Chapter 8:
The Primary–Backup Approach

Navin Budhiraja [*]
Keith Marzullo [†]
Fred B. Schneider [‡]
Sam Toueg [§]
Department of Computer Science
Cornell University
Ithaca, New York 14853, USA

1 Introduction

One way to implement a fault-tolerant service is by using multiple servers that fail independently. The state of the service is replicated and distributed among these servers, and updates are coordinated so that even when a subset of servers fail, the service remains available.

Such fault-tolerant services are generally structured in one of two ways. One approach is to replicate the service state at all servers and to present the client requests, in the same order, to all non-faulty servers. This service architecture is commonly called *active replication* or *the state machine approach* and is discussed in Chapter 7. The other approach is to designate one server as the *primary* and all the others as *backups*. Clients make requests by sending messages only to the primary. If the primary fails, then a *failover* occurs and one of the backups takes over. This service architecture is commonly called the *primary–backup* or the *primary–copy* approach [1] and has been widely used in commercial fault-tolerant systems.

With both the state machine approach and the primary–backup approach, the goal is to provide clients with the illusion of a service that is implemented by a single server. The

[*]Supported in part by an IBM Graduate Fellowship.

[†]Supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, and by grants from IBM T. J. Watson Research Center, IBM Endicott Programming Laboratory, Xerox Webster Research Center and Siemens. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

[‡]Supported in part by the Office of Naval Research under contract N00014-91-J-1219, NSF grant CCR-8701103, DARPA/NSF grant CCR-9014363, and by a grant from IBM Endicott Programming Laboratory.

[§]Supported in part by NSF grants CCR-8901780 and CCR-9102231 and by a grant from IBM Endicott Programming Laboratory.

approaches differ in how each handles failures. With the state machine approach, the effects of failures are completely masked by voting, and the resulting service is indistinguishable from a single non-faulty server. With the primary–backup approach, a request can be lost—additional protocols must be employed to retry such lost requests. The primary–backup approach, however, involves less redundant processing, is less costly, and therefore is more prevalent in practice.

In this chapter, we discuss some fundamental costs that arise in connection with building fault-tolerant services using the primary–backup approach. Here are three key cost metrics of any primary–backup protocol:

- *Degree of Replication:* The number of servers used to implement the service.
- *Blocking Time:* The worst-case period between a request and its response in any failure–free execution.
- *Failover Time:* The worst-case period during which requests can be lost because there is no primary.

The fundamental question, then, is:

> 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 this question defines lower bounds for the degree of replication, the blocking time and the failover time of primary–backup protocols, where a *lower bound* defines a necessary cost that any protocol purporting to solve a problem must incur. Nobody will ever design a protocol with lower cost than the lower bound. Knowing lower bounds for a problem, therefore, gives a basis to evaluate the quality of a protocol.

The existence of a lower bound does not imply the existence of a protocol having this cost, however. Every problem has a trivial lower bound (*viz* 0). We desire lower bounds that are tight—lower bounds that are actually achieved by some protocol. The cost of running any protocol defines an *upper bound* for the problem that protocol solves. If this upper bound is the same as the lower bound, then the lower bound is *tight* and the protocol is *optimal*. (The lower bound implies that a cheaper protocol cannot exist.) Thus, given a problem, we strive to identify tight lower bounds and optimal protocols.

In this chapter, we present and discuss lower and upper bounds for the above three questions. However, deriving these bounds requires a precise specification of the problem solved by a primary–backup protocol. We give this in Sections 2 and 3. Section 4 contains our lower bounds and Section 5 contains our upper bounds. Finally, in Section 6 we discuss some existing primary–backup protocols in terms of our specification, model, and bounds.

2 Specification of Primary–Backup

Informally, a service that is implemented using the primary–backup approach consists of a set of servers, of which no more than one is the *primary* at any time. A client sends a

request to the service by sending it to the server it believes to be the primary. A client learns from the service when the primary changes and directs future requests accordingly. We assume that a client can send any request to the service at any time.

More precisely, we require that to be a primary–backup protocol, four properties must be satisfied. 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). We can now formally define the *failover time* of a primary–backup service to be the longest period of time during which $Prmy_s$ is not true for any s.

Property Pb2 distinguishes the primary–backup approach from the state machine approach. In the latter, each client broadcasts its request to all the servers (at considerable cost).

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

We assume that requests sent to a server s are enqueued in a *message queue* at s. Property Pb3 says that requests that arrive at a backup are ignored.

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

It may appear that Pb1 and Pb3 eliminate the need for Pb2. This is not the case. Pb1–Pb3 ensure that no more than one server can enqueue each client request. Eliminating Pb2 would allow protocols in which this could be violated. In such a protocol, a client would send its request to multiple servers. And, if this request were sent while the identity of the primary is changing, then the request could get enqueued at more than one server. Some of our lower bounds do not hold for such protocols.

Properties Pb1–Pb3 specify a protocol for client interactions with a service but not the obligations of the service. For example, Pb1–Pb3 do not rule out a primary that ignores all requests. We now give a fourth property that precludes such trivial implementations.

For simplicity, we assume that every request requires a response to be sent. Consider a service that is implemented by a single server. Define a *server outage* to occur at time t in this service if some correct client sends a request at time t to the service but does not receive a response. The above server is called a (k, Δ)–*bofo server* (bounded outages, finitely often) if all server outages can be grouped into at most k intervals of time, with each interval having length at most Δ. For example, the computation shown in Figure 1 shows two server outages at times t_1 and t_2 respectively. If the length of this outage period is F and if this is the only such period, then the server is $(1, F)$–bofo. Thus, even though some requests made to a (k, Δ)–bofo server can be lost, the number of such requests is bounded. The fourth property states that a group of servers implementing a service using the primary–backup approach behave as a single bofo server implementing the same service:

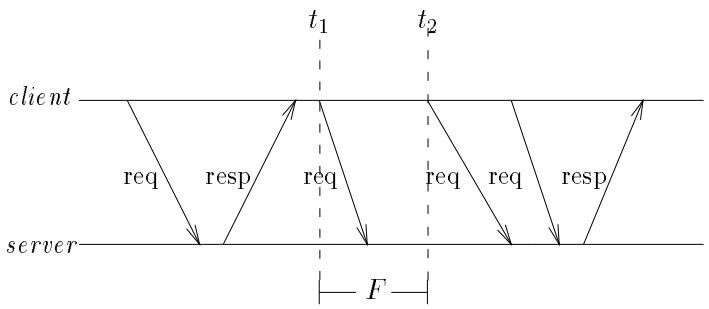

Figure 1: Service Outages

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

Pb4 implies that primary–backup protocols can be used only to implement services that tolerate a bounded number of failures over their lifetime. In practice, we can implement a service that tolerates an unbounded number of failures by partitioning its operation into periods. Only a bounded number of failures occur in each period, and repaired servers are reintegrated at period boundaries. Pb4 need only hold throughout each period. We do not discuss reintegration in this chapter.

2.1 A Simple Primary–Backup Protocol

As an example of a primary–backup protocol, here is one that tolerates the crash of a single server. Assume that all communication is over point-to-point non-faulty links and that each link has an upper bound δ on message delivery time. An execution of this protocol is shown in Figure 2. There exists a primary server p_1 and a backup server p_2 connected by a communications link. A client c initially sends a request to p_1 (indicated by the arrow labeled 1 in the figure). Whenever p_1 receives a request, it

- Processes the request and updates its state accordingly.
- Sends information about the update to p_2 (message 2 in the figure). We call such a message a *state update* message.
- Without waiting for an acknowledgement from p_2, sends a response to the client (message 3 in the figure).

The order in which messages 2 and 3 are sent is important because it guarantees that, given our assumption about failures, if the client receives a response, then either p_2 will eventually receive the state update message or p_2 will crash.

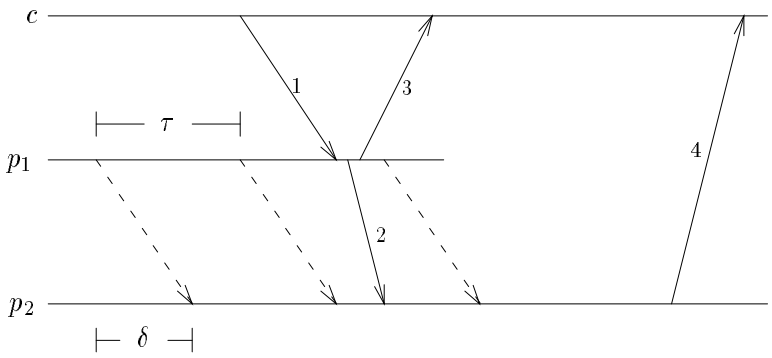

Figure 2: A Simple Primary–Backup Protocol

Server p_2 updates its state upon receiving a state update message from p_1. In addition, p_1 sends dummy messages to p_2 every τ seconds (shown by the dashed arrows in the figure). If p_2 does not receive such a message for $\tau + \delta$ seconds, then p_2 becomes the primary. Once p_2 has become the primary, it informs the clients (message 4 in the figure) and begins processing subsequent requests from the clients.

We now show that this protocol satisfies our characterization of a primary–backup protocol. Property Pb1 requires that there never be two primaries. This is satisfied by the following definitions of *Prmy*:

$$\begin{aligned} Prmy_{p_1} &\stackrel{\text{def}}{=} p_1 \text{ has not crashed} \\ Prmy_{p_2} &\stackrel{\text{def}}{=} p_2 \text{ has not received a message from } p_1 \text{ for } \tau + \delta \end{aligned}$$

Predicate $Prmy_{p_1} \wedge Prmy_{p_2}$ is always false in a system executing our protocol, hence Pb1 is satisfied. The failover time is the longest interval during which $\neg Prmy_{p_1} \wedge \neg Prmy_{p_2}$ can hold. In this protocol, this interval occurs when p_1 crashes immediately after sending a message to p_2 which takes δ to arrive, and so the failover time is $\tau + 2\delta$ seconds.

Property Pb2 follows trivially from the description of the protocol, and Pb3 holds because requests are not sent to p_2 until after p_1 has failed.

Finally, Pb4 requires that the protocol implement a single (k, Δ)–bofo server. To implement a single server, we require that if the primary updates its state and sends a response to a client, then any backup that might later become the primary knows about this state update. In our protocol, this is ensured since p_1 sends the state update message to the backup before it sends any response. To compute k, note that there is at most one switch of the primary, so there is at most one outage period: $k = 1$. To compute Δ, it suffices to compute the longest interval during which a client request may not elicit a response. Assume that p_1 crashes at time t_c. Thus any client request sent to p_1 at $t_c - \delta$ or later may be lost. Furthermore, p_2 may not learn about p_1's crash until $t_c + \tau + 2\delta$, and clients may not learn that p_2 is the

primary for another δ. So, the total period during which a request may not elicit a response is $t_c - \delta$ through $t_c + \tau + 3\delta$; the protocol implements a single $(1, \tau + 4\delta)$–bofo server.

3 System Model

Consider a system consisting of n servers and a set of clients, where server clocks are synchronized arbitrarily close to real time. We assume that clients and servers communicate by exchanging messages through a completely connected point-to-point network and that there is exactly one FIFO link between any two processes. Messages are enqueued in a message queue maintained by the receiving process, and the receiver accesses this queue by executing a **receive** statement. Furthermore, we assume that there is a known constant δ such that if processes p_i and p_j are connected by a (non-faulty) link, then a message sent from p_i to p_j at time t will be enqueued in p_j's queue at or before $t + \delta$.

For computing lower bounds on protocol execution, it is best to assume that a server can compute the response to a request in an arbitrarily short time. While not realistic for many servers, the resulting bounds reflect only the cost of the primary–backup protocols. And that is what we seek.

Execution of a system is modeled by a *run*, which is a sequence of events involving clients, servers, and message queues (see Chapter 4). These events are timestamped with the real time that each event occurs. The events include: sending a message, enqueuing a message, receiving a message, and computation at a process. Two runs σ_1 and σ_2 of the system are defined to be *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 p and p has the same initial state in both runs, then at any time t the state of p at t in σ_1 is the same as the state of p 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 assume that server and link failures occur independently and consider the following hierarchy of failure models:

Crash failures: A server may fail by halting prematurely. Until it halts, the server behaves correctly; once it halts, it never recovers [9].[1]

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 [11].

Send-Omission failures: A server may fail not only by crashing, but also by omitting to send some messages over a non-faulty link [8].

[1] The lower bounds for crash failures also hold for fail-stop failures [12] except for the bound on failover time. The lower bound on failover time depends on the maximum interval between when a server fails and when this failure is recognized by the other servers.

General-Omission failures: A server may exhibit send-omission and receive-omission failures [11].

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.

Note that crash+link failures and the various classes of omission failures are quite different. All admit loss of messages, but each class is handled by a different masking technique: link failures are tolerated by replicating links and omission failures are tolerated by replicating servers. Receive-omission failures model problems at a server, such as the failure to receive messages from the network due to high transfer rates or insufficient buffers. When such problems occur, sending messages using multiple links is not a remedy. Link failures, on the other hand, model problems in the network, such as congestion at a bridge or problems with the physical medium itself. Sending messages over multiple independent links is an effective way to remedy such problems.

4 Lower Bounds

For each failure model, we now give the lower bounds on the degree of replication, blocking time, and failover time for any primary–backup protocol.

4.1 Bounds on Replication

The following table summarizes the lower bounds on the degree of replication. Recall, n is the total number of servers and f is the maximum number of faulty components to be tolerated.

Failure Model	Degree of Replication
crash	$n > f$
crash+link	$n > f + 1$
receive-omission	$n > \lfloor \frac{3f}{2} \rfloor$
send-omission	$n > f$
general-omission	$n > 2f$

For crash failures and send-omission failures, the lower bound is $n > f$. In fact, this is a lower bound for all failure models, since if $n \leq f$ failures could occur, then all of the servers could crash, leaving no primary. A system that has no primary violates Pb4 since a primary is required for the service to behave like a single (k, Δ)–bofo server. For the other failure models, however, $n > f$ is not sufficient as no protocol can achieve this lower bound. As argued below, in the absence of further replication under these failure models, the servers can be divided into mutual non-communicating partitions. Neither partition can tell if the servers in the other partitions have crashed or not. Thus, each partition must eventually contain a primary so that Pb4 is not violated. Yet, if the other partitions have not crashed, then Pb1 will be violated.

The lower bounds on replication for crash+link failures and receive-omission failures are based a further (reasonable) assumption about primary–backup protocols. Let ? 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. For example, in the protocol of Section 2.1, $D = \delta$. For deriving our lower bounds, we assume that if ? is bounded then $D <$?, 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. If one assumes that ? is bounded and $D \geq$?, then protocols that require less replication can be constructed.[2]

We now informally derive the lower bounds for crash+link failures, receive-omission failures and general-omission failures. The arguments are not rigorous; for example, the assumption regarding D and ? is apparently not needed. However, the structure of our detailed formal proofs for the lower bounds in [7] parallel the arguments below.

For crash+link failures, the need for an additional server (*i.e.* $n > f + 1$) is illustrated by the counterexample shown in Figure 3. Assume that $n = f + 1$ and divide the n servers

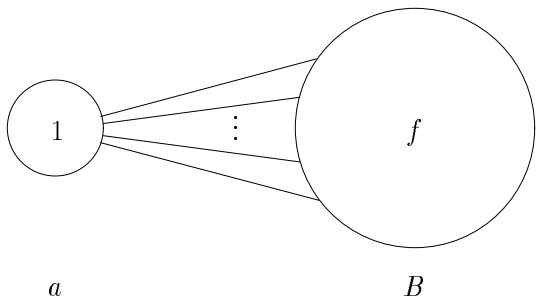

Figure 3: Crash+Link Failures

into a single server a and set B containing f servers. If, in some run, all of the servers in B crash, then a must eventually be the primary since otherwise there would be no primary. Similarly, if a crashes in some run, then a server in B must eventually become the primary. However, there are only f links between a and the servers in B, so all of the links can fail. If this occurs in some run, then to a this run is indistinguishable from the run in which all the servers in B crashed, and to B this run is indistinguishable from the run in which server a crashed. Hence, a will eventually become the primary and some process in B will eventually become a primary, violating Pb1. Note that this counterexample is not possible if $n > f + 1$, because then at least one link between a and B must remain non-faulty. *I.e.* with $n > f + 1$, two servers are always connected through some path of non-faulty links

[2]When $D \geq \Gamma$, there is a primary–backup protocol that tolerates a single crash+link failure using only two servers. In this protocol, when the backup stops receiving messages from the primary, it uses the client requests to distinguish between the primary having crashed and the link having failed.

and servers, so no partition can occur.

The need for still more replication (*i.e.* $n > \lfloor 3f/2 \rfloor$) in the face of receive-omission failures is illustrated in Figure 4. Assume that $n = \lfloor 3f/2 \rfloor$, the servers are divided into sets A and

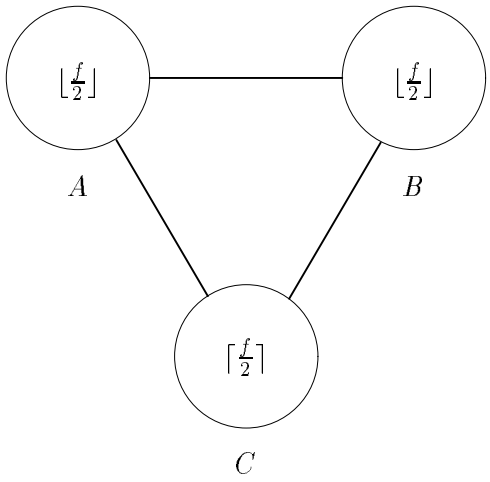

Figure 4: Receive-Omission Failures

B that each contain $\lfloor f/2 \rfloor$ servers, and a set C contains $\lceil f/2 \rceil$ servers. If all the servers in sets A and C crash in some run, then eventually a server in B will become the primary. Similarly, if in some run all the servers in B and C crash, then eventually a server in A will be the primary. However, if in some run all of the servers in A and B commit receive-omission failures for messages sent from outside their respective partitions, then this run is indistinguishable from the first run to the servers in B and is indistinguishable from the second run to the servers in A. Hence, there will eventually be a primary in A and a primary in B, violating Pb1. Finally, the need for replication degree $n > 2f$ to tolerate

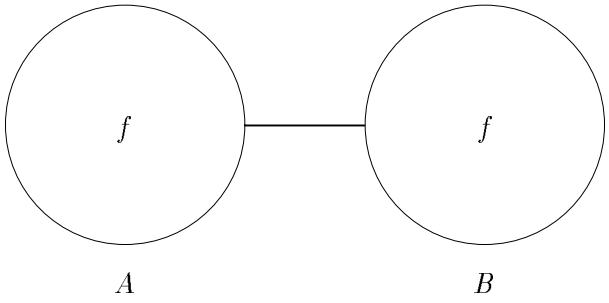

Figure 5: General-Omission Failures

general-omission failures is illustrated in Figure 5. Assume that $n = 2f$ and the servers are divided into sets A and B, each containing f servers. If all the servers in A crash in some run, then eventually a server in B will become the primary, and similarly if the servers in B crash in some run, then eventually a server in A will become the primary. However, if the servers in A commit general-omission failures for messages exchanged with servers in B (*i.e.* they omit to send messages or receive messages from severs in B), then this run is indistinguishable from the first run to the servers in B, and is also indistinguishable from the second run to the servers in A. Hence, there will eventually be a primary in A and a primary in B, violating Pb1.

4.2 Bounds on Blocking Time

The following table summarizes our lower bounds on the blocking time.

Failure Model	Blocking Time
crash	0
crash+link	0
receive-omission	δ when $f = 1$ and $n = 2$ 2δ when $f > 1$ and $n \leq 2f$ 0 when $n > 2f$
send-omission	δ when $f = 1$ 2δ when $f > 1$
general-omission	δ when $f = 1$ 2δ when $f > 1$

Recall that blocking time is the worst-case elapsed time between the receipt of a request *req* and the sending of the associated response *resp* in a run that contains no failures. As can be seen from the table, the values of the lower bounds for blocking time are 0, δ, or 2δ depending on the failure model, f and n. A value of 0 means that the primary can immediately respond to the client. We call such protocols *non-blocking* [7]. The simple protocol discussed in Section 2.1 is an example of a non-blocking protocol. Non-blocking protocols exist that tolerate crash failures and crash+link failures because in such systems there is always at least one non-faulty path from the primary to the other servers. Once the primary sends a state update message, it can reply to the client immediately because it knows that this message will eventually be received.

In contrast, non-blocking protocols cannot be built for send-omission failures, general-omission failures, and receive-omission failures where $n \leq 2f$. For these failure models, the blocking time is at least δ. For example, with send-omission failures, a primary cannot immediately send the response to a request after sending the state update message, because the primary may commit a send-omission failure in sending that message. Protocols for which the blocking time is δ can be built by making a backup send the response to the client. The primary sends the state update message to a backup b. Once b receives this message, it sends the response to the client. Such δ blocking protocols can, however, tolerate only a single failure ($f = 1$).

If more than one failure can occur ($f > 1$), then both the responding backup and the primary may crash after the response is sent, and because of the failure model being assumed (send-omission, general-omission or receive-omission with $n \leq 2f$) the state update messages may never be received by the other servers. These protocols have a blocking time of at least 2δ—further communication from the backups is needed prior to sending a response to the client. This ensures that the state update message has been received by the other backups as well.

4.3 Bounds on Failover Time

The following table summarizes lower bounds on failover time:

Failure Model	Failover Time
crash	$f\delta$
crash+link	$2f\delta$
receive-omission	$2f\delta$
send-omission	$2f\delta$
general-omission	$2f\delta$

For crash failures, a backup can detect the failure of a primary by the absence of an expected message from the primary. For example, if clocks are synchronized then the primary can send a "I am alive:$\ell\tau$" message to the backups at time $\ell\tau$ for $\ell = 0, 1, 2, \ldots$. 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 $(\ell - 1)\tau \leq t_x \leq \ell\tau$. In this case, the failover time, given a single failure, is at most $\tau + \delta$ and it approaches δ as τ becomes small. Similarly, if the backup that next becomes the primary crashes just before $t_x + \tau + \delta$, then there may not be a primary until $t_x + 2(\tau + \delta)$, and so on. Thus, for crash failures, the failover time can be $f\delta$ (as τ approaches zero). However, to show that this is also a lower bound, a fifth property about primary–backup protocols must be assumed:

Pb5: A correct server that is the primary remains so until there is a failure of *some* server or link.

Pb5 is not a very restrictive, and all existing primary–backup protocols satisfy it. However, if the identity of the primary changes rapidly enough—even when there are no failures—then there exist crash-resilient protocols with failover times that violate the $f\delta$ lower bound.[3]

[3]By violating Pb5, it is possible to construct a protocol where the passage of time, rather than the absencee of a message, is used to transfer the role of primary from one server to another. Having pre-agreed times for transfering the role of primary can, under assumptions (which are rarely satisfied in practice) allow smaller failover times.

5 Upper Bounds

There exist protocols that establish that all but two of our lower bounds are tight. In this section, we describe these protocols informally. The material here summarizes [6].

For crash failures, one can modify the simple protocol given in Section 2.1 to use "I am alive: $\ell\tau$" messages for failure detection (see Section 4.3). We can then extend that in a straightforward manner to tolerate f failures: whenever the primary receives a request from the client, it processes that request, sends the state update message to all the backups, and then sends a response to the client. In case the primary fails, one of the backups becomes the primary using an *a priori* defined order. This protocol uses $f + 1$ servers, so the lower bound on the degree of replication is tight. Furthermore, it is non-blocking and has failover time $f(\delta + \tau)$ for arbitrarily small and positive τ— the lower bounds on blocking time and failover time are tight as well.

According to our lower bounds for replication, in order for any primary–backup protocol to tolerate crash+link failures, an additional server is required. The additional server ensures that even in the presence of f failures, there is at least one non-faulty path between any two servers, where such a path contains zero or more intermediate servers. The protocol for crash failures outlined above can now be modified to tolerate crash+link failures by ensuring that any state update or "I am alive: $\ell\tau$" message that a backup receives is forwarded to the other backups. The forwarding of the messages ensures that at least one copy of a message will get to all the intended receivers, since there is at least one non-faulty path between the sender and the receivers. Thus the protocol masks link failures by sending messages over multiple, independent links. This protocol uses $f + 2$ servers, so our lower bound on the degree of replication is tight. Furthermore, the protocol is non-blocking and has failover time $f(2\delta + \tau)$ for arbitrarily small and positive τ, so the lower bounds on blocking time and failover time are also tight.

Most of the protocols for the various kinds of omission failures can be obtained by applying translation techniques [10] to the protocol for crash failures outlined above. These techniques re-implement the message send and receive routines in such a way that a faulty server can detect its failure to send or receive a message and halt. All of the omission–failure protocols obtained in this fashion have failover time $f(2\delta + \tau)$. Thus, our lower bounds on failover times are tight. The protocol for send-omission failures uses $f + 1$ servers and is $2\delta + \tau$–blocking. Furthermore, a send-omission protocol for $f = 1$ that is δ–blocking has been constructed. Thus, the lower bounds on the degree of replication and blocking time are also tight for send-omission failures. Finally, the protocol for general-omission failures obtained by translation uses $2f + 1$ servers and is 2δ–blocking, so the lower bounds on the degree of replication and blocking time are tight for general-omission failures as well.

For receive-omission failures, it is not known whether our lower bounds are tight for degree of replication or for blocking time when $n \leq 2f$ and $f > 1$. The protocol for this failure model obtained by translation uses $2f + 1$ servers, but the lower bound is $n > \lfloor \frac{3f}{2} \rfloor$. Individual protocols for $n = 2$, $f = 1$ and $n = 4$, $f = 2$ have been constructed, but have not been generalized. However, the protocol for $n = 2$, $f = 1$ is δ–blocking, so the lower bound on blocking time when $n = f$ and $f = 1$ is tight.

The following table summarizes the lower bounds and indicates which of these lower bounds are known to be tight.

Failure Model	Degree of Replication	Blocking Time	Failover Time
crash	$n > f$	0	$f\delta$
crash+link	$n > f + 1$ †	0	$2f\delta$
receive-omission	$n > \lfloor \frac{3f}{2} \rfloor$ * †	δ when $n \leq 2f$ and $f = 1$ † 2δ when $n \leq 2f$ and $f > 1$ * † 0 when $n > 2f$	$2f\delta$
send-omission	$n > f$	δ when $f = 1$ 2δ when $f > 1$	$2f\delta$
general-omission	$n > 2f$	δ when $f = 1$ 2δ when$f > 1$	$2f\delta$

* Bound not known to be tight.
† $D < ?$ assumed.

6 Existing Primary–Backup Protocols

We now discuss some existing primary–backup protocols: the Alsberg and Day protocol [1], the Tandem protocol [3], HA–NFS [4] and an experimental non-blocking protocol [5].

6.1 The Alsberg and Day Protocol

We believe this protocol to be the earliest primary–backup protocol appearing in the literature. It employs two servers and tolerates a single crash failure.[4]

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 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).

[4]The authors also claim that the protocol tolerates network partitions. However, during partitions the primary and the erstwhile backup can diverge, violating Pb4. In the analysis that follows, we assume that partitions do not occur.

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. And, when a primary has no backup (either because the backup crashed or the backup becomes the primary), the primary uses another protocol to recruit another server to become the backup.

The above protocol requires two servers to tolerate a single server crash and has a blocking time of δ. Note that the protocol does not satisfy Pb3. However, for crash failures, our lower bound results do not depend on Pb3, so the 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 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.

6.2 The Tandem Protocol

This protocol 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 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, our crash failure bounds apply to this protocol.[5] The 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.

6.3 HA-NFS

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 maintain Pb1, before it becomes the primary 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.

As with the Tandem protocol, our lower bounds for crash failures apply 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. This time is at least 2δ. The optimality of its failover time remains an open question because this protocol does not assume synchronized clocks (and our bounds do).

[5] We assumed that there is exactly one link between any two processes. In Tandem's 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.

6.4 Non-Blocking Protocol

Non-blocking protocols are of practical interest because they can achieve the fastest possible response times. To see how the response time for these protocols compares with conventional blocking protocols, a non-blocking protocol tolerating receive-omission failures was built [5].

An argument can be made that a receive-omission failure model is the most appropriate one for many environments. A 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. Furthermore, using a single local area network makes partitions that separate the servers unlikely. The kinds of message losses that are expected to occur on this network are restricted and correspond to our receive-omission failure model. According to [2], as technology improves and newer, faster networks such as FDDI are used, the following will be the dominant causes for message losses on a local area network:

- Failure to intercept messages from the network at high transfer rates due to interrupt misses.
- Buffer overflows at the receiver.

This set of failures corresponds to receive-omission failure model, and one can construct a non-blocking primary–backup protocol for this model when $n > 2f$.

We now briefly describe our non-blocking protocol tolerating receive-omission failures. The protocol consists of $2f + 1$ servers, one of which is the primary, and the rest of which are backups. When the current primary receives a client request, it sends the state update message to all the backups and then immediately responds to the client. A backup, upon receiving this message, updates its state. However, it is possible that some backup might experience a receive-omission fault and not receive the state update message. The protocol, therefore, must ensure that this faulty backup does not later become the primary with an out-of-date state. This is achieved by a failure detection scheme in which a faulty server detects its own failure to receive a message and halts. This failure detection technique requires $n > 2f$.

All the servers periodically exchange "I am alive: $\ell\tau$" messages to detect server crashes. The backups are ranked, and if the primary crashes then the backup with the lowest rank takes over as the new primary.

This protocol has failover time $f(2\delta + \tau)$, which is optimal as τ approaches zero, and optimal blocking time of zero. Furthermore, it also has the optimal degree of replication for non–blocking protocols.

When we implemented this protocol on a local area network, we were surprised to find that blocking time is not the dominant factor in determining response time as seen by clients. In particular, our non-blocking protocol generated $O(n^2)$ messages to implement the failure detection. When client requests are made with high frequency, this message traffic led to high contention on our local area network. This bandwidth saturation was the key factor in determining the response time seen by clients.

7 Conclusions

In this chapter, we have given a precise characterization for primary–backup protocols in a system with synchronized clocks and bounded message delays. We then presented lower bounds on the degree of replication, the blocking time, and the failover time under various kinds of server and link failures. We also outlined a set of primary–backup protocols that show which of our lower bounds are tight.

We have attempted to give a characterization of primary–backup that is broad enough to include most synchronous protocols that are considered to be instances of the approach. As we have seen, there are protocols that are incomparable to the class of protocols we analyzed. Some protocols do not assume synchronized clocks; some protocols do not even assume a synchronous system. Possible characterizations for a primary–backup protocol in an asynchronous system is an area under active investigation.

References

[1] 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.

[2] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *FTCS-22 The Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84. IEEE Computer Society Technical Committee on Fault-Tolerant Computing, July 1992.

[3] J.F. Barlett. A nonstop kernel. In *Proceedings of the Eighth ACM Symposium on Operating System Principles, SIGOPS Operating System Review*, volume 15, pages 22–29, December 1981.

[4] Anupam Bhide, E.N. Elnozahy, and Stephen P. Morgan. A highly available network file server. In *USENIX*, pages 199–205, 1991.

[5] Navin Budhiraja and Keith Marzullo. Tradeoffs in implementing primary–backup protocols. Technical Report TR 92-1307, Department of Computer Science, Cornell University, 1992.

[6] Navin Budhiraja, Keith Marzullo, Fred 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.

[7] 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.

[8] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Harvard University, June 1984. Department of Computer Science Technical Report 11-84.

[9] Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Op. 62, SRI International, April 1982.

[10] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[11] 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.

[12] 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.