** $l := 0$ to ∞
        when *current–time* $= l\tau$:
          fifo-broadcast($\texttt{init}, A^l_k, s_k, l\tau$)
    || **for** $l := 0$ to ∞
        when *current–time* $= l\tau + \delta$:
          $\forall j$: **if** not fifo-delivered $(\texttt{init}, A^i_j, s_j, l\tau)$ **then**
            fifo-broadcast $(\texttt{echo}, F^l_j, s_k, l\tau)$
    || **for** $l := 0$ to ∞
        when *current–time* $= l\tau + 2\delta$:
          $witness_k[k] := \{s_j|$fifo-delivered $(\texttt{echo}, A^l_k, s_j, l\tau)\}$
          $\forall j \neq k : witness_k[j] :=$
            $\{s_i|$fifo-delivered $(\texttt{echo}, A^l_j, s_i, l\tau)$
             or fifo-delivered $(\texttt{echo}, F^l_j, s_i, l\tau)\}$
          if $\exists j : |witness_k[j]| < 2$ **then stop**
          if $\exists j$: not delivered A^l_j **then** $Faulty_k[s_j] := true$
  coend

Figure 6.8: Procedures for general–omission failures($n_s = 3, f = 1$)-iii

Proof: We show that $\neg(Prmy_{s_0} \wedge Prmy_{s_2})$ at any time. Suppose that $Prmy_{s_2}$ is true at some time t.

Case 1: $Prmy_{s_2}$ becomes true for the first time at $l\tau + 2\delta \leq t$ because $Faulty_2[s_0] = true$ at that time (Figure 6.8). s_2 must have fifo–delivered at least two copies of $(\texttt{echo}, F^l_0, -, -)$, which implies that s_0 could not have fifo–delivered two copies of $(\texttt{echo}, A^l_0, -, -)$ because $n_s = 3$. Therefore, from Figure 6.8, s_0 would have halted by time $l\tau + 2\delta$. Thus $Prmy_{s_0}$ cannot be true at time $t \geq l\tau + 2\delta$.

Case 2: $Prmy_{s_2}$ becomes true for the first time at $t' \leq t$ because it delivered $(\texttt{takeover}, s_0, -)$ at time t'. This however implies that s_0 must have called **inform-new-primary** (Figure 6.7) and so $status = NOTOK$ (Figure 6.6) by time t' and so $Prmy_{s_0}$ cannot be true at time $t \geq t'$. $\square$

Theorem 37 *The protocol in Figures 6.6, 6.7 and 6.8 satisfies* Pb2.

Proof: Follows from Figure 5.2. $\square$

Since s_0 can become the backup and still not halt, in order to satisfy Pb3, we need some mechanism such that no requests are enqueued at s_0 after it becomes the backup. So we make the assumption that such a mechanism exists and discards all requests that arrive at a backup. Pb3 now holds.

Theorem 38 *The protocol in Figures 6.6, 6.7 and 6.8 satisfies* Pb3.

Proof: Follows from the above assumption. $\square$

Lemma 6.2.1 *If s_0 broadcasts b before b', then s_2 cannot deliver b' before b.*

Proof: Suppose s_2 delivered b' before b and s_2 fifo-delivered $(\mathtt{echo}, b', s_i, -)$ from s_i. This implies (from Figure 6.7) that s_i must have fifo-delivered $(\mathtt{init}, b', s_0, -)$ from s_0. Since s_0 fifo-broadcasted $(\mathtt{init}, b, s_0, -)$ before $(\mathtt{init}, b', s_0, -)$, by Lemma 5.3.2 s_i must have fifo-delivered $(\mathtt{init}, b, s_0, -)$ before $(\mathtt{init}, b', s_0, -)$, and therefore must have fifo-broadcasted $(\mathtt{echo}, b, s_i, -)$ before $(\mathtt{echo}, b', s_i, -)$. Again by Lemma 5.3.2, s_2 must have fifo-delivered $(\mathtt{echo}, b, s_i, -)$ before $(\mathtt{echo}, b', s_i, -)$ and hence delivered b before b'. $\square$

Lemma 6.2.2 *No correct server ever halts.*

Proof: For contradiction, assume that s_i is the first correct server to halt at time t.

Case 1: s_i halted in Figure 6.8. This implies that $t = l\tau + 2\delta$. First suppose $witness_i[i] < 2$. However, since s_i is correct and at least one of the other two servers is correct (say s_j) and hasn't halted by time $t = l\tau + \delta$ (s_i is the first correct server to halt), s_i will fifo-deliver $(\texttt{echo}, A_i^l, s_i, -)$ and $(\texttt{echo}, A_i^l, s_j, -)$ and so cannot halt. Similarly if $witness_i[k] < 2$, $k \neq i$, then s_i will fifo-deliver $(\texttt{echo}, \{A_k^l, F_k^l\}, s_i, -)$ and $(\texttt{echo}, \{A_k^l, F_k^l\}, s_j, -)$ and so cannot halt.

Case 2: s_i halted in **inform–new–primary** in Figure 6.7 and so $i = 0$. s_0 called **inform–new–primary** because it did not fifo-deliver $(\texttt{echo}, M, s_1, -)$ for some M, and so s_1 is faulty (s_0 is correct by assumption). Hence, s_2 is correct and so s_0 will fifo-deliver $(\texttt{echo}, (\texttt{takeover}, s_0, -), s_2, -)$ and cannot therefore cannot halt. □

Lemma 6.2.3 *If s_2 becomes the primary, then s_2 is correct.*

Proof: For contradiction, assume that s_2 is faulty and becomes the primary at time t.

Case 1: $l\tau + 2\delta = t$ because $Faulty_2[s_0] = true$ at time $l\tau + 2\delta$. s_2 must have fifo–delivered at least two copies of $(\texttt{echo}, F_0^l, -, -)$, which implies that s_0 could not have fifo–delivered two copies of $(\texttt{echo}, A_0^l, -, -)$ because $n_s = 3$. Thus so s_0 would have halted by time $l\tau + 2\delta$ and from Lemma 6.2.2, s_0 is faulty, a contradiction since $f = 1$.

Case 2: s_2 delivered $(\texttt{takeover}, s_0, -)$ at time t. Thus, s_0 did not fifo-deliver $(\texttt{echo}, M, s_1, -)$ for some M in Figure 6.7 and so either s_1 or s_0 is faulty, again a contradiction since $f = 1$. □

Lemma 6.2.4 *If $Prmy_{s_0}$= false at time t, then $Prmy_{s_2}$= true by time $t + \tau + 4\delta$.*

Proof: If s_0 halts at time t, where $(l-1)\tau \leq t < l\tau$, then by Lemma 6.2.2, s_0 is faulty and s_2 is correct and so s_2 will never halt. s_2 will not deliver A_0^l and become the primary by time $l\tau + 2\delta \leq t + \tau + 2\delta$.

On the other hand if $status = NOTOK$ at time t, then s_0 did not fifo-deliver $(\mathtt{echo}, M, s_1, -)$ for some M and so either s_1 or s_0 is faulty. Therefore s_2 is correct and will never halt by Lemma 6.2.2. If s_0 does not halt by time $t+2\delta$ in Figure 6.7, then s_2 delivered $(\mathtt{takeover}, s_0, -)$ by $t + 2\delta$. On the other hand, if s_0 halts by time $t + 2\delta$, where $(l-1)\tau \leq t + 2\delta < l\tau$, then again s_2 will become the primary by time $l\tau + 2\delta \leq t + \tau + 4\delta$. □

Lemma 6.2.5 *Let s_2 become the primary at time $\uparrow 2$. Then if s_0 sent sequence of broadcasts $b_0 \circ b_1 \ldots b_n$ then*

1. *s_2 must have delivered $b_0 \circ b_1 \ldots b_{n-1}$ by time $\uparrow 2$.*
2. *if $b_n = (\mathtt{log}, s_0, (r_n, c))$ and r_n was sent to the c, then s_2 must have delivered b_n by time $\uparrow 2$.*

Proof: If s_2 becomes the primary, then by Lemma 6.2.3, s_2 must be correct.

Case 1: $\uparrow 2 = l\tau + 2\delta$ because s_2 did not deliver A_0^l. If s_0 had broadcasted b_i, $i = 0 \ldots n-1$, by time $l\tau$, then from Figure 6.7, both s_0 and s_1 must have fifo–broadcasted $(\mathtt{echo}, b_i, -, -)$ by time $l\tau + \delta$, and since at least one of s_0 and

s_1 is correct and s_2 is also correct, s_2 must have fifo–delivered at least one copy of $(\mathtt{echo}, b_i, -, -)$ by time $l\tau+2\delta$ and we are done by Lemma 6.2.1. Similarly, if r_n was sent to the client, then both s_0 and s_1 must have fifo-broadcasted $(\mathtt{echo}, b_n, -, -)$, and s_2 must have delivered this.

On the other hand, if s_0 broadcasted b_i, $i = 0 \ldots n-1$ after time $l\tau$, then s_0 must have fifo–broadcasted $(\mathtt{init}, A_0^l, s_0, l\tau)$ earlier, and since s_1 fifo–delivered $(\mathtt{init}, b_i, s_0, -)$ it must have fifo–delivered $(\mathtt{init}, A_0^l, s_0, l\tau)$ (by Lemma 5.3.2). Thus, both s_0 and s_1 must have fifo–broadcasted $(\mathtt{echo}, A_0^l, -, l\tau)$, which implies that s_2 must have delivered A_0^l, a contradiction. Similarly for b_n.

Case 2: s_2 delivered $(\mathtt{takeover}, s_0, -)$ at time $\uparrow 2$ and s_2 fifo-delivered $(\mathtt{echo}, (\mathtt{takeover}, s_0, -), s_j, -)$ from s_j. Since s_0 fifo-broadcasted $(\mathtt{init}, b_i, s_0, -)$, $i = 0 \ldots n$, before $(\mathtt{init}, (\mathtt{takeover}, s_0, -), s_0, -)$, by Lemma 5.3.2, s_j must have fifo-broadcasted $(\mathtt{echo}, b_i, s_j, -)$ before $(\mathtt{echo}, (\mathtt{takeover}, s_0, -), s_j, -)$ and s_2 must have fifo-delivered them in the same order (again by Lemma 5.3.2). □

Lemma 6.2.6 *Let s_2 become the primary at time $\uparrow 2$ and $Prmy_{s_0} = false$ for the first time at $\downarrow 0 \leq \uparrow 2$. Then*

1. *$Init_2 \sqsubseteq Final_0$*
2. *If $Final_0 = M_0$ and $R_0 \neq \epsilon$, where $R_0 \subset M_0$ and $|M_0| - |R_0| = 2$, then R_0 was sent to the client.*
3. *If $Init_2 \sqsubset Final_0$, then $last(Final_0)$ was not sent to the client.*

Proof: Let $Final_0 = r_0 \circ r_1 \ldots \circ r_n$. If s_0 broadcasted $(\mathtt{log}, s_0, (r_n, -))$ then it must broadcasted $(\mathtt{log}, s_0, (r_i, -))$, $i = 0 \ldots n$, and therefore from Lemma 6.2.5

$(\texttt{log}, s_0, (r_i, -))$, $i = 0 \ldots n-1$, must have been delivered at s_2 and (1) is satisfied. If $(\texttt{log}, s_0, (r_n, -))$ was not delivered at s_2, then r_n was not sent to the client (Lemma 6.2.5) and (3) is satisfied.

If s_0 does not broadcast $(\texttt{log}, s_0, (r_n, -))$, then it must have halted. Therefore, from Lemma 6.2.2, s_0 is faulty and so s_1 is correct ($f = 1$). s_0 must have broadcasted $(\texttt{log}, s_0, (r_i, -))$, $i = 0 \ldots n-1$ and since s_1 is correct, r_{n-1} must have been sent to the client (from Figures 6.6 and 6.7). (1) now follows from Lemma 6.2.5. (3) is trivially true because s_0 does not broadcast $(\texttt{log}, s_0, (r_n, -))$ and so r_n could not have been sent to the client.

(2) can be seen to be true from Figure 6.6 because s_0 must have fifo-delivered $(\texttt{echo}, (\texttt{log}, s_0, r_i), s_1, -)$, $i = 0 \ldots n-1$ and so s_1 must have sent r_i, $i = 0 \ldots n-2$, to the client. □

Let $\rho = n_c(2\delta) + \delta$.

Lemma 6.2.7 *If s_0 remains the primary through time t, then there can be at most one request from any client i in the queue of s_0 at time t.*

Proof: Assume, for contradiction, that $t_2 \leq t$ is the first time that there are two requests (say m_1 and m_2) from some client i in the queue of s_0. Assume m_1 was sent at time t_1 and so it was enqueued by $t_1 + \delta$. From Figure 5.2, $2\rho = 2(n_c 2\delta + \delta)$ and so client i cannot send m_2 before time $t_1 + 2(n_c 2\delta + \delta)$ *i.e.* $t \geq t_2 \geq t_1 + 2(n_c 2\delta + \delta)$. Since m_2 is the first duplicate request, there are at most $n_c - 1$ requests ahead of m_1 in s_0's queue Therefore, from Figures 6.6 and 6.7 , s_0 will dequeue m_1 by time $t_1 + \delta + (n_c - 1)2\delta < t$. Therefore, $t_2 \leq t_1 + \delta + (n_c - 1)2\delta$ which is a contradiction. □

Lemma 6.2.8 *If a request m is sent by the client at time t_m to server s_j, and s_j remains the primary through the interval $[t_m..t_m + \rho]$, then m is received by s_j by time $t_m + \rho$.*

Proof: Since the message delivery time is at most δ, m is enqueued at s_j by time $t_m + \delta$. If $j = 0$, then from Lemma 6.2.7, there can be at most $n_c - 1$ requests ahead of m in s_j's queue. Therefore, from Figures 6.6 and 6.7, m is received by time $t_m + (n_c - 1)2\delta + \delta \leq t_m + \rho$.

On the other hand, if $j = 2$, then from Figure 6.2, m is received by time $t_m + \delta \leq t_m + \rho$. □

Lemma 6.2.9 *Let $Prmy_{s_0}$= $false$ for the first time at $\downarrow 0$ and $Final_0 = r_0 \circ r_1 \ldots r_n$. Let m_n be the request corresponding to r_n. If r_n was not sent to the client, then m_n must have been sent in the interval $[max(0, \downarrow 0 - 2\delta - \rho)..\downarrow 0]$ and received in the interval $[max(0, \downarrow 0 - 2\delta)..\downarrow 0]$.*

Proof: Suppose for contradiction that m_n was sent in the interval $[0..\downarrow 0 - 2\delta - \rho)$. From Lemma 6.2.8, m_n must have been received by time $t < \downarrow 0 - 2\delta$ by s_0, and since $Prmy_{s_0}$= $true$ at time $t + 2\delta < \downarrow 0$, s_0 must have received $(\texttt{echo}, (\texttt{log}, s_0, r_n), s_1, -)$ and $status = OK$. Since r_n was not sent to the client, s_1 must have halted by time $t + \delta < \downarrow 0$. Therefore, form Lemma 6.2.2, s_1 is faulty and so s_0 is correct and can never halt. Since s_0 makes no broadcasts after $(\texttt{log}, s_0, (r_n, -))$, the only way that $Prmy_{s_0}$= $false$ at time $\downarrow 0$ is that s_0 halts at time $\downarrow 0$, a contradiction.

Now if m_n was received at some time $t < \downarrow 0 - 2\delta$, then this is a contradiction as shown in the previous paragraph. Therefore, m_n can only be received in the interval $[max(0, \downarrow 0 - 2\delta)..\downarrow 0]$. □

Lemma 6.2.10 *For any run σ_{PB} of the primary–backup system there exists a run σ_P of the $(\rho, 2, \tau + 7\delta + \rho)$ bofo server P such that $\forall j : \sigma_P \approx_j \sigma_{PB}$.*

Proof: **Case 1:** $Prmy_{s_0}$ is always true in σ_{PB}. If s_0 makes infinitely many broadcasts in σ_{PB}, then from Figure 6.6, all requests have responses. Therefore, assume that all the requests m that s_1 responds to are responded to in σ_P as well. m is received at the same times in σ_{PB} and σ_P. However, m is enqueued at the same time it is received in σ_P (this has to be true of all runs of P). We can do this from Lemma 6.2.8. Furthermore, in σ_P the response r has to be sent at the same time the request m is received, whereas in σ_{PB}, r might be sent after δ time units by s_1. However, r has to be received at the same times by the clients in σ_{PB} and σ_P. We can do this because r is received by the client within 2δ after m is received, and $\rho \geq 2\delta$. However, if s_0 makes only finitely many broadcasts, then again from Figure 6.6, the last request might not have a response. Assume that the last request m_n is received by s_0 at time t. Let the first (and only) outage interval of σ_P be $[t..t]$ and assume in σ_P that the request m is received, but a response never sent.

Case 2: $Prmy_{s_0}$ is false for the first time at time $\downarrow 0$ in σ_{PB}. We first construct the run σ_P till time $\uparrow 2$, and then show how to extend it.

By Lemma 6.2.4, s_2 becomes the primary at time $\uparrow 2 \leq \downarrow 0 + \tau + 4\delta$. As in case 1, all requests sent in the interval $[0..\downarrow 0]$ that have responses by s_1 in σ_{PB} are responded to in σ_P as well. Let $Final_0 = r_0 \circ r_1 \ldots r_{n-1} \circ r_n$. By Lemma 6.2.6, r_{n-1} and r_n might not have been sent to the client. Let m_{n-1} and m_n be the requests corresponding to r_{n-1} and r_n respectively and these requests were received at times t_{n-1} and t_n respectively by s_0. Let the two outage intervals of σ_P be $I_1 = [t_{n-1}..t_{n-1}]$ and $I_2 = [max(0, \downarrow 0 - 2\delta)..\uparrow 2]$. If r_{n-1} was not sent to the client, assume that in σ_P, m_{n-1} was received, but a response never sent. This outage belongs to I_1.

From Lemma 6.2.6, $Init_2 = R = r_0 \circ r_1 \ldots r_{n-1}$ or $Init_2 = M = r_0 \circ r_1 \ldots r_{n-1} \circ r_n$. If $Init_2 = R$, then from Lemma 6.2.6, r_n was not sent to the client and from Lemma 6.2.9, m_n was sent in the interval $[max(0, \downarrow 0 - 2\delta - \rho)..\downarrow 0]$. We now show that $Init_2 = state_P(\uparrow 2)$. From Lemma 6.2.6, $r_0 \circ r_1 \ldots r_{n-2}$ must have been sent to the client and from construction, these responses are sent in σ_P as well. Furthermore, as argued before, m_{n-1} is always received in σ_P. Therefore, we are done if we can show that P receives no further requests until time $\uparrow 2$. Assume in σ_P that m_n is never enqueued (outage belongs to I_2) or hasn't yet been enqueued, depending on the time m_n is actually sent. Futhermore, all other requests that were sent in the interval $[0..\uparrow 2]$ that haven't been received in $\sigma_{PB}[\uparrow 2]$ must have been sent in the interval $[\downarrow 0 - \rho..\uparrow 2]$ (Lemma 6.2.8). Assume, therefore, that in σ_P these requests are never enqueued (outage belongs to I_2) or haven't yet been enqueued in σ_P. Therefore P receives no further requests and $Init_2 = state_P(\uparrow 2)$.

On the other hand, if $Init_2 = M$ and r_n is sent to the client, we can similarly show that the states are equal. On the other hand, if r_n is not sent to the client, assume that in σ_P, m_n is received at time t where $\downarrow 0 - 2\delta \leq t \leq \downarrow 0$ (from Lemma 6.2.9) but a response never sent (outage belongs to I_2).

We now extend σ_P. Extend I_2 as $I_2 = [max(0, \downarrow 0 - 2\delta)..\uparrow 2 + \delta + \rho]$. Therefore, $|I_2| \leq \tau + 7\delta + \rho$ (from Lemma 6.2.4). We first show that $\forall j : queue_{c_j,P}(\uparrow 2) = \phi$.

By the above construction of σ_P, all requests sent in $[0..\uparrow 2]$ other than m_n, that are received in $\sigma_{PB}[\uparrow 2]$ are received in σ_P as well. Also m_n is not enqueued at time $\uparrow 2$. Finally, all requests sent in $[0..\uparrow 2]$ that are not received in $\sigma_{PB}[\uparrow 2]$ must must have been sent in $[\downarrow 0 - \rho..\uparrow 2]$ (by Lemma 6.2.8). Assume that these are never enqueued in σ_P (again outages lies in I_2) or haven't yet been enqueued. Thus, $\forall j : queue_{c_j,P}(\uparrow 2) = \phi$. Furthermore, requests sent before $\uparrow 2$ to s_0 will never be enqueued after $\uparrow 2$ in σ_P

It is now easy to extend σ_P. Assume now that all requests that clients send in $[\uparrow 2..\uparrow 2+\delta]$ to s_0 are never enqueued in σ_P. This outage again lies in I_2. However, any respose that s_2 sends in σ_{PB} after $\uparrow 2$ is sent in σ_P as well. We can do this since $Init_2 = state_P(\uparrow 2)$, $queue_{c_j,P}(\uparrow 2) = \phi = queue_{c_j,s_0}(\uparrow 2)$ and no request sent to s_0 is enqueued after $\downarrow 0$ in σ_P . Since, by Lemma 6.2.3, s_2 is correct and will never halt, we are done. □

Theorem 39 *The protocol in Figures 6.5,6.6, 6.7 and 6.8 satisfies* Pb4.

Proof: Follows from Lemma 6.2.10. □

Theorem 40 *The protocol in Figures 6.5,6.6, 6.7 and 6.8 satisfies* Pb5.

Proof: If s_0 is correct, then by Lemma 6.2.2, it never halts. Therefore, s_0 can become the backup only if it does not receive a $(\texttt{echo}, M, s_1, -)$, which implies that s_1 must have failed.

On the other hand, if s_2 becomes the primary and is correct, then again by Lemma 6.2.2, it cannot halt and continues to be the primary. □

Theorem 41 *The protocol in Figures 6.5,6.6, 6.7 and 6.8 is δ-blocking.*

Proof: Follows from the description of the protocol. □

6.3 Receive–Omission failures

Finally, in this section, we give the δ–blocking protocol tolerating receive–omission failures. The protocol is shown in Figures 6.9, 6.10, 6.11 and 6.12, where $f = 1$ and $n_s = 2f = 2$. The procedures **initialize**, **deliver** and **delivery–process** are the same as those for send–omission failures. This protocol works for all values of D and Γ and, as required by Corollary 6.0.3, is "pass-the-buck". Furthermore, as required by Theorem 10, this protocol has a run in which a correct primary cedes to a faulty backup. We assume for this protocol that $\tau > 4\delta$.

initialize(i)
cobegin
  || **if** $i = 0$ **then primary**() **else backup**($i, 0$)
  || **delivery–process**(i)
coend

Figure 6.9: Protocol run by server s_i for receive–omission failures($n_s = 2, f = 1$)

There are two servers—s_0 and s_1. Initially, s_0 executes the procedure **primary** and s_1 executes the procedure **backup**.

In the procedure **primary**, the time interval τ is divided into two disjoint intervals (however, it turns out that we do not require the intervals to be completely disjoint and can use $\tau > 2\delta$). In the first interval, the primary sends a `alive`

```
procedure primary()
  cobegin
    || inform-clients("Dest = s_0")
    || pstatus := OK
      Isprimary[0, 1] := true
      /* the alive message is being sent now */
      for l := 0 to ∞ do
        when current-time = lτ: fifo-send(init,A^l_0, s_0, lτ)

        /* check if failure has occured */
        when current-time = lτ + 2δ:
          if not fifo-delivered (echo, A^l_0, s_1, lτ) then
            detected-failure(0, l)

        /* now process messages from the client */
        time := current-time
        while (lτ + 2δ < time < (l + 1)τ − 2δ) and pstatus = OK
          if received request from client c then
            response := Π(state_0, request)
            state_0 := state_0 ∘ response
            pstatus :=broadcast ((log, s_0, (green, response, c)), 0)
            time := current-time
    coend

  procedure detected-failure(k, i)
    Isprimary[0, 1] := false
    if delivered (takeover,s_1, iτ + δ) then backup(k, iτ + 2δ)
    else failure-mode-primary(k, (i + 1)τ)
```

Figure 6.10: Procedures for receive–omission failures($n_s = 2, f = 1$)-i

```
procedure backup(k, switch)
  j := k̄
  bstatus := OK
  cobegin
    || while bstatus = OK
         ((tag, −, (color, r, c)), −) :=Deq(Rqueue_k)
         /* synchronizing with new primary */
         if tag = mylastlog then
           if r ∈ state_k then
             if r = last(state_k) then skip
             else state_k := state_k \ last(state_k)
           else state_k := state_k ∘ r

         /* logging response from primary */
         if tag = log then
           state_k := state_k ∘ r
           if color = green then send r to client c

    || /* detecting primary failures */
      for l := 0 to ∞ do
        when current-time = lτ + switch + δ
          if not delivered A_j^{lτ+switch} then
            bstatus := NOTOK
            time := current-time
            fifo-send (init,(takeover,s_k, time), s_k, time)
            wait until current-time = time + δ
            failure-mode-primary(k, time + δ)
  coend
```

Figure 6.11: Procedures for receive–omission failures($n_s = 2, f = 1$)-ii

```
procedure failure–mode–primary(k,first)
  start := current-time
  cobegin
    || inform–clients("Dest = s_k")
    || Isprimary[k, 2] := true
       R := (red, last(state_k), −)
       fifo–send(init, (mylastlog, s_k, R), s_k, current-time)
       do forever
         when received request from client c sent after start
           response := Π(state_k, request)
           state_k := state_k ∘ response
           R := (red, response, c)
           broadcast (init, (log, s_k, R), s_k, current-time)
           send response to client c
       od
    || for l := 0 to ∞ do
         when current-time = first + lτ:
           fifo-send (init, A_k^{first+lτ}, s_k, first + lτ)
  coend

procedure broadcast(M, k, type)
  j := k̄
  time := current-time
  fifo-send(init, M, s_k, time)
  if by time + 2δ not fifo-delivered (echo, M, s_j, time) then
    return(NOTOK)
  else return(OK)
```

Figure 6.12: Procedures for receive–omission failures($n_s = 2, f = 1$)-iii

message and waits for the corresponding acknowledgement to arrive. If the acknowledgement arrives, the primary processes requests from the clients in the second interval.

In a failure-free run of this protocol, since the backup responds to the client, the primary forwards any response to the backup (with a `green` tag as shown in the protocol) and the backup sends this response to the client. However, if there is a failure, then the primary responds to the clients. In this case, the primary will forward a response to the backup with a `red` tag. The backup does not forward a response to the client if the response has a `red` tag.

Whenever s_0 receives a request from the client, it computes a response r, changes state, and sends r to s_1 with a `green` tag. Upon receiving this message, s_1 updates its state, acknowledges to s_0, and then sends r to the client. Because it is the backup that responds to the client, the protocol is δ–blocking. Server s_0 processes a new request only after receiving the acknowledgement from s_1 for the previous request.

Suppose that s_0 does not get s_1's acknowledgement for some response r, and therefore **broadcast** returns a status of $NOTOK$ (the argument is similar if no acknowledgement is received for an `alive` message). There are three possibilities: (1) s_1 has crashed, (2) s_1 omitted to receive the response r and so did not send the acknowledgement, (3) s_0 omitted to receive the acknowledgement. s_0 now waits until it is supposed to send the next `alive` message. s_0 sends this `alive` message and waits for an acknowledgement. We now consider the above three cases separately.

Case 1: s_1 has crashed. As a result, s_0 will not receive the acknowledgement to the `alive` message and, therefore, will call procedure **detected–failure**. Furthermore, since s_1 has crashed, s_0 will not deliver the `takeover` message either, and so s_0 again becomes the primary by calling procedure **failure–mode–primary**. From now on, whenever s_0 receives a request from a client, it computes the response r, sends r to s_1 with a `red` tag, and then sends the response r back to the client. Also, s_0 continues to send `alive` messages. Since s_0 is correct, it can continue like this forever.

Case 2: s_1 is faulty and omitted to receive the response r. By the property of **fifo–send** and **fifo–deliver** in Lemma 5.3.2, s_1 will not receive the `alive` message that s_0 sends, and so does not send the corresponding acknowledgement. Server s_1 concludes that s_0 has crashed, sends a `takeover` message to s_0 and becomes the primary by calling **failure–mode–primary**. After that, it behaves like s_0 in case 1 above (including sending `alive` messages to s_0). Since s_0 will not receive the acknowledgement (none was sent), it again calls procedure **detected–failure**. However, since s_0 is correct, it receives the `takeover` message (as opposed to case 1) and so it becomes the backup. Also, since s_0 is correct, it will not omit to receive the response messages that s_1 sends and so s_0 keeps its state consistent with s_1. Subsequently, if s_0 stops receiving `alive` messages from s_1, then s_1 has crashed and s_0 becomes the primary once again by calling **failure–mode–primary**.

Case 3: s_0 is faulty. Since s_1 is correct, it receives the `alive` message from s_0, sends the corresponding acknowledgement and remains the backup (as opposed to case 2). However, by the property of **fifo–broadcast** and **fifo–receive** in Lemma 5.3.2, s_0 will not receive this acknowledgement to the `alive` message, and so again calls **detected–failure**. It will not receive the `takeover` message either (none was sent), and so it behaves as in case 1, and again becomes the primary by calling **failure–mode–primary**. Similar to case 2, s_1 receives all response messages that s_0 sends and so its state is consistent with s_0. Finally, s_1 becomes the primary if it stops receiving `alive` messages from s_0.

Case 2 shows the run required by Theorem 10. In this run, the correct primary s_0 cedes to a faulty backup s_1.

6.3.1 Proof of Correctness

We now formally show that the protocol in Figures 6.9, 6.10, 6.11 and 6.12 satisfies Pb1—Pb5.

Lemma 6.3.1 *Let $j = \bar{k}$. If s_j fifo-sends m and s_k is correct, then s_k fifo-delivers m.*

Proof: Easy to see because the links are FIFO are there are no send–omission failures. □

Define:

$Prmy_{s_0} \triangleq ((pstatus = OK \land Isprimary[0,1]) \lor Isprimary[0,2]) \land s_0$ has not halted.

$Prmy_{s_1} \triangleq Isprimary[1,2] \land s_1$ has not halted.

Lemma 6.3.2 *Let $j = \bar{k}$. If s_j calls* **failure–mode–primary**, *then either s_j is faulty (by receive–omission) or s_k has halted.*

Proof: **Case 1:** s_j calls **failure–mode–primary** in **detected–failure** in Figure 6.10. This has to be the first time s_0 does not get $b = (\texttt{echo}, (\texttt{alive}, s_0, l\tau), s_1, l\tau)$ at time $l\tau + 2\delta$ in Figure 6.10. If b was sent, then by Lemma 6.3.1 s_0 is faulty and we are done. So assume that b was not sent. If s_1 halted, then again we are done. If s_1 did not halt and did not send b, then again by Lemma 6.3.1 s_1 is faulty. As can been from Figure 6.11, s_1 will send $(\texttt{init}, (\texttt{takeover}, s_1, time), s_1, time)$ at time $l\tau + \delta$. Since s_0 is not faulty ($f = 1$), it will receive this message by time $l\tau + 2\delta$ (Lemma 6.3.1), and from Figure 6.10, will not call **failure–mode–primary**, a contradiction.

Case 2: s_j calls **failure–mode–primary** in **backup** in Figure 6.11. First assume that $j = 0$. s_0 became the backup at time $switch = i\tau + 2\delta$ for some $i \geq 0$ after receiving the $(\texttt{init}, (\texttt{takeover}, s_1, time), s_1, time)$ message that s_1 sent at $i\tau + \delta$. Suppose s_0 called **failure–mode–primary** at time $t = l\tau + (i\tau + 2\delta) + 2\delta$ as it did not deliver $(\texttt{alive}, s_1, l\tau + i\tau + 2\delta)$ from s_1. From Figure 6.11, s_1 should have called **failure–mode–primary** at time $i\tau + 2\delta$, and should have sent $(\texttt{alive}, s_1, l\tau + i\tau + 2\delta)$ for all $l \geq 0$. Therefore, if s_0 does not deliver this, then either s_0 is faulty, or s_1 has halted.

Assume now that $j = 1$. s_1 became the backup at time $switch = 0$. Suppose s_1 called **failure–mode–primary** at time $t = l\tau + 2\delta$ because it did not deliver $(\texttt{alive}, s_1, l\tau)$ from s_0, which again implies that either s_0 has halted, or s_1 is faulty. □

The following corollary now follows from Lemma 6.3.2:

Corollary 6.3.1 *If $Isprimary[0,2]$ and s_0 has not halted, then $\neg Isprimary[1,2]$ or s_1 has halted.*

Proof: For contradiction, assume that $Isprimary[0,2]$, $Isprimary[1,2]$ and neither s_0 or s_1 have halted. However, this then implies from Lemma 6.3.2 that both s_0 and s_1 are faulty, a contradiction since $f = 1$. □

Theorem 42 *The protocol in Figures 6.9, 6.10, 6.11 and 6.12 satisfies* Pb1.

Proof: From Corollary 6.3.1, if $Isprimary[0,2]$ and s_0 has not halted, then either $\neg Isprimary[1,2]$ or s_1 has halted. Therefore, the only way Pb1 can be violated if $Isprimary[0,1] \wedge Isprimary[1,2]$ and neither servers have halted. Suppose they become true for the first time at t. s_1 would have called **failure–mode–primary** at time $t' = l\tau + 2\delta$ for some $t' \leq t$ after it failed to deliver $(\texttt{alive}, s_0, l\tau)$. Consequently, s_0 will not fifo-deliver $(\texttt{echo}, (\texttt{alive}, s_0, l\tau), s_1, l\tau)$, and so set $Isprimary[0,1]$ to $false$, at time $t' = l\tau + 2\delta$, which implies that $Isprimary[0,1] = false$ at time $t \geq t'$, a contradiction. □

Theorem 43 *The protocol in Figures 6.9, 6.10, 6.11 and 6.12 satisfies* Pb2.

Proof: Trivially true. □

Since s_0 can become the backup and then become the primary again, in order to satisfy Pb3, we need some mechanism such that no requests are enqueued at s_0 while it is the backup. So we make the assumption that such a mechanism exists and discards all requests that arrive at a backup. Pb3 now holds.

Theorem 44 *The protocol in Figures 6.9, 6.10, 6.11 and 6.12 satisfies* Pb3.

Lemma 6.3.3 *Let* $j = \bar{k}$. *If* $\exists t$ *such that* $Isprimary[j, 2]$ *for the first time at* t *and* $\neg\exists t' < t$ *such that* $Isprimary[k, 2]$ *at* t', *then* $Isprimary[k, 2]$ *cannot become true before time* $t + \delta$.

Proof: If s_k has halted before t, then we are done. So assume that this is not the case.

Case 1: s_j calls **failure–mode–primary** in **detected–failure**. This has to be the first time s_0 does not get $b = (\texttt{echo}, (\texttt{alive}, s_0, l\tau), s_1, l\tau)$ at time $t = l\tau + 2\delta$. If b was sent, then s_1 cannot set $Isprimary[1, 2]$ before time $(l + 1)\tau + 2\delta > t + \delta$ as $\tau > 4\delta$. If b was not sent, then since s_1 has not halted by time $l\tau + \delta$, it is faulty by Lemma 6.3.1 and will send $(\texttt{init},(\texttt{takeover},s_1, time), s_1, time)$ at time $l\tau + \delta$. Since s_0 is not faulty ($f = 1$), it will receive this message by time $l\tau + 2\delta$ (Lemma 6.3.1), and from Figure 6.10, will not call **failure–mode–primary**, a contradiction.

Case 2: s_j calls **failure–mode–primary** in **backup**. First assume that $j = 0$. s_0 became the backup at time $switch = i\tau + 2\delta$ after receiving the $(\texttt{init},(\texttt{takeover},s_1, time), s_1, time)$ message that s_1 sent at $i\tau + \delta$. Suppose s_0 called **failure–mode–primary** at time $t = l\tau + (i\tau + 2\delta) + 2\delta$ as it did not deliver $(\texttt{alive}, s_1, l\tau + i\tau + 2\delta)$ from s_1. Since $Isprimary[1, 2]$ is not true before t, s_1 must have halted by time $i\tau + 2\delta < t$, a contradiction since we have assumed that that s_1 has not halted before time t.

Now assume that $j = 1$. If s_1 calls **failure–mode–primary** at time $t = i\tau + 2\delta$, then s_0 cannot call **failure–mode–primary** before time $(i\tau + 2\delta) + 2\delta > t + \delta$ □

Let $k = \bar{j}$. Define:

$q[0] = s_0$

$q[1] = s_j$ if $\exists t$ such that $Isprimary[j, 2]$ at t and $\neg \exists t' < t$ such that $Isprimary[k, 2]$ at t'

$= \perp$ otherwise.

$q[2] = s_k$ if $q[1] = s_j$ and $\exists t$ such that $Isprimary[k, 2]$ at time t

$= \perp$ otherwise

From Lemma 6.3.3, $q[1]$ is well defined.

Lemma 6.3.4 *Let $\downarrow 0$ be the smallest of the times that $q[0]$ halts, or $pstatus_0 = NOTOK$ or $Isprimary[0, 1] = false$. Then $q[1] = s_k$ for $k \in \{0, 1\}$ and $Isprimary[k, 2]$ is true by time $\downarrow 0 + 2\tau$.*

Proof:

Case 1: s_0 halts at time $\downarrow 0$, where $(l-1)\tau \leq \downarrow 0 < l\tau$. Then s_1 will not deliver $(\texttt{alive}, s_0, l\tau)$ and become the primary by time $l\tau + 2\delta \leq \downarrow 0 + \tau + 2\delta < \downarrow 0 + 2\tau$ since $\tau > 4\delta$.

Assume now that s_0 does not halt at time $\downarrow 0$.

Case 2: $pstatus_0 = NOTOK$ because s_0 did not fifo-deliver $(\texttt{echo}, M, s_1, -)$ for some M by time $\downarrow 0$, where $(l-1)\tau + 4\delta < \downarrow 0 < l\tau$. From the property of fifo-send and fifo-deliver, s_0 will not fifo-deliver $(\texttt{echo}, (\texttt{alive}, s_0, l\tau), s_1, l\tau)$. Now if s_1 did not fifo-send $(\texttt{init}, (\texttt{takeover}, s_1, l\tau + \delta), s_1, -)$, then either $Isprimary[0, 2]$ becomes true by time $l\tau + 2\delta < \downarrow 0 + \tau - 2\delta < \downarrow 0 + 2\tau$, or s_0 halts by time $l\tau + 2\delta$ and $Isprimary[1, 2]$ becomes true by time $(l+1)\tau + 2\delta < \downarrow 0 + 2\tau - 2\delta < \downarrow 0 + 2\tau$.

On the other hand, if s_1 did fifo-send $(\mathtt{init}, (\mathtt{takeover}, s_1, l\tau + \delta), s_1, -)$, then either $Isprimary[1,2]$ becomes true by time $l\tau + 2\delta < \downarrow 0 + \tau - 2\delta < \downarrow 0 + 2\tau$, or s_1 halts by time $l\tau + 2\delta$ and $Isprimary[0,2]$ becomes true by time $\tau + (l\tau + 2\delta) + 2\delta < \downarrow 0 + 2\tau$.

Case 3: $Isprimary[0,1] = false$ at time $\downarrow 0 = l\tau + 2\delta$ because s_0 did not fifo-deliver $(\mathtt{echo}, (\mathtt{alive}, s_0, l\tau), s_1, l\tau)$. The rest of the argument is now similar to case 2. □

Lemma 6.3.5 *Suppose $q[1] = s_1$ and s_1 become the primary at time $\uparrow 1 = l\tau + 2\delta$. If s_1 halts at time $\downarrow 1$, then*

1. *s_0 would have called* **backup** *at time $\uparrow 1$ and*

2. *s_0 cannot set $bstatus_0 = NOTOK$ before time $\downarrow 1 + \delta$.*

Proof: Since s_1 halts, it is faulty and so s_0 is correct. s_0 would not have fifo-delivered $(\mathtt{echo}, (\mathtt{alive}, s_0, l\tau), s_1, l\tau)$ and would have delivered $(\mathtt{takeover}, s_1, l\tau + \delta)$. Thus s_0 would have called **backup** at time $\uparrow 1 = l\tau + 2\delta$. Now let $i\tau + l\tau + 2\delta < \downarrow 1 \leq (i+1)\tau + l\tau + 2\delta$. s_0 would have delivered $(\mathtt{alive}, s_1, i'\tau + l\tau + 2\delta)$ for all $i' \leq i$, and so cannot set $bstatus_0 = NOTOK$ before time $(i+1)\tau + l\tau + 3\delta \geq \downarrow 1 + \delta$. □

Lemma 6.3.6 *Let $q[1] = s_j$. If s_j halts at time $\downarrow j$, then $q[2] = s_k$, $k = \bar{j}$, and $Isprimary[k, 2] = true$ by time $\downarrow j + \tau + 2\delta$.*

Proof: **Case 1:** $j = 0$ and let $(l-1)\tau \leq \downarrow j < l\tau$. Then s_1 cannot halt $(f = 1)$ and will not fifo-deliver $(\mathtt{alive}, s_0, l\tau)$ and so become the primary by time $l\tau + 2\delta \leq \downarrow j + \tau + 2\delta$.

Case 2: $j = 1$ and s_1 called **failure–mode–primary** at time $l\tau + 2\delta$. From Lemma 6.3.5, s_0 would have called **backup** by time $l\tau + 2\delta$. Now let $i\tau + l\tau + 2\delta \leq \downarrow j < (i+1)\tau + l\tau + 2\delta$. Then s_0 will not deliver $(\texttt{alive}, s_1, (i+1)\tau + l\tau + 2\delta)$ and become the primary by time $(i+1)\tau + l\tau + 4\delta < \downarrow j + \tau + 2\delta$. □

Let $k = \lceil \frac{n_c}{\lceil \frac{\tau - 4\delta}{2\delta} \rceil} \rceil$, and $\rho = 4k\delta + 2n_c\delta + \delta$.

Lemma 6.3.7 *If $Isprimary[0,1] = true$ and $pstatus_0 = OK$ and s_0 has not halted through time t, then there can be at most one request from any client i in the queue of s_0 at time t.*

Proof: Assume, for contradiction, that $t_2 \leq t$ is the first time that there are two requests (say m_1 and m_2) from some client i in the queue of s_0. Assume m_1 was sent at time t_1 and so it was enqueued by $t_1 + \delta$. Furthermore, since m_2 is the first duplicate request, there are at most $n_c - 1$ requests ahead of m_1 in s_0's queue. From Figure 5.2, $2\rho = 2(n_c 2\delta + \delta + 4k\delta)$ and so client i cannot send m_2 before time $t_1 + 2(n_c 2\delta + \delta + 4k\delta)$ *i.e.* $t \geq t_2 \geq t_1 + 2(n_c 2\delta + \delta + 4k\delta)$. From Figures 6.10 and 6.12, s_0 will dequeue m_1 by time $t_1 + \delta + (n_c - 1)2\delta + 4k\delta < t$. (The last term comes about because in procedure **primary** only $\tau - 4\delta$ every τ is used for processing requests.) Therefore, $t_2 \leq t_1 + \delta + (n_c - 1)2\delta + 4k\delta$, which is a contradiction. □

Lemma 6.3.8 *If a request m is sent by the client at time t_m to server s_0, and $Isprimary[0,1] = true$ and $pstatus_0 = OK$ through the interval $[t_m..t_m + \rho]$, then if s_0 has not halted by time $t_m + \rho$ then m is received by s_0 by time $t_m + \rho$.*

Proof: Since the message delivery time is at most δ, m is enqueued at s_0 by time $t_m + \delta$. From Lemma 6.3.7, there can be at most $n_c - 1$ requests ahead of m in s_0's queue. Therefore, from Figures 6.10 and 6.12, m is received by time $t_m + \delta(n_c - 1)2\delta + 4k\delta < t_m + \rho$, and the lemma holds. □

Lemma 6.3.9 *If $Isprimary[j,2]$ for the first time at t, then there is a time $\downarrow 0 \leq t$ such that s_0 halts at $\downarrow 0$, or $pstatus_0 = NOTOK$ at time $\downarrow 0$ or $Isprimary[0,1] = false$ at time $\downarrow 0$.*

Proof: Trivially true if $j = 0$ because s_0 sets $Isprimary[0,1] = false$ before calling **failure–mode–primary**. Now assume that $j = 1$. If $q[1] = s_1$ and s_1 calls **failure–mode–primary** a time $t = l\tau + 2\delta$, then s_0 could not have fifo-delivered $(\texttt{echo}, (\texttt{alive}, s_0, l\tau), s_1, l\tau)$ and would have either halted or set $Isprimary[0,1] = false$ by time t. Similarly, if $q[2] = s_1$, then $q[1] = s_0$ and s_0 would have set $Isprimary[0,1] = false$ by time t. □

Lemma 6.3.10 *Let $q[1] = s_1$ and let $\downarrow 0$ be the smallest of the times that s_0 halts or $pstatus_0 = NOTOK$ or $Isprimary[0,1] = false$ (this is defined by Lemma 6.3.9). Let $Final_0 = r_0 \ldots r_n$ and let m_n be the request corresponding to r_n. If r_n was not sent to the client, then m_n must have been received in the interval $[max(0, \downarrow 0 - 2\delta)..\downarrow 0]$ and sent in the interval $[max(0, \downarrow 0 - \rho - 2\delta)..\downarrow 0]$.*

Proof: Suppose m_n was sent before time $\downarrow 0 - \rho - 2\delta$. Then by Lemma 6.3.8, it must be received at time $t' < \downarrow 0 - 2\delta$. Let $l\tau + 2\delta < t' < (l+1)\tau - 2\delta$. From Figures 6.10 and 6.11, s_1 can set $bstatus_1 = NOTOK$ earliest by time $(l+1)\tau + \delta$ because it would have delivered $(\texttt{alive}, s_0, l'\tau)$ for all $l' \leq l$.

If s_1 fifo-sent $(\texttt{echo}, (\texttt{log}, s_0, r_n), s_1, -)$ by time $t' + \delta < (l+1)\tau + \delta$, then s_1 would have sent r_n to the client, a contradiction.

If s_1 did not fifo-send $(\texttt{echo}, (\texttt{log}, s_0, r_n), s_1, -)$, then s_0 would have set $pstatus_0$ $= NOTOK$ by time $t' + 2\delta < \downarrow 0$, again a contradiction.

Similarly,if we assume that m_n was received at time $t' < \downarrow 0 - 2\delta$, then this is a contradiction as proved in the previous paragraph. $\square$

Lemma 6.3.11 *Let $q[1] = s_0$ and let $\downarrow 0$ be the smaller of the times that $pstatus_0 =$ $NOTOK$ or $Isprimary[0,1] = false$ (this is defined by Lemma 6.3.9). Let $Final_0 =$ $r_0 \ldots r_n$ and let m_n be the request corresponding to r_n. If r_n was not sent to the client, then m_n must have been received in the interval $[max(0, \downarrow 0 - \tau)..\downarrow 0]$ and sent in the interval $[max(0, \downarrow 0 - \tau - \rho)..\downarrow 0]$.*

Proof:

Suppose m_n was sent before time $\downarrow 0 - \tau - \rho$. Then it must have been received at time $t' < \downarrow 0 - \tau$. Let $l\tau + 2\delta < t' < (l+1)\tau - 2\delta$.

Case 1: s_0 sets $pstatus_0 = NOTOK$ at time $\downarrow 0$ as it did not fifo-deliver $(\texttt{echo}, (\texttt{log}, s_0, (\texttt{green}, r_n, -)), s_1, -)$. If m_n was received at time $t' < \downarrow 0 - \tau$, then s_0 would have set $pstatus_0 = NOTOK$ by time $t' + 2\delta < \downarrow 0 - \tau + 2\delta < \downarrow 0$ because $\tau > 4\delta$, a contradiction. Similarly, if we assume that m_n was received at time $t' < \downarrow 0 - \tau$, then this is a contradiction as proved earlier.

Case 2: s_0 sets $Isprimary[0,1] = false$ at time $\downarrow 0$ because it did not fifo-deliver $(\texttt{echo}, (\texttt{alive}, s_0, l'\tau), s_1, l\tau)$. We show that $l' = l+1$. Since r_n was not sent to the client and s_0 fifo-delivered $(\texttt{echo}, (\texttt{log}, s_0, (\texttt{green}, r_n, -)), s_1, -)$, s_1 must have halted by time $t' + \delta < (l+1)\tau$. Therefore, s_0 cannot fifo-deliver

$(\texttt{echo}, (\texttt{alive}, s_0, (l+1)\tau), s_1, -)$ and so $l' = l + 1$. Therefore, $\downarrow 0 = (l+1)\tau + 2\delta$ and since $l\tau + 2\delta < t'$, $\downarrow 0 - \tau < t'$ and we are done. Similarly, if m_n was sent before $\downarrow 0 - \tau - \rho$, then it must have been received at time $t' < \downarrow 0 - \tau$, which is a contradiction because $\downarrow 0 - \tau < t'$. □

Lemma 6.3.12 *Let* $q[1] = s_k$ *call* **failure–mode–primary** *at time* $\uparrow k$ *and halt at time* $\downarrow k$. *Let* $Final_k \setminus Init_k = r_0 \ldots r_n$ *and let* m_n *be the request corresponding to* r_n. *If* r_n *was not sent to the client, then* m_n *must have been received at time* $\downarrow k$ *and sent in the interval* $[max(\uparrow k, \downarrow k - \rho)..\downarrow k]$.

Proof: Easy to see because s_k would have halted just after receiving m_n. □

Lemma 6.3.13 *Suppose* $q[1] = s_1$ *and* s_1 *becomes the primary at time* $\uparrow 1$. *Let* $\downarrow 0 \leq \uparrow 1$ *be the smallest time such that* s_0 *halts, or* $pstatus_0 = NOTOK$ *or* $Isprimary[0, 1] = false$ *(this is defined by Lemma 6.3.9). Then*

1. $Init_1 \sqsubseteq Final_0$
2. *If* $Final_0 = M_0$ *and* $R_0 \neq \epsilon$, *where* $R_0 \sqsubset M_0$, *then* R_0 *was sent to the client.*
3. *If* $Init_1 \sqsubset Final_0$, *then* $last(Final_0)$ *was not sent to the client.*

Proof: Let $\uparrow 1 = l\tau + 2\delta$ and let $Final_0 = r_0 \ldots r_n$. Furthermore from the definition of $q[1]$, s_0 could not have called **failure–mode–primary** before time $\uparrow 1$. Since $\downarrow 0 \leq l\tau + 2\delta$, s_0 would have sent sequence of broadcasts $(\texttt{log}, s_0, (\texttt{green}, r_i, -))$, $i = 0 \ldots n-1$ by time $l\tau - 2\delta$ and fifo-delivered $(\texttt{echo}, (\texttt{log}, s_0, (\texttt{green}, r_i, -)), s_1, -)$. Therefore, s_1 would have delivered $(\texttt{log}, s_0, (\texttt{green}, r_i, -))$ by time $l\tau - \delta$ and

(1) now follows from Lemma 6.1.1 and the fact that $bstatus_1 = NOTOK$ only at time $l\tau + \delta$. (2) is true as s_1 would have sent r_i, $i = 0 \ldots n-1$ to the client by time $l\tau - \delta$. (3) is trivially true because only s_1 can send r_n to the client. □

Lemma 6.3.14 *If $q[2] = s_1$, then $q[1] = s_0$ must have called* **failure–mode–primary** *in* **detected–failure**.

Proof: For contradiction, assume that s_0 called **failure–mode–primary** in **backup** at time $t = i\tau + (l\tau + 2\delta) + 2\delta$. Since $q[1] \neq s_1$, s_1 did not call **failure–mode–primary** at time $l\tau + 2\delta$ and so must have halted by time $l\tau + 2\delta < t$, a contradiction since s_1 calls **failure–mode–primary** after t. □

Lemma 6.3.15 *Let $q[2] = s_1$ and s_1 becomes the primary at time $\uparrow 1$. Let s_0 call* **failure–mode–primary** *at time $\uparrow 0$ and $\downarrow 0 \geq \uparrow 0$ be the greatest time s_0 was the primary. Then*

1. *$Init_1 \sqsubseteq Final_0$.*
2. *If $Final_0/Init_0 = M_0$ and $R_0 \neq \epsilon$, where $R_0 \sqsubset M_0$, then R_0 was sent to the client.*
3. *if $Init_1 \sqsubset Final_0$, then $last(Final_0)$ was not sent to the client.*
4. *If $Final_0 = Init_0$ then $Init_1 = Final_0$*

Proof: From Lemma 6.3.14, s_0 must have called **failure–mode–primary** in **detected–failure** and so $\uparrow 0 = l\tau + 2\delta$ for some $l \geq 0$ and $first = (l+1)\tau$. s_0 is faulty and must have halted at time $\downarrow 0$. Let $(i+l)\tau < \downarrow 0 \leq (i+l+1)\tau$. Since s_1

is correct, it would have delivered $(\mathtt{alive}, s_0, j'\tau)$ for all $j' \leq i+l$, and so cannot set $bstatus_1 = NOTOK$ before time $(i+l+1)\tau + \delta \geq \downarrow 0 + \delta$. Consequently, $\uparrow 1 \geq \downarrow 0 + 2\delta$.

Let $Init_0 = r_0 \ldots r_n$. s_0 must have sent sequence of broadcasts $(\mathtt{log}, s_0, (\mathtt{green}, r_i, -))$, $i = 0 \ldots n$ by time $l\tau - 2\delta$ and since s_1 is correct, it would have delivered these in the right order by time $l\tau + \delta$. Therefore $state_1(\uparrow 0) = r_0 \ldots r_n$. Furthermore, the delivery of (if ever delivered) $b = (\mathtt{mylastlog}, s_0, (-, r_n, -))$ will not change $state_1$. This satisfies (4). Now let $Final_0 = r_0 \ldots r_n \circ r_{n+1} \ldots r_m$.

Since s_0 would have broadcasted $(\mathtt{log}, s_0, (\mathtt{red}, r_i, -))$, $n+1 \leq i < m$ by time $\downarrow 0$ and since s_1 is correct and does not set $bstatus_1 = NOTOK$ before time $\downarrow 0 + \delta$, (1) follows. Also (2) follows from Figure 6.12.

(3) can also be seen to be true because if s_1 did not deliver $(\mathtt{log}, s_0, (\mathtt{red}, r_m, -))$, then s_0 could not have broadcasted $(\mathtt{log}, s_0, (\mathtt{red}, r_m, -))$, and consequently from Figure 6.12, could not have sent r_m to the client. □

Lemma 6.3.16 *Suppose* $q[2] = s_0$ *and* $q[1] = s_1$. *Let* s_1 *call* **failure–mode–primary** *at time* $\uparrow 1$ *and* $\downarrow 1$ *is the greatest time that* s_1 *is the primary. Let* $Isprimary[0,2] = true$ *for the first time at* $\uparrow 0$. *Then if* $\mathtt{sf}(s_1)$ *then*

1. $Init_0 \sqsubseteq Final_1$.

2. *If* $Final_1/Init_1 = M_1$ *and* $R_1 \neq \epsilon$, *where* $R_1 \sqsubset M_1$, *then* R_1 *was sent to the client.*

3. *If* $Init_0 \sqsubset Final_1$, *then* $last(Final_1)$ *was not sent to the client.*

4. *If* $Init_1 = Final_1$ *then* $Init_0 = Init_1$

Proof: Let $\uparrow 1 = l\tau + 2\delta$. From Lemma 6.3.5, s_0 would have called **backup** by time $\uparrow 1$ and cannot set $bstatus_0 = NOTOK$ before time $\downarrow 1 + \delta$.

From Lemma 6.3.13, $Init_1 \sqsubseteq state_0(\uparrow 1)$. By Lemma 6.3.2 s_1 is faulty and so s_0 is correct. By assumption, s_1 makes a successful broadcast of $b = (\mathtt{mylastlog}, s_1, (-, last(Init_1), -))$ and s_0 will deliver this by time $\downarrow 1 + \delta$, making the two states equal. This satisfies (4). Now let $Final_1 = Init_1 \circ r_0 \ldots r_n$. Therefore, from Figure 6.12, s_1 would have successfully broadcast $(\mathtt{log}, s_1, (\mathtt{red}, r_i, -))$, $i = 0 \ldots n-1$, by time $\downarrow 1$ and s_0 would have delivered these after b (since s_0 is correct) by time $\downarrow 1 + \delta$. This satisfies (1). (2) and (3) can be seen to be true from Figure 6.12. □

Lemma 6.3.17 *Suppose $q[i] = s_j$, where $i > 0$ and s_j calls* **failure–mode–primary** *at time $\uparrow j$. Furthermore, if $i = 2$, then $q[1]$ did not initiate a successful broadcast of the form $(\mathtt{mylastlog}, q[1], -)$. Let $\downarrow 0 \leq \uparrow j$ be the smaller of the times that s_0 halts, or $pstatus_0 = NOTOK$ or $Isprimary[0,1] = false$ (defined by Lemma 6.3.9). Then*

1. *$Init_j \sqsubseteq Final_0$.*
2. *If $Final_0 = M_0$ and if $R_0 \neq \epsilon$, where $R_0 \subset M_0$ and $|M_0| - |R_0| = 2$, then R_0 was sent to the client.*
3. *If $Init_j \sqsubset Final_0$, then $last(Final_0)$ was not sent to the client.*

Proof: Let $Final_0 = r_0 \ldots r_{n-2} \circ r_{n-1} \circ r_n$. (2) is true because $r_0 \ldots r_{n-2}$ must have been sent to the client by s_1. We now prove (1) and (3).

Case 1: $i = 1$. If $j = 1$, the result then follows from Lemma 6.3.13. On the other hand, if $j = 0$, the result trivially follows.

Case 2: $i = 2$. If $q[1] = s_1$, then s_1 can initiate only one broadcast of the form $(\texttt{mylastlog}, s_1, (-, r_k, -))$, where $k \in \{n-1, n\}$ (from Lemma 6.3.13). If $q[2]$ (*i.e.* s_0) ever delivers this, then (1) is satisfied (from Lemma 6.3.5 and Figure 6.11). The precondition of (3) can be true iff s_1 broadcasted $(\texttt{mylastlog}, s_1, (-, r_{n-1}, -))$ (as s_0 has to deliver this). This implies that s_1 did not deliver $(\texttt{log}, s_0, (-, r_n, -))$, and so r_n could not have been sent to the client.

If $q[1] = s_0$, then s_0 would have called **failure–mode-primary** at time $t' = l\tau + 2\delta$ in **detected–failure** (by Lemma 6.3.14) and so $first = (l+1)\tau$. Also, $state_0(t') = Final_0 = r_0 \dots r_n$. By Lemma 6.3.2 s_0 is faulty and must have halted at time t' because it did not initiate a successful broadcast at time t'. s_0 must have sent sequence of broadcasts $(\texttt{log}, s_0, (\texttt{green}, r_i, -))$, $i = 0 \dots n$, by time $l\tau - 2\delta$ and since s_1 is correct, it would have delivered these in the right order by time $l\tau + \delta$. Therefore $state_1(t') = r_0 \dots r_n$. Furthermore, the delivery of (if ever delivered) $(\texttt{mylastlog}, s_0, (-, r_n, -))$ will not change $state_1$ (from Figure 6.11) and (1) is satisfied. (3) is trivially true because $Init_j = Final_0$. □

Lemma 6.3.18 *Suppose* $q[i] = s_j$, *where* $i > 0$ *and* s_j *calls* **failure–mode–primary** *at time* $\uparrow j$. *Furthermore,* $q[i]$ *initiates a successful broadcast of the form* $(\texttt{mylastlog}, q[i], -)$ *and* $q[i]$ *is the* n^{th} *server,* $n \in \{1, 2\}$ *to initiate such a successful broadcast. Then*

1. *there is a run* σ_P *of a* $(\rho, n+1, 4\tau + 3\delta + \rho)$ *bofo server* P *such that* $\forall k$: $\sigma_P \approx_k \sigma[\uparrow j]$

2. $\uparrow j \in ff(\sigma_P, n+1)$ *and* $|ff(\sigma_P, n+1)| \leq 4\tau + 2\delta$.

3. $state_P(\sigma_P[\uparrow j]) = Init_j$

4. $\forall k : queue_{c_k,P}(\sigma_P[\uparrow j]) = \phi$.

5. Response r *is sent to client* c_k *in* σ_P *iff* r *is sent to client* c_k *in* $\sigma[\uparrow j]$*. Furthermore, if* r *is sent at time* t *in* σ_P*, then* r *is sent at time* t' *in* $\sigma[\uparrow j]$*, where* $t \leq t' \leq t + \delta$.

Proof: **Case 1:** $n = 1$. Let t be the smaller of the times when s_0 halts, or $pstatus_0 = NOTOK$ or $Isprimary[0,1] = false$ (defined by Lemma 6.3.9).

Assume that all the requests m that s_0 responds to in the interval $[0..t]$ are responded to in σ_P as well. m is received at the same times in σ_{PB} and σ_P. However, m is enqueued at the same time it is received in σ_P (this has to be true of all runs of P). We can do this from Lemma 6.3.8. Furthermore, in σ_P the response r has to be sent at the same time the request m is received, whereas in σ_{PB}, r might be sent after δ time units by s_1 (this satisfies (5)). However, r has to be received at the same times by the clients in σ_{PB} and σ_P. We can do this because r is received by the client within 2δ after m is received, and $\rho \geq 2\delta$.

Let $state_0(t) = r_0 \circ r_1 \ldots r_{n-1} \circ r_n$. Let m_{n-1} and m_n be the requests corresponding to r_{n-1} and r_n respectively, which were received at times t_{n-1} and t_n respectively. Let the two outage intervals of σ_P be $I_1 = [t_{n-1}..t_{n-1}]$ and $I_2 = [max(0, t-\tau)..t]$. From Lemma 6.3.17, $Init_j = R = r_0 \circ r_1 \ldots r_{n-1} \circ r_{n-1}$ or $Init_j = M = r_0 \circ r_1 \ldots r_{n-1} \circ r_n$ and $r_0 \circ r_1 \ldots r_{n-2}$ must have been sent to the client. Therefore, from construction, these responses are sent in σ_P as well. If r_{n-1} was not sent to the client, assume in σ_P that m_{n-1} was received, but a response never sent (outage belongs to I_1). If $Init_j = R$, then r_n was not sent to the client (from Lemma 6.3.17)and from Lemmas 6.3.10 and 6.3.11, m_n must have been sent in the interval $[max(0, t - \tau - \rho)..t]$ $(\tau > 2\delta)$. Assume in σ_P that m_n is never enqueued (outage belongs to I_2) or hasn't yet been enqueued, depending on the time m_n is actually sent. We show that P receives no further requests until time $\uparrow j$. Extend I_2 as $I_2 = [max(0, t - \tau)..\uparrow j]$. Therefore, $|I_2| \leq 4\tau + 2\delta$ (using Lemmas 6.3.4 and 6.3.6 one after the other). This satisfies (2). By construction, all requests (other than m_n) that were sent before $\uparrow j$ and were received in $\sigma_{PB}[\uparrow j]$ have already been received in σ_P. Furthermore m_n is never received in σ_P. All requests that were sent before $\uparrow j$ and are not received in $\sigma_{PB}[\uparrow j]$ must have been sent in the interval $[t - \rho..\uparrow j]$ (Lemma 6.3.8). Assume, therefore, that in σ_P these requests are never enqueued (outage belongs to I_2) or haven't yet been enqueued. Therefore, we have shown that P receives no further requests until $\uparrow j$ and so $Init_j = state_P(\uparrow j)$ satisfying (1) and (3).

On the other hand, if $Init_j = M$ and if r_n was not sent to the client, then assume in σ_P that m_n was received, but a response never sent. By Lemmas 6.3.10 and 6.3.11, this outage will again belong to I_2. The rest of the argument then remains as the previous paragraph, satisfying (1), (2) and (3).

We finally show that $\forall k : queue_{c_k,P}(\uparrow j) = \phi$. By the above construction of σ_P, all requests sent before $\uparrow j$, other than m_n, that are received in $\sigma_{PB}[\uparrow j]$ are received in σ_P as well. Also, m_n is not enqueued at time $\uparrow j$. Finally, all requests sent before $\uparrow j$ that are not received in $\sigma_{PB}[\uparrow j]$ must must have been sent in $[t - \rho..\uparrow j]$ (by Lemma 6.3.8). Assume that these are never enqueued in σ_P (again outages lies in I_2) or haven't yet been enqueued. Thus, $\forall k : queue_{c_k,P}(\uparrow j) = \phi$.

Case 2: $n = 2$. From case 1, the lemma is true when $q[1] = s_k$ calls **failure–mode–primary** at time $\uparrow k$ and so there exists a run σ'_P, such that $\forall i : \sigma'_P \approx_i \sigma[\uparrow k]$. Let s_k halt at time $\downarrow k$. We now extend σ'_P. Extend the interval I_2 by $\delta + \rho$. All requests sent in the interval $[\uparrow k - \rho..\uparrow k]$ that haven't been enqueued in σ'_P are never enqueued in σ_P. Futhermore, all requests sent in $[\uparrow k..\uparrow k + \delta]$ that are not sent to s_k are never enqueued in σ_P. All these outages lie in I_2. We now extend σ'_P to σ_P until $\uparrow j$ using an argument similar to case 1.

Assume that all the requests m that s_k responds to in the interval $[\uparrow k..\downarrow k]$ are responded to in σ_P as well. m is received at the same times in σ_{PB} and σ_P. Furthermore, in σ_P the response r is sent at the same time it is sent in σ_{PB} (this satisfies (5)) and r is received at the same times by the clients in both σ_{PB} and σ_P.

Define $I_3 = [\downarrow k..\uparrow j]$. Thus by Lemma 6.3.6, $|I_3| \leq \tau + 2\delta$, which satisfies (2). Using Lemmas 6.3.15 and 6.3.16, $Init_j \sqsubseteq Final_k$. Let $r_n = last(Final_k/Init_k)$ and m_n be the request corresponding to r_n. By an argument similar to case 1, we can show that $Init_j = state_P(\uparrow j)$ with the outage corresponding to m_n (if there is one) lying in the interval I_3, which satisfies (1) and (3). Furthermore, we can show that $\forall k : queue_{c_k,P}(\uparrow j) = \phi$, satisfying (4). □

Lemma 6.3.19 *For any run σ_{PB} of the primary–backup system there exists a run σ_P of the $(\rho, 3, 4\tau + 3\delta + \rho)$ bofo server P such that $\forall k : \sigma_P \approx_k \sigma_{PB}$.*

Proof:

Case 1: $q[i] = \perp$ for all $i > 0$. Therefore, from Lemma 6.3.4, s_0 always remains the primary. If s_0 makes infinitely many broadcasts in σ_{PB}, then from Figure

6.10, all requests have responses. Therefore, assume that all the requests m that s_1 responds to are responded to in σ_P as well. m is received at the same times in σ_{PB} and σ_P. However, m is enqueued at the same time it is received in σ_P (this has to be true of all runs of P). We can do this from Lemma 6.3.8. Furthermore, in σ_P the response r has to be sent at the same time the request m is received, whereas in σ_{PB}, r might be sent after δ time units by s_1. However, r has to be received at the same times by the clients in σ_{PB} and σ_P. We can do this because r is received by the client within 2δ after m is received, and $\rho \geq 2\delta$. However, if s_0 makes only finitely many broadcasts, then again from Figure 6.10, the last request might not have a response. Assume that the last request m_n is received by s_0 at time t. Let the first (and only) outage interval of σ_P be $[t..t]$ and assume in σ_P that the request m is received, but a response never sent.

Case 2: $q[i] = s_j$ for some $i > 0$. Furthermore, let $q[i]$ be such that $q[i]$ never halts in σ_{PB}. There has to exist such an $i \leq 2$ from Lemma 6.3.6 and the fact that $f = 1$. Therefore, from Figure 6.12, $q[i]$ initiates a successful broadcast of the form $(\texttt{mylastlog}, q[i], -)$ at time $\uparrow j$ and from Lemma 6.3.18, there exists a run σ'_P of the $(\rho, 3, 4\tau + 3\delta + \rho)$ bofo server P such that $\forall k : \sigma'_P \approx_k \sigma_{PB}[\uparrow j]$. It is now easy to extend σ'_P to σ_P using an argument similar to case 2 of Lemma 6.3.18. □

Theorem 45 *The protocol in Figures 6.9, 6.10, 6.11 and 6.12 satisfies* Pb4.

Proof: Follows from Lemma 6.3.19. □

Theorem 46 *The protocol in Figures 6.9, 6.10, 6.11 and 6.12 satisfies* Pb5.

Proof: In **failure–mode–primary**, a server remains the primary until it halts. Futhermore, s_0 stops being the primary in procedure **primary** only if it omits to receive a message from s_1. This, however, implies that either s_0 or s_1 must have failed. □

Theorem 47 *The protocol in Figures 6.9, 6.10, 6.11 and 6.12 is δ–blocking.*

Proof: Follows from the description of the protocol. □

Chapter 7

Implementation Results

As we saw in the preceding chapters, 0–blocking primary–backup protocols can theoretically achieve the smallest possible response time. Unfortunately, these protocols cannot be constructed for some kinds of failures, for example send–omission and general–omission failures. We, however, show in this chapter that 0–blocking protocols are still very useful as they can be constructed for the kinds of failures that are expected to occur in many practical primary–backup systems. We also implement two of our 0–blocking protocols, and compare the performance of these protocols with 2δ–blocking protocols.

The chapter is organized as follows. Section 7.1 discusses the failure assumptions of our implementation. Section 7.2 describes our implementation, and Sections 7.3 and 7.4 discuss our results. We conclude the chapter in Section 7.5.

7.1 Failure Assumptions

We implement 0–blocking protocols under two kinds of failures: crash failures and receive–omission failures.[1] Crash failures are a reasonable assumption in a system where network partitions and message losses are highly unlikely. Receive–omission failures are a reasonable assumption in a system where partitions are unlikely, but it is possible that some servers may become overloaded and miss messages, or message buffers at these servers may become full and drop messages.

Even though the 0–blocking protocols that we implement can theoretically achieve the smallest possible response time, it is not clear whether the above restricted set of failures is realistic in practical systems. However, we believe that many primary-backup systems will only have to tolerate a sufficiently restricted set of failures to allow our 0–blocking protocols to be constructed. Given no extra constraints, a practical primary-backup system should have all servers on a single local area network. This is because the time required between the failure of a primary and the takeover by a backup is determined by the bandwidth between the primary and the backups. Using a single local area network makes partitions that separate the servers unlikely. Other failures can still occur due to process crashes and message losses. However, the kinds of message losses that are expected to occur on this type of network are restricted and correspond to our receive–omission failure model. According to [ADKM92], as technology improves and newer, faster networks (such as FDDI) are used, there will be mainly the following two causes for message losses on a local area network:

[1] As we saw earlier, 0–blocking protocols can be constructed for crash+link failures as well. However, to get meaningful results for this model, we would need multiple independent links between servers, which were not available.

1. failure to intercept messages from the network at high transfer rates due to interrupt misses;

2. buffer overflows at the receiver.

As can be seen, these set of failures corresponds to our receive–omission failure model, and therefore one can construct a 0–blocking primary–backup protocol for such a system.

7.2 Description of the implementation

The two 0–blocking protocols that we implement are described in Chapter 5. We, however, extend these protocols so that crashed servers can be re-intergrated. Using these primary–backup protocols, we implement a service that maintains a *fault–tolerant counter.* Clients request a counter value from the service, and on receiving such a request, the service responds by sending the current counter value to the client and incrementing the counter. Informally, we require the counter service to satisfy the following three properties:

1. Two different requests cannot return the same counter value.

2. No counter value can be skipped.

3. Let request R_1 be sent by client c_1 and request R_2 be sent by client c_2, where c_1 may be the same as c_2. If the response to R_1 was received by c_1 before R_2 was sent by c_2, then R_2 cannot return a smaller counter value than the one returned by R_1.

This is a simple service, yet it is a practical one; for example, such a service is a central component of a multicast transport protocol designed by Apple and

Xerox [AFM92]. Furthermore, even this simple counter service is not easy to implement because the counter has to be replicated across servers, and the various copies of the counter have to be kept consistent with each other. Our primary–backup protocol, therefore, was not simplified by using a simple service semantics. Also, since we only analyze the tradeoffs inherent in primary–backup protocols (and not the tradeoffs due to a particular service), we believe our results are applicable to other service semantics as well.

In order to compare the response time of our 0–blocking protocols with the response time of 2δ–blocking protocols, we also implement a simple 2δ–blocking protocol that tolerates server crashes and transient network partitions. This protocol has the same **primary** and **backup** procedures as in Chapter 5. The implementation of the procedures **broadcast**, **delivery–process** and **failure–detector** is informally described below, and the implementation of the procedures **initialize** and **deliver** is the same as in Figure 5.5 for crash failures.

broadcast: The procedure first sends the message to be broadcast to all the backups, and then waits for these messages to be acknowledged. Either all of the backups acknowledge or a timeout occurs. If a timeout occurs, then another copy of the message is sent to the backups that did not acknowledge the first message. Such retries happen for some fixed number of times. If some backup s_j still does not acknowledge, then it is assumed that s_j has crashed as we are assuming that network failures are transient.

delivery–process: Similar to the implementation for crash failures in Figure 5.5, this delivers any new message that is received. In addition, an acknowledgement is sent to the sender of the message. Note that in failure–free runs, this implementation is more efficient in the number of messages sent than the implementation for receive–omission failures ; however, it is less efficient than the implementation for crash failures.

failure–detector: Similar to the implementation for crash failures in Figure 5.5, except that an `alive` message may need to be sent multiple times (as in the implementation of **broadcast**) in order to tolerate network failures.

In our implementation, we assume that there is a known upper bound on the message delivery time and that the servers have approximately synchronized clocks [Cri89,KO87,LMS85]. The implementations were written in C and run on RS/6000 550s with AIX Version 3.1. The machines were connected over a 16 Mbit/s token ring and messages were sent using UDP sockets.

7.3 Results

In this section, we analyze the tradeoffs in failure–free runs between 0–blocking and 2δ–blocking protocols with respect to the response time. In the next section, we discuss how these results can be extended to runs in which failures do occur.

We first show our results for the protocols that use point–to–point communication, and then show the corresponding results for hardware broadcasts. Due to space limitations, we only present a subset of our experiments. Our conclusions in the chapter, however, are consistent with the experiments that we have omitted.

7.3.1 Communication using point–to–point messages

In the following sections, we plot the average response time of the protocols versus the degree of replication, while keeping other parameters constant. The maximum degree of replication that we consider is 5 and the maximum number of clients that we consider is 2. This is because we had only a limited number of machines on which we could run our tests. However, for each figure we discuss how we expect the results to extend to higher degrees of replication and to more clients. Also, in these figures, we only show the response time for odd values of n for receive–omission failures, where n is the degree of replication. This is because, as shown in Chapter 4, the 0–blocking receive–omission protocol can only tolerate up to $\lceil \frac{n}{2} \rceil - 1$ server failures. Finally, in order to control the frequency of requests that are sent by the clients, we add a parameter to the client protocol. We call this the *compute time*, and it corresponds to the amount of time a client waits after receiving the response to a request before it sends out the next request.

Figure 7.1 shows the average response time (in the absence of failures) for the three protocols when there was a single client with a compute time of 10 ms. As a comparison, we have also shown the response time in the absence of any replication *i.e.* $n = 1$. As can be seen from the figure, the two 0–blocking protocols can achieve a small response time for all degrees of replication. However, surprisingly this was no longer true in Figure 7.2 where we have added one more client to the system.

The above behavior occurs because of the following reason. As shown in Chapter 5, the receive–omission protocol generates a large number of messages. On a shared medium like the token ring, these messages can interfere with future requests sent by the clients, thereby increasing the average response time. In Figure 7.2, we see that the interference was large enough to make the response time of the 0–blocking receive–omission protocol greater than the response time of the 2δ–blocking protocol. In fact, Figure 7.3 shows a run in which we have increased

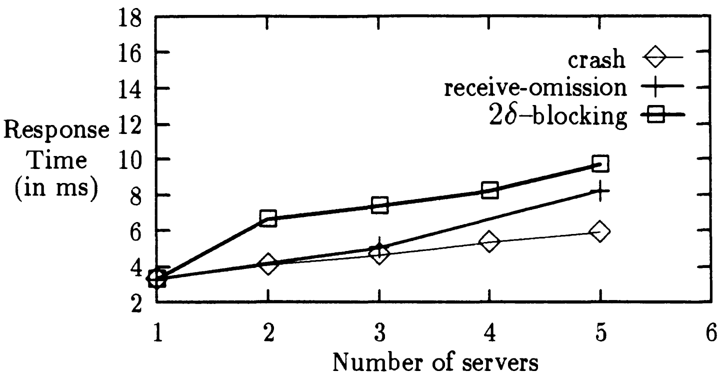

Figure 7.1: Avg. response time vs. degree of replication. Number of clients = 1 and compute time = 10 ms

the compute time to 30 ms. In this case, since fewer requests are being sent per unit time, interference is less likely, and the average response time becomes smaller again.

To verify that the behavior of the receive–omission protocol in Figure 7.2 is due to this interference, we changed the service slightly so that the clients could send *query* requests in which the clients just queried about the current value of the counter. When the primary receives such a query request, it does not have to inform the backups about any update (there is none) and can immediately respond to the client. Thus, if the clients send a large number of query requests, then the average number of messages sent per request are reduced, reducing the interference. The result is shown in Figure 7.4. As in Figure 7.2, there are two clients, each with a compute time of 10 ms. However, in Figure 7.4, half of the requests sent by the clients are query requests. As can be seen, the average response time is now

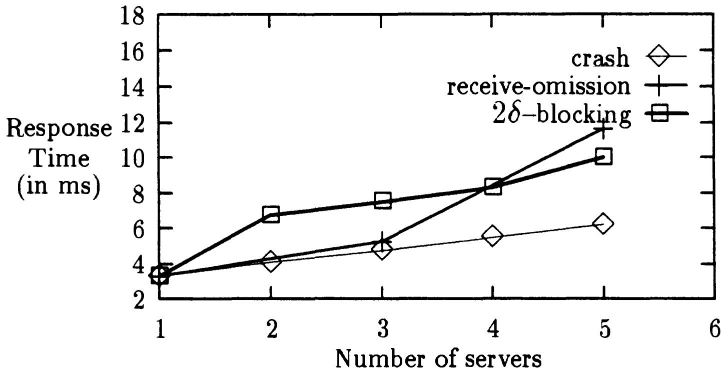

Figure 7.2: Avg. response time vs. degree of replication. Number of clients = 2 and compute time = 10 ms

smaller than in Figure 7.2.

We now attempt to extrapolate the results in this section to higher degrees of replication and to more clients. Clearly more experiments are needed in order to verify these conclusions. We expect the graphs of the protocol for crash failures and the 2δ–blocking protocol to remain essentially the same, with response time increasing more or less linearly with the number of servers becasue both of these protocols only send a linear number of messages per request. Furthermore, the average response time of the crash failure protocol should always remain less than the response time of the 2δ–blocking protocol, because the crash failure protocol never sends more messages than the 2δ–blocking protocol.

On the other hand, if we add more servers in Figure 7.1, then we can expect the response time of the receive–omission protocol to eventually become greater than that of the 2δ–blocking protocol because more servers imply more messages

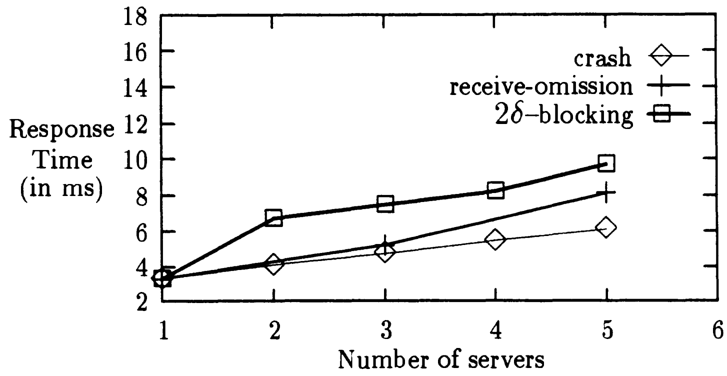

Figure 7.3: Avg. response time vs. degree of replication. Number of clients = 2 and compute time = 30 ms

per request, which then implies more interference. Furthermore, the larger the number of clients, the earlier the two graphs will intersect. However, if we lower the frequency of the requests (as in Figure 7.3), then we can expect the interference to become smaller again, which will move the intersection point to higher degrees of replication.

7.3.2 Communication using hardware broadcasts

In the previous section, all communication between servers was through point-to-point messages. However, most local area networks allow for more efficient communication by using hardware broadcasts. We therefore reimplemented all our of protocols to use broadcast communication wherever possible.

The new experiments corresponding to the parameters in Figures 7.1 and 7.2 are shown in Figures 7.5 and 7.6 respectively. In both figures, the graphs of the two 0–blocking protocols are almost indistinguishable. As can be seen again, a

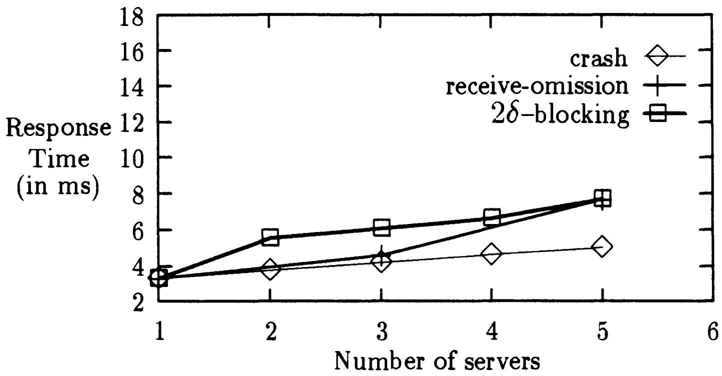

Figure 7.4: Avg. response time vs. degree of replication with query requests. Number of clients = 2 and compute time = 10 ms

very small response time can be achieved by using 0–blocking protocols. Also, as expected, the response time of most of the protocols is smaller than the corresponding response time for point–to–point communication because fewer messages were being generated (one message instead of n messages in most cases). A smaller number of messages means that there is less interference. The only exception was the crash failure protocol with $n = 2$. This was because only one message was being sent (to the lone backup) per request, both in the point–to–point and the broadcast case. However, when this message was sent using a hardware broadcast, the primary received its own message, which slowed down the subsequent response. However, what surprised us more was that the 0–blocking protocol for receive–omission failures gave a flat curve—the response time was independent of the number of servers. We expected such a behavior for the crash failure protocol because in this case, the primary sends just one broadcast message per request,

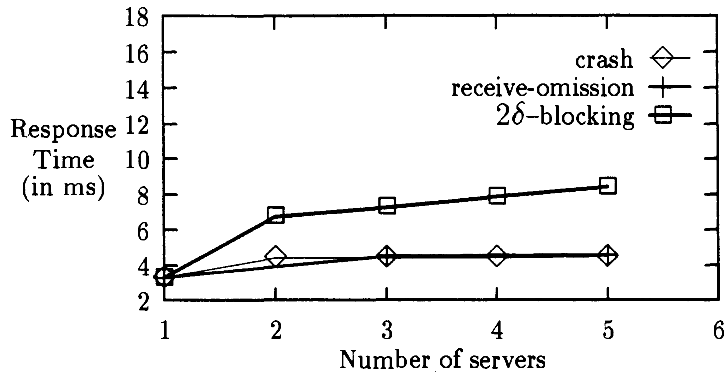

Figure 7.5: Avg. response time vs. degree of replication. Number of clients = 1 and compute time = 10 ms

independent of the number of servers. In the receive–omission protocol, however, each backup relays the primary's message, and therefore the total number of relays depend on the number of backups. Thus, more relays should have caused more interference, resulting in higher response times.

The next experiment (shown in Figure 7.7) showed us the reason for the response time being independent of the degree of replication. We reduced the compute time to 2 ms, thereby increasing the possibility of interference. We now saw the positively sloped curve for receive–omission failures that we expected earlier. In the earlier figures, the number of messages generated by the backups was not sufficient to interfere with future requests, because a small number of requests were being sent per unit time. On the other hand, when we increased the frequency of the requests, the interference became large enough to increase the response times.

Note, however, that the response time of the 2δ–blocking protocol was never

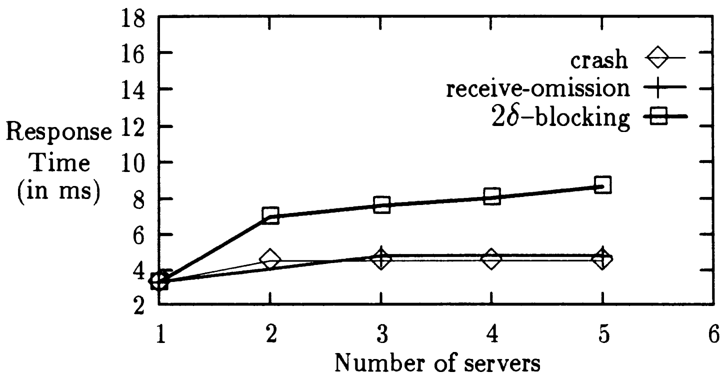

Figure 7.6: Avg. response time vs. degree of replication. Number of clients = 2 and compute time = 10 ms

independent of the number of servers. As described in the implementation of the **broadcast** procedure for the 2δ–blocking protocol, the primary waits for the backups to acknowledge before responding to the client. Thus, the number of messages that the primary waits for increases with the number of servers, causing the response time to increase as well.

We now extrapolate our results to higher degrees of replication and to more clients. Keeping other parameters constant, we should expect the response time of the crash failure protocol to be always independent of the number of servers. This should be the case, because as discussed before, the primary always sends one message per request. However, the exact value of this response time will clearly depend on the number of clients and the frequency of requests. On the other hand, for reasons described in the previous paragraph, the response time of the 2δ–blocking protocol should increase more or less linearly with the number of

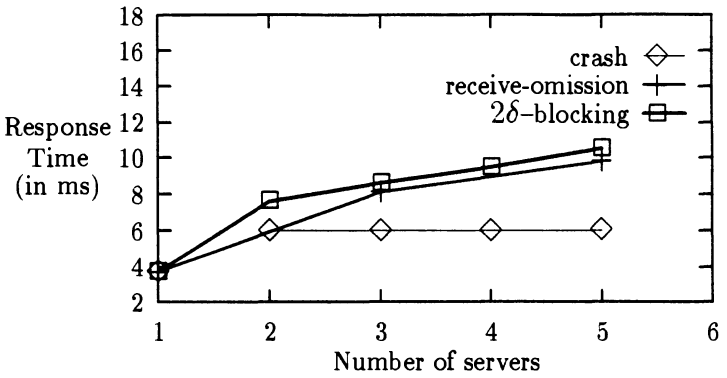

Figure 7.7: Avg. response time vs. degree of replication. Number of clients = 2 and compute time = 2 ms

servers.

The situation is a little more complicated for receive–omission failures, because our experiments obtained both flat and positively sloped graphs for this protocol. Given the results in Figure 7.7, one can expect the response times in Figures 7.5 and 7.6 to eventually start increasing, possibly as a step function, as the number of servers increase. A similar behavior should be seen if we increase the number of clients. However, unlike the point–to–point case, the response time of the receive–omission protocol should always be less than the response time of the 2δ–blocking protocol. This is because the former never sends more messages than the latter.

7.4 Extending the results to runs with failures

We did not consider failures in the previous section because the results in the case of failures will depend on the underlying system and the particular service being implemented. For example, the amount of time it takes to reintegrate a new server usually depends on the service semantics and the amount of information that has to be sent to the new server.[2] Similarly, the failover time depends on the maximum message delivery time. However, we can still draw some useful conclusions about runs with failures from our results.

If failures are rare and the 0–blocking and 2δ–blocking protocols have similar failover and reintegration times, then it is easy to see that the results in the previous sections should still be qualitatively true *i.e.* the 0–blocking protocols can achieve a smaller response time than 2δ–blocking protocols in many cases. In addition, these results can also be used in deciding the right degree of replication to be used when implementing these protocols. If servers use point–to–point communication, then we saw that the average response time of all protocols (0–blocking and 2δ–blocking) rises with the number of servers. Therefore, if failures are infrequent (which is usually the case), and reintegration does not cost too much (which was also true in our counter service), then in designing a primary–backup service that uses point–to–point communication, one should keep the degree of replication as low as possible, and reintegrate servers only when the older servers have crashed. This should result in a smaller average response time since the degree of replication is always kept low.

On the other hand, if communication can be done using hardware broadcasts, then we saw that the response time of 0–blocking protocols can be independent of the number of servers. In this case, it may be possible to keep the degree of replication for these protocols high (since this does not increase the response time),

[2]In our case, reintegration took almost no time because only the current counter value and some information regarding the last request sent by each client needed to be sent to the new server.

and save on some reintegration cost as well, further reducing the response time. We can achieve this by not doing any reintegration until a large number of servers have crashed, and then reintegrating more than one server at a time, thereby saving on the total reintegration time. This clearly assumes that servers can be reintegrated in parallel without too much additional expense.

7.5 Discussion

In this chapter, we analyzed a subclass of primary–backup protocols, *i.e.* 0–blocking protocols, and showed that these protocols can be constructed for the kinds of failures that are expected to occur in many primary–backup systems. As expected, our experiments showed that these protocols can achieve a very small response time as compared to the 2δ–blocking protocols, and therefore, these protocols should be preferred when designing many primary–backup systems.

However, our results also showed that the 0–blocking property is not always enough to achieve a small response time. We designed the 0–blocking protocol tolerating receive–omission failures using the translation technique [NT88], as this technique resulted in a protocol that used the optimal number of servers and had the optimal failover time. This technique, however, can create protocols that send a large number of messages, and as we showed in Section 7.3, this large number of messages sometimes can lead to overly-high contention on the network, causing the response time of the receive–omission protocol to become large. Thus the communication cost of a protocol is also an important parameter in determining the response time, and we are currently looking at ways to reduce this cost.

Chapter 8

Survey of Existing Primary–Backup Protocols

This chapter presents a brief survey of the primary–backup protocols that exist in the literature. This survey is by no means complete because we have only included protocols that we consider to be well known. If the reader is interested, the bibliography gives references to some other protocols that we have omitted.

Furthermore, in this survey it is not possible for us to talk about every possible characteristic of the primary–backup protocols. We will, therefore, concentrate only on those characteristics that relate to this thesis. These are the system model (asynchronous or synchronous), the types of failures that are tolerated by the protocols, and the values of the three metrics—degree of replication, blocking time and the failover time.

The rest of the chapter presents the various protocols in chronological order. Section 8.1 presents a primary–backup protocol due to Alsberg and Day [AD76], Section 8.2 presents Tandem [Bar81], Section 8.3 presents Viewstamp Replication [OL88], Section 8.4 presents ECHO [MHS89], Section 8.5 presents HA-NFS [BEM91] and Section 8.6 presents HARP [LGG+91]. Most of the descriptions of the protocols have been taken directly from the above references. They have, however, been edited for conciseness. Furthermore, while giving the values for blocking time and failover time, we will assume that local computation at a server takes negligible time (since this is what we had assumed in our system model), and concentrate only on the delays due to message delivery.

8.1 The Alsberg–Day Protocol

We believe this protocol to be the earliest primary–backup protocol appearing in the literature. The protocol is synchronous, and tolerates server crashes.[1] The protocol that we outline in this section only tolerates a single server crash, but it can be extended to tolerate multiple failures.

In this protocol, a client sends a request to the service and then blocks waiting for either a response from the service or a timeout.

- If the request arrives at the primary, then the primary performs the requested update, sends a state update message to the backup, and blocks. The backup, upon receiving the state update message, updates its state, sends the response to the client, and finally sends an acknowledgement to the primary saying that it performed the update. On receiving the acknowledgement, a primary can unblock and process the next pending request.

- If the request arrives at the backup, then the backup forwards the request to

[1]The authors also claim that the protocol tolerates network partitions. However, during partitions the primary and the erstwhile backup can diverge, giving inconsistent responses. In the analysis that follows, we assume that partitions do not occur.

the primary. The primary, upon receiving the forwarded request, performs the update, sends the response to the client, and finally sends a state update message to the backup (which then updates its state and discards the request).

Failures are detected by lost acknowledgement messages. In addition, failures are also detected by sending periodic "Are you alive" messages. In case the primary fails, the backup takes over as the new primary.

The above protocol requires two servers to tolerate a single server crash and has a blocking time of δ, where δ is the maximum message delivery delay. The failover time depends on the frequency that "Are you alive" message are sent. If we assume that the period between "Are you alive" messages is τ, then the failover time for this protocol is $\tau + 2\delta$.

8.2 The Tandem Protocol

This protocol is synchronous and is designed to tolerate a single crash+link failure. Any Tandem system consists of multiple nodes connected by a network. Each of these nodes consists of multiple processor and I/O controller modules interconnected by redundant buses. Each processor in the node can support concurrent processes (system or application), and the goal of the system is to make these processes fault–tolerant.

Processes are made fault–tolerant by using *process–pairs.* Process pairs are implemented by replicating each process on two different processors in the node, with one process being the primary and the other being the backup. Requests are sent to the primary of such a pair. The primary then sends a state update message to the backup over one of the redundant busses. Once an acknowledgement is received from the backup, the response is sent to the client. If an acknowledgement is not received for some time (one second in the protocol), then the underlying message mechanism resends the state update message over the second bus. Sequence numbers are used in order to prevent duplicates.

The backup process becomes the primary when it detects that the processor on which the primary resided has crashed, as follows. Every processor in the node periodically sends an "I am alive" message to all other processors, over all the redundant buses. If such a message is not received from a processor, then that processor is declared crashed and any backup whose primary was on that processor becomes the primary.

The above protocol uses two processors to tolerate a single failure. As can be seen from the description, the blocking time is 2δ (one seconds in the protocol). Also, the failover time of the protocol is 4 seconds. However, just from the description in the paper, we are unable to relate this time to τ and δ.

8.3 The Viewstamped Replication Protocol

This protocol is asynchronous and tolerates failures of both the servers and the network. Servers can crash, and the network can lose, delay and duplicate messages, or deliver messages out of order. Furthermore, link failures can cause the network to partition into subnetworks.

The computation model consists of objects (called *modules*) and objects can make calls on each other. The caller object is the client and the callee is the server. The goal is to make the objects fault-tolerant by using replication. This is done by designating one copy of the object as the primary and the others as backups. The primary is responsible for the processing of transactions [BHG87] that use the object; it notifies the backups of what it has done.

The various replicas of an object monitor each other by sending periodic "I am alive" messages. If a replica notices that it is not communicating with some other replica, or if it notices that it is communicating with a replica that it could not communicate with previously, it initiates a *view change.* This view change might result in a reorganization with a new primary being selected if necessary. However, if a view change occurs, then it is ensured that the effect of committed transactions survives this change. Informally, this is guaranteed by ensuring that the new view contains at least a majority of the replicas. One of these replicas must have been active in the last view, and therefore would have an up-to-date copy of the object's state. This up-to-date replica is identified by using a new kind of timestamp, called *viewstamp*, and is then used to bring the other replicas in the newly formed view up-to-date.

However, the view change algorithm given in the paper is not guaranteed to terminate. Message delays due to the asynchrony of the system might cause the algorithm to run forever without selecting a new primary. Thus, the primary–backup protocol is not live, and as we showed in Chapter 2, this is in fact unavoidable in an asynchronous system.

The above protocol tolerates any number of server failures. However, it requires at least a majority of the servers to be up in order to make progress. Since the protocol is asynchronous, the blocking time and the failover time are both unbounded. However, for asynchronous protocols, it is possible to change the definition of blocking so that a blocking protocol is one that requires a message exchange before a response can be sent, whereas a non–blocking protocol does not require such an exchange.[2] We do not address this issue in this thesis.

8.4 ECHO

This protocol is asynchronous in the sense that there is no bound on the speeds of servers or messages. However, the protocol assumes that each server has a local clock, and as long as the server is up, this clock has a bounded drift rate of ρ. This clock will be used by the protocol to measure time intervals. The ECHO protocol tolerates sever crashes and network failures. The network may duplicate or lose messages, but it does not alter or spontaneously generate messages.

The protocol was designed to provide a replicated, fault–tolerant file system. Copies of a file are replicated across severs, with one server being the the primary and the remaining servers being backups. A server s can become the primary only if a sub-majority of the other servers (who will act as backups) accept s as the primary. Once s becomes the primary, it remains so for some bounded amount of time, following which it has to ask the backups again whether it can continue to be the primary. This bounded time interval can be measured using the clock at s. Since a server can become the primary only if it can convince a majority of the servers to be its backups, the protocol ensures that there is at most one primary at any time.

Read and write requests to a file are sent to the primary. If the primary receives a read request, it handles it locally because there can be no concurrent writes in

[2]The viewstamped replication protocol is, therefore, blocking under this definition.

progress (all writes have to go through the primary). If a write request arrives, then the primary sends the write request to the backups, and starts an internal timer. If the timer expires before all the acknowledgements are received, then an exception is raised. On the other hand, if all the acknowledgement are received, then the write completes. This ensures that completed writes survive failures in a manner similar to the protocol in the previous section (however, the techniques used to identify an up-to-date server are quite different). The protocol also allows for caching at the clients. However, before a client can cache a copy, the client has to obtain a token for the file from the primary. Information about these tokens is also replicated at the servers.

The ECHO protocol tolerates any number of server failures, but requires at least a majority of the servers to be up in order to make progress. The protocol, however, is not live. In particular, it is possible that due to asynchrony, the protocol may never elect a new primary after a failure, and so the failover time is unbounded. The severs in the ECHO protocol maintain clocks that run at the rate of real time, and, therefore, the blocking time of the protocol is bounded. As described in the previous paragraph, whenever the primary receives a write request, it sends that request to the backups and starts an internal timer. If we assume that the length of this timer is γ (the notation used in the paper), then the blocking time of the protocol is $\gamma(1+\rho)$.

8.5 HA-NFS

This is a synchronous protocol and the goal of this protocol is to provide a highly available network file server (HA-NFS) under crash+link failures. The protocol tolerates a single crash failure by using two servers. One server is the primary, the other is the backup. The servers are connected to a dual-ported disk (in reality, there could be multiple disks). Only one server (the current primary) has access to the disk at any time. Disk failures are tolerated by mirroring the disk, and link failures are tolerated by replicating the network between the clients and the servers. The dual-ported disk is used as an additional communications link between the two servers.

During normal operation, client requests are sent to the primary, which writes the updates to the disk and then replies to the client. The primary does not inform the backup of the update, because the disk is dual-ported and the backup can access the disk when it takes over as the primary. The only communication between the two servers during normal operation is to exchange periodic "Are you alive" messages that must be acknowledged.

In case the backup does not receive an acknowledgement after repeated "Are you alive" messages, then either the primary has crashed or the link between the primary and the backup has failed. In order to prevent the backup from taking over while the primary is still up, the backup tries to communicate with the primary using the dual-ported disk hardware. If the backup finds that it cannot communicate with the primary even over this redundant link, then it becomes the new primary and takes over control of the dual-ported disk.

The above protocol requires two servers to tolerate a single failure, and has a blocking time of zero. The failover time depends on the interval between successive "Are you alive" messages, as well as the number of retries of this message. If we assume that these values are τ and k respectively, then the failover time is

$\tau + (k+1)2\delta$. To simplify things, we have assumed that it takes the same amount of time to communicate over the link or the disk hardware.

8.6 HARP

This protocol is asynchronous, but the servers have local clocks which are approximately synchronized with the clocks of the other servers. The goal of the protocol is to provide a reliable file server, and it tolerates power failures, servers failures and network failures. The network may lose or duplicate messages, or deliver them late or out of order; it addition it may partition so that some servers are unable to send messages to some other servers. The protocol described in the paper tolerates a single failure, but can be extended to tolerate multiple failures.

In this protocol, the copies of the files are replicated over two servers, with one server being the primary and the other being the backup. There is a third server, called the *witness* which takes part in the protocol in case of failures. Updates are sent to the primary, who first informs the backup and waits for it to acknowledge. Similar to ECHO, if an acknowledgement is not received within some timeout interval, then an exception is raised. On the other hand, if an acknowledgement is received from the backup, then the primary replies to the client. Harp achieves good performance by recording the effects of an operation in a log that resides in volatile memory; operations in the log are applied to the file system in the background. However, Harp tolerates power failures by equiping the servers with an uninterruptible power supply (UPS). This allows the severs to write the logs to disk in case the power fails.

The primary and the backup exchange "Are you alive" messages with each other in order to detect failures. In case one of the servers detects that it cannot communicate with the other server, it tries the form a new *view* along with the witness, and attempts to elect a new primary if necessary. A new view can be formed only if it has two servers in it. Since there are only three servers, this prevents the primary and the backup from concurrently forming two independent views. The view change protocol given in the paper requires two phases of message exchange in order to form a new view; one to send a request to the witness asking it to join the new view, and the second to inform the witness of the formation of the new view. However, since the protocol is asynchronous, it is possible that due to message delays, a new primary is never elected.

The protocol requires $2f + 1$ servers in order to tolerate f failures, and has unbounded failover time. Even though the protocol assumes that the servers maintain synchronized clocks, it does not mention whether these clocks are also synchronized with real time. However, if we assume this fact, and assume that these clocks have some bounded drift rate ρ, then as in ECHO, Harp also has a blocking time of $\gamma(1 + \rho)$ for some timeout interval γ.[3]

8.7 Summary

Table 8.1 summarizes the values of the three metrics for the protocols surveyed in this chapter. Since most of the protocol descriptions in this chapter tolerate only a single failure, the table shows the values of the metrics assuming that the number of failures to be tolerated equals one. Some of these, however, can be generalized as described earlier.

We now briefly compare the results in Table 8.1 with those in Table 4.1. How-

[3]On the other hand, if we do not assume this, then the protocol has a blocking time of ∞.

Table 8.1: Summary of survey results

Protocol name	Failure type	Degree of replication	Blocking time	Failover time
Alsberg & Day	crash	2	δ	$\tau + 2\delta$
Tandem	crash + link	2	2δ	?†
Viewstamped replication	crash + partitions	3	∞	∞
Echo	crash + partitions	3	$\gamma(1+\rho)$	∞
HA-NFS	crash + link	2	0	$\tau + (k+1)2\delta$
Harp	crash + partitions	3	$\gamma(1+\rho)$	∞

† Not available.

ever, since we have considered only synchronous primary–backup protocols in this thesis, we restrict our attention to Alsberg & Day, Tandem and HA-NFS.

In the Alsberg & Day protocol, a backup can recive requests from the clients, and therefore, the protocol does not satisfy Pb3. However, for crash failures, our lower bound results do not depend on Pb3, and so the Alsberg & Day protocol is optimal for degree of replication and not optimal for blocking time. The sub-optimal blocking time is the result of allowing the backup to send a response. The paper is not clear why the authors chose to allow this. One can hypothesize that they were concerned with *transient* link failures. In particular, suppose the protocol were changed so that the primary sent the response to the client after queueing the state update message to be sent to the backup. Now if the primary crashes before the state update message is sent to the backup, then a client has received a response to a request that was never received by the backup. This would violate Pb4. By having the backup send the response, as done in the protocol, if a partition does not occur, then both the primary and the backup will update their state with respect to any request. The failover time of the protocol depends on the frequency that "Are you alive" message are sent. If we assume that the period between "Are you alive" messages is τ, then the failover time for this protocol is $\tau + 2\delta$. The protocol does not, however, use synchronized clocks. Our upper bounds on failover times do assume synchronized clocks. Thus, our upper bounds on failover time are incomparable. We do not know whether this protocol achieves optimal failover time.

The Tandem protocol uses two servers to tolerate a single crash failure, and two links to tolerate a single link failure. Since there are two links between the two servers, and only one of these links can fail, it turns our that our crash failure bounds apply to this protocol. This is for the following reason. Our system model assumed that there is exactly one link between any two processes. In the Tandem protocol, it is assumed that no more than one of the two links can be faulty. If this is the case, then any message can be simultaneously sent over both the links, thus guaranteeing that at most one copy of the message can be lost due to link failures. All our bounds that hold in the absence of link failures can be applied to protocols, like Tandem's, that utilize multiple links. The Tandem protocol, therefore, has optimal degree of replication. The blocking time for this protocol is 2δ, and this is not optimal. However, using our optimal protocol would increase message traffic, which Tandem might not want to do. Finally, because this protocol does not assume synchronized clocks, the optimality of its failover time remains an open question.

As with the Tandem protocol, our lower bounds for crash failures apply to HA-NFS as well because only one of the communication channels between the servers can fail. The HA-NFS protocol requires two servers to tolerate a single failure, and has a blocking time of zero. Thus the protocol has optimal degree of replication and optimal blocking time. The failover time depends on the interval between successive "Are you alive" messages, the number of times it is sent before detecting a failure, and the time needed to communicate using the disk as a channel. The optimality of the failover time remains an open question because this protocol does not assume synchronized clocks (and our bounds do).

Chapter 9

Conclusions

This thesis presents a systematic analysis of the primary–backup approach, both from the theoretical viewpoint of specification, lower bounds and upper bounds, as well as from the practical viewpoint of performance tradeoffs among protocols.

In Chapter 2, we begin by showing that it is impossible to construct primary–backup protocols that are both safe and live in an asynchronous environment. Chapter 3 then presents a precise specification for primary–backup protocols in a synchronous environment, and using this specification, Chapter 4 presents lower bounds for three key cost metrics of primary–backup protocols—degree of replication, blocking time and failover time. These lower bounds characterize the minimum number of servers needed to implement any protocol (n_s), the amount of time it can take to respond to a client request, and the amount of time during which there may not be a primary. The lower bounds depend on three parameters—the message delivery delay (δ), the number of failures to be tolerated (f), and the class of failures to be tolerated. In our analysis, we consider the following classes of failures—crash, crash+link, send–omission, receive–omission and general–omission failures.

In some cases, our lower bound results were very surprising. For example, in earlier results that we are aware of, the degree of replication only affected the correctness of the protocol, and never the performance. However, for receive–omission failures, it is the case that the degree of replication affects both the correctness as well as the performance of the protocol. In particular, if we increase the degree of replication, then we can construct a more efficient protocol by reducing the blocking time. Using these lower bounds, we have also been able to understand some previously unexplained properties of primary–backup protocols. For example, some protocols use more than $f + 1$ servers in order to tolerate f failures. Our lower bounds on the degree of replication, however, show that this is in fact necessary for some kinds of failures. Similarly, in some other protocols, the response time is large because the primary has to wait for acknowledgements from the backups before it can respond to a client request . Again, we have been able to show that for some kinds of failures, this is also necessary.

Chapters 5 and 6 show primary–backup protocols, and prove that all except two of our lower bounds are tight. These two lower bounds correspond to the degree of replication and the blocking time respectively for receive–omission failures. Showing these lower bounds tight, therefore, remains an open problem. Here also, there were some surprising results. For example, in case of receive–omission failures, the optimal protocol in one case had a surprising (and undesirable) scenario: a non-faulty primary is forced to relinquish control to a backup that it knows to be faulty! The existence of such a scenario is, however, not peculiar to our protocol. We show that relinquishing control to a faulty backup is indeed necessary

Table 9.1: Summary of lower bound and upper bound results

Failure model	Degree of replication	Blocking time	Failover time
crash	$n_s > f$	0	$f\delta$
crash+link	$n_s > f + 1$ †	0	$2f\delta$
receive omission	$n_s > \lfloor \frac{3f}{2} \rfloor$ * †	δ if $n_s = 2f$ and $f = 1$ † 2δ if $n_s \leq 2f$ and $f > 1$ * † 0 if $n_s > 2f$	$2f\delta$
send omission	$n_s > f$	δ if $f = 1$ 2δ if $f > 1$	$2f\delta$
general omission	$n_s > 2f$	δ if $f = 1$ 2δ if$f > 1$	$2f\delta$

† $D < \Gamma$ assumed.

* Bound not known to be tight.

to achieve optimal protocols in this failure model. Another surprising property was the following. In some protocols, in order to achieve the optimal response time, the sever that received the request (*i.e.* the primary) was not the server that sent the response to the client. We show that this is necessary as well. Table 9.1 summarizes our lower bound and upper bound results.

As we had mentioned earlier, the contributions of this thesis are not entirely of a theoretical nature. In Chapter 7, we implement and analyze an interesting subclass of primary–backup protocols—protocols that can theoretically achieve the smallest possible response time. Even though these protocols, called 0–blocking protocols, can only be constructed for a restricted class of failures (as shown in Chapter 4), we argue that these are exactly the kinds of failures that are expected to occur in many practical systems, and as a result, one can implement 0–blocking primary–backup protocols for these systems.

In order to obtain our performance results, we implement these protocols on a cluster of RS/6000 550s connected over a 16Mbit/s token ring. As expected, our experiments show that these protocols can achieve a very small response. In fact, some of the response times achieved by these protocols were less than 1 ms larger than the response time of an unreplicated service.

However, surprisingly, there were cases under which the 0–blocking protocol for receive–omission failures did not achieve a small response time. In particular, when servers used point–to–point communication, then under certain circumstances (which depended both on the number of servers and clients, and on the frequency of client requests), this protocol lead to overly-high contention on the network, causing its response time to become large. Thus, the 0–blocking property of a protocol may not sufficient under all conditions to achieve a small response time, and the message cost of a protocol might become the dominating factor under high loads. We are currently looking at ways to reduce the message cost of our protocols.

Finally, in Chapter 8, we present a brief survey of existing primary–backup protocols. Applying our results to the synchronous protocols, we show that the Alsberg & Day protocol, Tandem and HA-NFS have optimal degrees of replication. HA-NFS also has optimal blocking time, whereas Alsberg & Day and Tandem do not have optimal blocking times. Our results on failover time, however, are incomparable because we assume synchronized clocks and the protocols we survey do not assume this.

The results of this thesis appear in the following papers: [BMST91,BMST92a, BMST92b,BM92,BMST93].

9.1 Future Directions

In this thesis, we analyze only synchronous primary–backup protocols because we had shown that it is impossible to construct safe and live protocols in an asynchronous environment with failures. Our impossibility result relied on the fact that consensus can not be solved in an asynchronous environment. However, weakenings of the asynchronous model, in which consensus can be solved, have been studied in the literature [DLS88,CT91], and therefore, it might be possible to construct primary–backup protocols in such a weakened environment. Finding lower bounds and upper bounds on primary–backup protocols in such cases remains an open problem.

As we had mentioned earlier, another technique for building replicated services is the state–machine approach. Even though both the state–machine approach and the primary–backup approach have been used in practice, the tradeoffs between these two approaches have not been systematically studied. For example, it is not known which of the two approaches is better suited to any particular application. However, given the results of this thesis about the primary–backup approach, and similar existing results for the state–machine approach, we hope that it will be possible to compare and contrast the two approaches, and better identify the applicability of either approach to a given application.

Bibliography

[AD76] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 627–644, October 1976.

[ADKM92] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *The Twenty-Second International Symposium on Fault-Tolerant Computing Systems*, pages 76–84. IEEE Computer Society Technical Committee on Fault-Tolerant Computing, July 1992.

[AFM92] Susan M. Armstrong, Alan O. Freier, and Keith Marzullo. Multicast transport protocol. Internet RFC 1301, February 1992.

[AS86] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 23:177–180, November 1986.

[BAA+92] O. Babaoglu, L. Alvisi, S. Amoroso, R. Davoli, and L. A. Giachini. Paralex: An environment for parallel programming in distributed systems. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 178–187, July 1992.

[Bar81] J.F. Barlett. A NonStop^{TM} Kernel. In *Proceedings of the Eighth ACM Symposium on Operating System Principles, SIGOPS Operating System Review*, volume 15, pages 22–29, December 1981.

[BBG+89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM TOCS*, 7(1):1–24, February 1989.

[BEM91] Anupam Bhide, E.N. Elnozahy, and Stephen P. Morgan. A highly available network file server. In *Winter 1991 USENIX Conference*, pages 199–205. USENIX Association, January 1991.

[BGT90] Navin Budhiraja, Ajei Gopal, and Sam Toueg. Early stopping distributed bidding and applications. In *Proceedinds of the Fourth Inter-*

national Workshop on Distributed Algorithms, pages 304–320, September 1990. Springer Verlag, Lecture Notes in Computer Science 486.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BM92] Navin Budhiraja and Keith Marzullo. Highly-available services using the primary-backup approach. In *Proceedings of the Second Workshop on the Management of Replicated Data (WMRD-II)*, pages 47–50, Monterey, CA, November 1992.

[BMST91] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Lower bounds for primary backup implementations of bofo services. In *Proceedings of the ONR Second Annual Workshop on Ultradependable Multicomputers and Electronic Systems, Washington D.C.*, pages 81–86, November 1991.

[BMST92a] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, pages 362–378, Haifa, Israel, November 1992.

[BMST92b] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary–backup protocols: Lower bounds and optimal implementations. In *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*, pages 187–198, Mondello, Italy, September 1992.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary–backup approach. In Sape Mullender, editor, *Distributed Systems.* ACM Press, 1993.

[CASD85] Flaviu Cristian, Houtan Aghili, H. Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, Michigan, June 1985. A revised version appears as IBM Technical Report RJ5244.

[CDD90] Flaviu Cristian, Bob Dancey, and Jon Dehn. Fault-tolerance in the advanced automation system. In *20 Int. Symp on Fault-Tolerant Computing Systems (FTCS 20), Invited Paper*, pages 6–17, June 1990.

[CDS90] Flaviu Cristian, Danny Dolev, and Ray Strong. New latency bounds for atomic broadcast. Technical report, IBM Almaden Research Center, 1990.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Cri89] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[CT91] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340, Montreal, Canada, August 1991. ACM SIGOPS-SIGACT.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DRS90] D. Dolev, R. Reischuk, and H.R. Strong. Early stopping in byzantine agreement. *JACM*, 37:720–741, 1990.

[Fis83] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical Report DCS/RR-273, Department of Computer Science, Yale University, June 1983.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[GSTC90] Ajei Gopal, Ray Strong, Sam Toueg, and Flaviu Cristian. Early-delivery atomic broadcast. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 297–310, Qeubec City, Quebec, August 1990. ACM SIGOPS-SIGACT.

[GT91] Ajei Gopal and Sam Toueg. Inconsistency and contamination. In *Tenth ACM Symposium on Principles of Distributed Computing*, pages 257–272, Montreal, Canada, August 1991. ACM SIGOPS-SIGACT.

[Had84] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations.* Ph.D. dissertation, Harvard University, June 1984. Department of Computer Science Technical Report 11-84.

[Her91] Maurice P. Herlihy. Wait free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[KO87] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam85] Leslie Lamport. Basic Concepts. In *Distributed Systems–Methods and Tools for Specification.* Lecture Notes in Computer Science, Volume 193, Springer-Verlag, Berlin, 1985.

[LF82] Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Op. 62, SRI International, April 1982.

[LGG+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Michael Williams. Replication in the Harp file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 226–238, October 1991.

[LL86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 29–39, Calgary, Alberta, August 1986. ACM SIGOPS-SIGACT.

[LMS84] Leslie Lamport and Peter M. Melliar-Smith. Byzantine clock synchronization. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 68–74, Vancouver, British Columbia, August 1984. ACM SIGOPS-SIGACT.

[LMS85] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[MHS89] Timothy Mann, Andy Hisgen, and Garret Swart. An algorithm for data replication. Technical Report 46, Digital Systems Research Center, 1989.

[MS88] Keith Marzullo and Frank Schmuck. Supplying high availability with a network file system. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 447–453, May 1988.

[NT87] Gil Neiger and Sam Toueg. Substituting for real time and common knowledge in asynchronous distributed systems. In *Sixth ACM Symposium on Principles of Distributed Computing*, pages 281–293, Vancouver, Canada, August 1987. ACM SIGOPS-SIGACT.

[NT88] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed systems. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 248–262, Toronto, Ontario, August 1988. ACM SIGOPS-SIGACT.

[OL88] B. Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly available distributed systems. In *Seventh ACM Symposium on Principles of Distributed Computing*, pages 8–17, Toronto, Ontario, August 1988. ACM SIGOPS-SIGACT.

[PSL80] M. Pease, R. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PT86] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.

[SB89] N. A. Speirs and P. A. Barrett. Using passive replicates in delta-4 to provide dependable distributed computing. In *The Nineteenth International Symposium on Fault-Tolerant Computing Systems*, pages 184–190, June 1989.

[Sch80] Fred B. Schneider. Ensuring consistency on a distributed database system by use of distributed semaphores. In *Proceedings of the International Symposium on Distributed Databases*, pages 183–189, Paris, France, March 1980.

[Sch90] Fred B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[Sie92] Alexander Siegel. *Performance in Flexible Distributed File Systems.* Ph.D. dissertation, Cornell University, Department of Computer Science, 1992. Department of Computer Science TR 92-1266.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[WLG+78] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of IEEE*, 66(10):1240–1255, October 1978.