Primary-Backup Protocols: Lower Bounds and Optimal Implementations

Navin Budhiraja*
Keith Marzullo*
Fred B. Schneider**
Sam Toueg***

TR 92-1265
January 1992

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

*Supported by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 and by grants from IBM 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, the National Science Foundation under Grant No. CCR-8701103, DARPA/NSF Grant No. 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.

Primary–Backup Protocols: Lower Bounds and Optimal Implementations

Navin Budhiraja*
Keith Marzullo*
Fred B. Schneider†
Sam Toueg‡

Department of Computer Science
Cornell University
Ithaca NY 14853, USA

Abstract

We present a precise specification of the primary–backup approach. Then, for a variety of different failure models we prove lower bounds on the degree of replication, failover time, and worst-case blocking time for client requests. Finally, we outline primary–backup protocols and indicate which of our lower bounds are tight.

Keywords: Fault-tolerance, reliability, availability, primary–backup, lower bounds, optimal protocols.

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 all client requests, in the same order, to all non-faulty servers. This service architecture is commonly called *active replication* or *the*

*Supported by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593 and by grants from IBM, Siemens, and Xerox. Budhiraja is also supported by an IBM Graduate Fellowship. 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, the National Science Foundation under Grant No. CCR-8701103, DARPA/NSF Grant No. 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.

state machine approach [22] and has been widely studied from both theoretical and practical viewpoints (*e.g.*, [9, 11, 19]).

The other approach to building replicated services 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. However, the approach has not been analyzed nearly as extensively as the state machine approach. Little is known of its costs and tradeoffs, the degree of replication required, or the worst-case response time for various failure models. In this paper, we derive some of these tradeoffs. For example, in some primary–backup protocols [15] the number of servers used is more than twice the number of failures to be tolerated. We are now able to explain this phenomenon by showing that the number of servers needed depends on the failure model.

With both active replication and the primary-backup approach, the goal is to provide a client with the illusion of a service that is implemented by a single server, despite failures. The key difference between active replication and the primary-backup is how each handles failures. With active replication, the effects of failures are completely masked by voting, and the service implemented is indistinguishable from a single non-faulty server. With the primary-backup approach, a request to the service can be lost if it is sent to a faulty primary.[1] Thus, clients can observe the effects of failures. However, the periods during which requests can be lost are bounded by the length of time that can elapse between failure of the primary and takeover by a backup. Such behavior is an instance of what we call *bofo* (*bounded outage finitely often*).

To formulate the notion of a bofo server, define a *server outage* to occur at time t if some client makes a server request at that time but never receives a response to that request.[2] In a (k, Δ)*–bofo server*, all server outages can be grouped into at most k intervals of time, with each interval having length at most Δ. Accordingly, even though some requests made to a bofo service (that is, a service that implements the abstraction of a bofo server) will be lost, this number is limited. Note that if clients of a service are restricted to send requests only to one server, then it is not possible to implement a specification that is stronger than bofo. This is because if the client sends a request to a (single) server and that server subsequently crashes, then the request can be lost and will not be processed.

This paper gives lower bounds for various costs associated with implementing a bofo service by using the primary-backup approach. These lower bounds depend on message delivery delay and the class of failures to be tolerated. These bounds characterize the degree of replication, the time during which the service can be without a primary, and the amount of time it can take to respond to client requests (blocking time). In some cases, our results are surprising. For example, more than $f + 1$ servers are necessary to tolerate f failures of certain types (crash and link failures, receive-omission failures, or general-omission failures). Also, we have proved that if a majority of the servers can be faulty, then any primary–backup protocol for receive–omission failures will have a run in which a non-faulty primary is is forced to let a faulty server become the primary in its place. Finally, we outline some

[1] Of course, the client can subsequently resend a copy of that request to the new primary.

[2] For simplicity, we assume in this paper that every request elicits a response.

primary–backup protocols. This allows us to determine which of our lower bounds are tight.

The paper is organized as follows. Section 2 gives a precise specification of the primary-backup approach. Section 3 describes the system model we consider. Section 4 discusses lower bounds, and in Section 5 we outline our protocols and state which of our bounds are tight. We conclude in Section 6.

2 Specification of the Primary–Backup Approach

Since we wish to derive lower bounds, we must first give a precise specification of primary-backup that is general enough to satisfy any protocol one would characterize as being primary-backup. The following four properties do this.

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 $Prmy_s$.[3]

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 define the *failover time* of a 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 active replication, where each client broadcasts its request to all the servers.

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

We assume that requests sent to a server s are enqueued in a *message queue* at s.

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).

Properties Pb1–Pb3 specify a protocol for client interactions with a service, but not the obligations of the service. For example, the properties do not rule out a primary that ignores all requests. A fourth property eliminates such trivial implementations by stipulating that the service implements a single bofo server for some values of k and Δ:

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

We believe that the above four properties characterize a primary-backup approach and have checked that many primary-backup protocols in the literature (*e.g.* [1, 3, 4, 7]) do satisfy this characterization.

Note that Pb4 is not implementable if the number of failures (that is, the number of servers and communication components that fail) can not be bounded *a priori*. This is because an unbounded number of servers would be required to implement the service. In a practical system, one can implement service outages of bounded lengths by bounding the rate of failures and allowing reintegration of recovered servers and communication links. We do not address failure rates or reintegration in this paper.

[3]The protocol of [15] allows concurrent primaries, but only for bounded periods. If one replaces Pb1 by this weaker property, then except for the bounds on failover times, the bounds shown in Section 4 continue to hold.

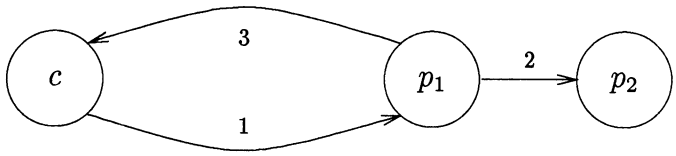

Figure 1: A Simple Primary–Backup Protocol.

2.1 A Simple Primary–Backup Protocol

As an example of a service based on the primary–backup approach, consider the following protocol, which tolerates crash failures 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.[4] Refer to Figure 1. There is a primary server p_1 and a backup server p_2 connected by a communications link. A client C initially sends all requests 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),
- without waiting for an acknowledgement from p_2, sends a response to the client (message 3 in the figure).

The order in which these messages 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 message 2 or p_2 will crash.

Server p_2 updates its state upon receiving messages from p_1. In addition, p_1 sends dummy messages to p_2 every τ seconds. 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 (who update their copies of *Dest*) and begins processing subsequent requests from the clients.

We now show how 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*:

$$Prmy_{p_1} \stackrel{\text{def}}{=} (p_1 \text{ has not crashed})$$
$$Prmy_{p_2} \stackrel{\text{def}}{=} (p_2 \text{ has not received a message from } p_1 \text{ for } \tau + \delta)$$

Predicate $Prmy_{p_1} \wedge Prmy_{p_2}$ is always false in a system executing our protocol, and hence Pb1 is satisfied. The *failover time* for this protocol is the longest interval during which $\neg Prmy_{p_1} \wedge \neg Prmy_{p_2}$ can hold, and it is $\tau + 2\delta$ seconds. Property Pb2 follows trivially from

[4] To simplify exposition, we assume that the maximum message delay between the clients and the servers is the same as the delay between the servers. However, our results can be easily extended to the case when the delays are different.

the description of the protocol. Property Pb3 is true because requests are not sent to p_2 until after p_1 has failed. Finally, Pb4 requires that the protocol implements a single bofo server for some values of k and Δ. Since p_1 sends message 2 before message 3, it will never be the case that p_1 sends a response to the client and p_2 does not get information about that response from p_1. In this protocol, there is at most one switch of the primary. So there is at most one outage period *i.e.* $k = 1$. 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 since p_1 crashes at t_c. 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 The Model

We consider a system consisting of n servers and a set of clients. We assume that server clocks are perfectly synchronized with real time.[5] Clients and servers communicate by exchanging messages through a completely connected point-to-point network.[6] Each message sent is enqueued in a queue maintained by the receiving process, and a process accesses its message queue by executing a **receive** statement. We assume that links between processes are FIFO (*i.e.* if p_i sends message m followed by m' to process p_j, then p_j will never receive m after m') and 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 of before $t + \delta$.

We are interested in identifying the costs inherent in primary–backup protocols, and so we assume that it takes no time for a server to compute a response. Our theorems characterize lower bounds; they are not invalidated by servers that require a substantial amount of time to compute a response.

We model an execution of a system by a *run*, which is a sequence of timestamped events involving clients, servers, and message queues. These events include sending messages, enqueuing messages, receiving messages, and internal events that model computation at processes. 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 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. It is not hard to extend the definition of indistinguishability to handle nondeterministic servers.

We 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 paper.

---

[5] Our protocols can be extended to the case where clocks are only approximately synchronized [14].

[6] Another approach would be assume that servers are interconnected with redundant broadcast busses [2, 8]. We have not pursued this approach.

Define $\prec$ to be the *potential causality* relation [12] on server events e_1 and e_2. Thus $\prec$ is the transitive closure of the following relation $\rightsquigarrow$: $e_1 \rightsquigarrow e_2$ 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. Informally, we say a request m is an *update request* if it changes the state of the service in such a way that the responses to subsequent requests 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). Then m is an update request if all request/response pairs m'/r', where m' was sent after r, have $e(m) \prec e(r')$. We assume that update requests exist since otherwise the actions performed by the primary do not have to be communicated to the backups.

We assume that failures occur independently from each other. We consider the following hierarchy of failure models:

Crash failures: A server may fail by halting prematurely. Until it halts, it behaves correctly [13]. [7]

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.

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 [10].

General-Omission failures: A server may exhibit send-omission and receive-omission failures [20]).

Note that crash+link failures and the various types of omission failures are quite different. Although all of these failure models concern loss of messages, each class of failures is dealt with by a different masking technique. In particular, crash+link failures can be masked by adding redundant communication paths, while omission failures can only be masked by adding redundant servers so that faulty processes can detect their own failure and halt. We return to these masking techniques in Section 5.

Failures are counted by the number of failing components (either servers or links). We say that a protocol tolerates f failures if it works correctly despite the failure of up to f faulty components (note that each faulty component may fail many times during an execution).

4 Lower Bounds

For each failure model, we now give lower bounds for implementing a single (k, Δ)–bofo server using the primary–backup approach.

[7] The lower bounds we derive for crash failures also hold for fail-stop failures [21] except for the bound on failover time. The lower bound on failover time depends on the maximum duration between when a server p_i fails and when $failed_i$ becomes true.

4.1 Bounds on Replication

The first bound is obvious. However, to introduce our notation and the proof technique that will be used later in the section, we give a formal proof of the theorem.

Theorem 1 *Any primary–backup protocol tolerating f crash failures requires $n \geq f + 1$.*

Proof: We prove the result by contradiction. Suppose there is a protocol P for $n < f+1$. Thus, P satisfies Pb4. Consider a run in which all n servers are crashed initially and clients submit $R > k\lceil \Delta/d \rceil$ requests, where d is the minimum time between the sending of any two requests ($d > 0$). By Pb4, at least one of these requests must elicit a response. 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. However, such a response is impossible since, by assumption, all servers have crashed. □

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

Lemma 4.1 *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 ($d > 0$). Furthermore, 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.*

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.

For σ_a, by Pb4 at least $R - k\lceil \Delta/d \rceil$ of the requests must have been received by servers in A. Similarly, for 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 2 and 3 depend on two parameters of primary–backup protocols. Let Γ be the maximum time that can elapse between any two successive client requests (possibly from different clients), and let D be a duration such that if some server s becomes the primary at time t_0 and remains the primary through time $t \geq t_0 + D$ when a client c_i sends a request,

then $Dest_i = s$ at time t. Hence, D is the minimum delay until all clients know the identity of a new primary. For simplicity of notation, we write $D < \Gamma$ to mean that D is bounded and Γ is either unbounded or bounded and greater than D. Note that when $D < \Gamma$ the service must be able to detect the failure of a primary and disseminate the new primary's identity to the clients without using any messages from clients.

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, as they require more replication.

Theorem 2 *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 $n \geq f + 2$.*

Proof: For contradiction, assume the existence of a protocol P with $n < f + 2$. We will show that P has three runs σ_a, σ_b and σ_{ab} that satisfy the conditions of Lemma 4.1. 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, 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, by construction these requests will arrive at the servers according to the same rules used in constructing σ_{ab}. Of course, a client request may not be delivered to the same servers in σ_a or σ_b as in σ_{ab}, since different servers are operational in these runs.

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 contradiction: let t be the earliest time that s_a can distinguish between these two runs.

Thus, at time t either s_a received a request m in σ_{ab} but not in σ_a or it received a request m in σ_a but not in σ_{ab}. We will assume the former; the proof for the latter is similar. The request m must have been enqueued at some time $t' \leq t$ at s_a in σ_{ab}. Since m was received by 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.

Since s_a cannot distinguish between the runs before t, s_a cannot receive m before t in σ_a, and s_a must execute a receive in both σ_a and σ_{ab} at time t. So, it must be the case that s_a receives another request $m' \neq m$ at time t in σ_a. Assume that m' was enqueued at time t''. By an indistinguishability argument similar to above, m' must be enqueued at time t'' at s_a in σ_{ab} as well. Therefore, if s received m' in σ_a at time t, it must receive m' in σ_{ab} as well, a contradiction.

A similar argument can be used to show that the servers in B receive the same requests in σ_b and σ_{ab}, and so these two runs are indistinguishable to the servers in B. Thus, by Lemma 4.1 P cannot be a primary–backup protocol. □

The assumption in this theorem that $D < \Gamma$ is significant. As we discuss in Section 5, when $D \geq \Gamma$ protocols that tolerate f crash+link failures can be constructed that use only $f + 1$ servers.

The next theorem states that additional replication is required in order to tolerate receive-omission failures. The proof is similar to that of Theorem 2, and so it is omitted.

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

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

Theorem 4 *Any primary–backup protocol tolerating f general-omission failures requires $n > 2f$.*

Proof: Assume for contradiction that there is a protocol for $n \leq 2f$. 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

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. Consider a failure-free run of a primary-backup protocol that is handling a request. Let the time that the request is received be t_m and the time that the response is sent be t_r. We say that this protocol is *C–blocking* if it is guaranteed that $t_r - t_m \leq C$ holds. 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.

As shown in Section 5, 0–blocking primary-backup protocols can be built for crash and crash+link failure models. For servers that take no time to compute the response to a request, the simple protocol tolerating crash failures presented in Section 2 is 0–blocking. We call such protocols *nonblocking* because the primary can send a reply to the client as soon as the reply has been computed. Nonblocking protocols tolerating receive-omission failures also exist as long as $n > 2f$, but there is can be no nonblocking primary–backup protocol tolerating send-omission or general–omission failures.

Theorem 5 *Any primary–backup protocol tolerating f receive-omission failures with $f > 1$, $n \leq 2f$ and $D < \Gamma$ is C–blocking for some $C \geq 2\delta$.*

Proof: For contradiction, suppose there is a primary–backup protocol for $n \leq 2f$ 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 $p \in A$ and r be sent at time t_r by a server $q \in A$ (q could be the same as p). 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 p 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 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 $s \in B$ 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 B at time t. This request arrives immediately by Pb3 will never be enqueued at any server.

3. A server $s \in B$ is the primary at time t but another server $s' \in B$ 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 no the primary, and so will not be enqueued (again, by Pb3).

Notice that eventually, there will be a response (say r') in σ_2 because the protocol satisfies Pb4, 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. 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 $t_r - t_m < 2\delta$, p (the server that received request m at time t_m in σ_1) and q (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 q sends response r in σ_3 as well. Then, using an argument similar to the one in Theorem 2, 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 violates the assumption that m is an update request. □

In run σ_3 of the above proof, a correct primary (p in set A) becomes the backup, while a faulty server from set B becomes the primary in p's place. It is always possible to construct such a run. This is a disconcerting property: there does not exist a primary–backup protocol that tolerates receive-omission failures with $n \leq 2f$ in which a primary cedes only when it fails. Moreover, this lower bound is tight—in [6], we give a receive-omission primary–backup protocol with $n = 2f + 1$ 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 5 (and is therefore omitted), except that $p = q$.

Theorem 6 *Any primary–backup protocol tolerating receive-omission failures with $f = 1$ and $n \leq 2f$ and having $D < \Gamma$ is C–blocking for some $C \geq \delta$.*

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 7 *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 p and r be sent at time t_r by a server q (again q could be p). 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 through t_m. After t_m, p and q fail and omit to send all messages to all servers except each other. Since, by construction, all messages take δ to arrive, servers and clients cannot distinguish between σ_1 and σ_2 through $t_m + \delta$ and, as a result, p and q cannot distinguish the two runs through $t_m + 2\delta$. Therefore, since $t_r - t_m < 2\delta$, q sends the response r at time t_r in σ_2 as well. Now let p and q crash at time t_r and the clients send requests after time t_r. By Pb4, there eventually must be some request m' that results in a response r'. However, $\neg(e(m) \prec e(r'))$, which violates the assumption that m us an update request. □

Again, if $f = 1$, then the following theorem can be proved using a proof similar to Theorem 7, except that $p = q$.

Theorem 8 *Any primary–backup protocol tolerating send-omission failures with $f = 1$ is C–blocking for some $C \geq \delta$.*

4.3 Bounds on Failover Times

Recall that the failover time is the longest interval during which $Prmy_s$ is not true for any server s. In this section, we give lower bounds for failover times.

In order to discuss these bounds, we postulate a fifth property of primary–backup protocols.

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

This is a reasonable expectation and it is valid for all protocols that we have found in the literature.

Theorem 9 *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 p_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 p_1 crashes at time t_1. By assumption, there exists a new primary (say p_2) at time $t_2 < t_1 + \delta$. Since p_1 crashes at time t_1, p_2 does not receive any messages from p_1 that were sent after time t_1.

σ_2: Assume that p_1 is correct, there are no other crashes at or after t_1 and all messages sent by p_1 after time t_1 take δ to arrive.

Since p_2 cannot distinguish σ_1 from σ_2 through time t_2, p_2 becomes the primary at time t_2 in σ_2. By Pb5, however, p_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 10 *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 p_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 p_1 crashes at time t_1 and all messages sent after t_1 take δ to arrive. Furthermore, the crash of p_1 is the only failure at or after t_1. By assumption, there exists a new primary (say p_2) at time $t_2 < t_1 + 2\delta$. Since p_1 crashes at time t_1, p_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 after $t_1 + \delta$ can be received by p_2 only after time t_2.

σ_2: Assume that p_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 p_1 and p_2 is faulty and does not deliver any message sent by p_1 to p_2 after time t_1.

By construction, p_2 cannot distinguish σ_1 from σ_3 through time t_2, and so p_2 becomes the primary at time t_2 in σ_3. Similarly, p_1 cannot distinguish σ_2 from σ_3 through time t_2 and so p_1 remains the primary until time t_2 in σ_3. This violates Pb1, and so P is not a primary–backup protocol. □

We omit the proofs of the following two theorems because they are similar to Theorem 9.

Theorem 11 *Any primary–backup protocol tolerating f receive-omission failures has a failover time of at least $2f\delta$.*

Theorem 12 *Any primary–backup protocol tolerating f send-omission failures has a failover time of at least $2f\delta$.*

5 Outline of the Protocols

In order to establish that the bounds given above are tight, we have developed primary–backup protocols for the different failure models [6]. In this section, we outline these protocols and discuss which of our lower bounds are tight.

Our protocol for crash failures is similar to the protocol given in Section 2. Whenever the primary receives a request from the client, it processes that request and sends information about state updates to the backups before sending a response to the client. All servers periodically send messages to each other in order to detect server failures. This protocol uses $(f + 1)$ servers and so the lower bound in Theorem 1 is tight. Furthermore, it is nonblocking and so incurs no additional delay. It has the failover time $f\delta + \tau$ for arbitrarily small and positive τ, and so the lower bound in Theorem 9 is tight.

In order for the protocol to tolerate crash+link failures, we add an additional server. By Theorem 2, this server is necessary. The additional server ensures that there is always at least one non-faulty path between any two correct servers, where a path contains zero or more intermediate servers. The protocol for crash failures outlined above is now modified so that a primary ensures any message sent to a backup is sent across at least one non-faulty path. This protocol uses $(f + 2)$ servers and so Theorem 2 is tight. Furthermore, it is nonblocking and so incurs no additional delay. It has the failover time $2f\delta + \tau$ for arbitrarily small and positive τ, and so Theorem 10 is tight.

Most of our protocols for the various kinds of omission failures apply translation techniques [17] to the protocol for crash failures outlined above. These techniques ensure that a faulty server detects its own failure and halts, thereby translating a more severe failure to a crash failure. The translations of [17] assume a round-based protocol. Since our crash failure protocol is not round-based, we must modify the translations so that a server can send and receive messages at any time rather than just at the beginning or the end of a round. All of these resulting omission–failure protocols have failover time $2f\delta + \tau$, and thus Theorems 11 and 12 are tight. The protocol for send-omission failures uses $f + 1$ servers and is $2\delta + \tau$–blocking. Furthermore, we also have a send-omission protocol for $f = 1$ that is δ–blocking. Thus, Theorems 7, 8 and 12 are tight. The protocol for general-omission failures also uses $2f + 1$ servers and is $2\delta + \tau$–blocking, and so Theorem 4 is tight, and Theorems 7 and 12 are tight for general-omission failures as well.

We have not been able to determine whether Theorems 3 and 5 are tight. Our protocol for receive-omission failures uses $2f + 1$ servers whereas the lower bound in Theorem 3 only requires $n > \lfloor \frac{3f}{2} \rfloor$. We have constructed protocols for $n = 2$, $f = 1$ and $n = 4$, $f = 2$ but are unable to generalize these protocols. We can also show that any protocol for $n \leq 2f$ has the following odd property: there is at least one run of the protocol in which a non-faulty primary is forced to relinquish control to a backup that is faulty. However, the protocol for $n = 2$, $f = 1$ is δ–blocking and so Theorem 6 is tight.

Table 1 summarizes all of our results.

failure model	degree of replication	amount of blocking	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 if $n \leq 2f$ and $f = 1$ † 2δ if $n \leq 2f$ and $f > 1$ * † 0 if $n > 2f$	$2f\delta$
send omission	$n > f$	δ if $f = 1$ 2δ if $f > 1$	$2f\delta$
general omission	$n > 2f$	δ if $f = 1$ 2δ if$f > 1$	$2f\delta$

* Bound not known to be tight.
† $D < \Gamma$.

Table 1: Lower Bounds.

6 Discussion

We give a precise characterization for primary–backup protocols in a system with synchronized clocks and bounded message delays. We then present lower bounds on the degree of replication, the blocking time, and the failover time under various kinds of server and link failures. We finally outline a set of primary–backup protocols that show which of our lower bounds are tight.

We now briefly compare our results to existing primary–backup protocols. The protocol presented in [3] tolerates one crash+link failure by using only two servers. This appears to contradict Theorem 2 which states that at least three servers are required to tolerate one failure. However, the protocol in [3] assumes that there are two links between the two servers, effectively masking a single link failure. Hence, only crash failures need to be tolerated, and this can be accomplished using only two servers (Theorem 1).

A more ambitious primary–backup protocol is presented in [15]. This protocol works for the following failure model (quoted from [15]):

> The network may lose or duplicate messages, or deliver them late or out of order; in addition it may partition so that some nodes are temporarily unable to send messages to some other nodes. As is usual in distributed systems, we assume the nodes are fail-stop processors and the network delivers only uncorrupted messages.

This failure model is incomparable with those in the hierarchy we presented. However, the protocol does tolerate general-omission failures and has optimal degree of replication for general-omission failures as it uses $2f + 1$ servers.

In Theorem 2, we assumed that $D < \Gamma$. This assumption is crucial: we are able to construct a two-server primary–backup protocol tolerating one crash+link failure for which $D \geq \Gamma$. Recall that link failures are masked by adding redundant paths between the servers.

Our two-server crash+link protocol essentially uses the path from the primary to the backup through the client as the redundant path. Thus, there appears to be a tradeoff between the degree of replication and the time it takes for a client to learn that there is a new primary.

The lower bounds on failover times given in Section 4.3 assume Pb5. This is necessary as we have constructed protocols that have failover times smaller than the lower bounds given in Section 4.3 and these protocols do not satisfy Pb5. This smaller failover time is achieved at a cost of an increased variance in service response time.

Finally, in this paper 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. There are protocols, however, that are incomparable to the class of protocols we analyze [5, 16, 18] since they were developed for an asynchronous setting. Such protocols cannot be cast in terms of implementing a (k, Δ)–bofo server for finite values of k and Δ. We are currently studying possible characterizations for a primary–backup protocol in an asynchronous system and hope to extend our results to this setting.

Acknowledgements

We would like to thank Lorenzo Alvisi, Mike Reiter and the anonymous conference referees for their helpful comments on earlier drafts of this paper.

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] Özalp Babaoğlu and Rogério Drummond. Streets of Byzantium: Network architectures for fast reliable broadcasts. *IEEE Transactions on Software Engineering*, 11(6):546–554, June 1985.

[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] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Eleventh ACM Symposium on Operating System Principles*, pages 123–138, November 1987.

[6] Navin Budhiraja, Keith Marzullo, Fred Schneider, and Sam Toueg. Optimal primary–backup protocols. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, Haifa, Israel, November 1992. To Appear.

[7] IBM International Technical Support Centers. IBM/VS extended recovery facility (XRF) technical reference. Technical Report GG24-3153-0, IBM, 1987.

[8] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Journal of Real-Time Systems*, 2:195–212, September 1990.

[9] 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.

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

[11] Thomas Joseph and Kenneth Birman. *Reliable Broadcast Protocols*, pages 294–318. ACM Press, New York, 1989.

[12] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[14] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[15] 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, 1991.

[16] Timothy Mann, Andy Hisgen, and Garret Swart. An algorithm for data replication. Technical Report 46, Digital Systems Research Center, 1989.

[17] 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.

[18] 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.

[19] M. Pease, R. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[20] 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.

[21] 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.

[22] Fred B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299–319, December 1990.