Optimal Primary-Backup Protocols

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

Department of Computer Science, Cornell University, Ithaca NY 14853, USA

Abstract. We give primary–backup protocols for various models of failure. These protocols are optimal with respect to degree of replication, failover time, and response time to client requests.

1 Introduction

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

A common approach to structuring such replicated services is to designate one site 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].

In [5] we give lower bounds for implementing primary-backup protocols under various models of failure. These lower bounds constrain 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 a client request. In this paper, we show that most of these lower bounds are tight by giving matching protocols.

Some of the protocols that we describe have surprising properties. In one case, the optimal protocol is one in which a non-faulty primary is forced to relinquish control to a backup that it knows to be faulty! However, the existence of such a scenario is not peculiar to our protocol. As shown in [5], relinquishing control to a faulty backup is indeed necessary to achieve optimal protocols in some failure models. Another surprise is that in some protocols that achieve optimal response time, the site that receives the request (*i.e.* the primary) is not the site that sends the response to the clients. We show that this anomaly is not idiosyncratic to our protocols—it is necessary for achieving optimal response time.

* Supported by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593 and by grants from IBM and Siemens.

** 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.

The rest of the paper is organized as follows. Section 2 gives a specification for primary-backup protocols, Sect. 3 discusses our system model, Sect. 4 summarizes the lower bounds from [5], and Sect. 5 summarizes our results. Sections 6, 7 and 8 describe the protocols that achieve our lower bounds, and Sect. 9 describes a protocol in which the primary is forced to relinquish control to a faulty backup. We conclude in Sect. 10. Due to lack of space, the description of some of the protocols and all proofs are omitted from this paper. See [4] for a complete description and proofs.

2 Specification of Primary-Backup Services

Our results apply to any protocol that satisfies the following four properties, and many primary-backup protocols in the literature (*e.g.* [1,2,3]) do satisfy this characterization.

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

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

For the next property, we model a communications network by assuming that client requests are enqueued in a message queue of a site.

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

A request sent to a primary–backup service can be lost if it is sent to a faulty primary. Periods during which requests are lost, however, are bounded by the time required for a backup to take over as the new primary. Such behavior is an instance of what we call *bofo* (*bounded outage finitely often*). We say that an *outage* occurs at time t if some client makes a request at that time but does not receive a response[1]. A (k, Δ)*-bofo server* is one for which all outages can be grouped into at most k periods, each period having duration of at most Δ.[2]. The final property of the primary–backup protocols is that they implement a bofo–server (for some values of k and Δ).

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

Clearly, Pb4 can not be implemented if the number of failures is not bounded. In particular, if all sites fail, then no service can be provided and so the service is not (k, Δ) for any finite k and Δ.

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

[2] Therefore, as well as being finite, the number of such periods of service outages can occur is also bounded (by k).

3 The Model

Consider a system with n_s sites and n_c clients. Site clocks are assumed to be perfectly synchronized with real time[3]. Clients and sites communicate through a completely connected, point-to-point, FIFO network. Furthermore, if processes (clients or sites) p_i and p_j are connected by a (nonfaulty) link, then we assume for some *a priori* known δ, a message sent by p_i to p_j at time t arrives at p_j at some time $t' \in (t..t+\delta]$.

We assume that all clients are non-faulty and consider the following types of site and link failures: *crash failures* (faulty sites may halt prematurely; until they halt, they behave correctly) [4], *crash+link failures* (faulty sites may crash or faulty links may lose messages), *receive–omission failures* (faulty sites may crash or omit to receive some messages), *send–omission failures* (faulty sites may crash or omit to send some messages), *general–omission failures* (faulty sites may fail by send-omission, receive-omission, or both). Note that link failures and the various types of omission failures are different only insofar as a message loss is attributed to a different component. Link failures are masked by adding redundant communication paths; omission failures are masked by adding redundant sites. As we will see, the lower bounds for the two cases are different.

Let f be the maximum number of components that can be faulty (*i.e.* f is the maximum number of faulty sites in the case of crash, send–omission, receive–omission and general–omission failures, whereas f is the maximum number of faulty sites and links in the case of crash+link failures).

4 Lower Bounds

In Tab. 1, we repeat the lower bounds from [5] for the degree of replication, the blocking time and the failover time for the various kinds of failures. Informally, a protocol is *C–blocking* if in all failure–free runs, the time that elapses from the moment a site receives a request until a site sends the associated response is bounded by C.[5] Failover time is defined to be the longest duration (over all possible runs) for which there is no primary. However, the failover time bounds only hold for protocols that satisfy the following additional (and reasonable) property.

Pb5: A correct site that is the primary remains so until there is a failure.

---

[3] The protocols can be extended to the more realistic model in which clocks are only approximately synchronized [7].

[4] The lower bounds are also tight for fail-stop failures [10] except for the bound on failover time.

[5] We assume that it takes no time for a site to compute the response to a request.

Table 1. Lower Bounds—Degree of Replication, Blocking Time and Failover Time

Failure type	Replication	Blocking time (C)	Failover Time
Crash	$n_s > f$	0	$f\delta$
Crash+Link	$n_s > f+1$	0	$2f\delta$
Send–Omission	$n_s > f$	δ if $f=1$ 2δ if $f>1$	$2f\delta$
Receive–Omission	$n_s > \lfloor \frac{3f}{2} \rfloor$	δ if $n_s \leq 2f$ and $f=1$ 2δ if $n_s \leq 2f$ and $f>1$ 0 if $n_s > 2f$	$2f\delta$
General–Omission	$n_s > 2f$	δ if $f=1$ 2δ if $f>1$	$2f\delta$

5 Summary of Results

We first present a primary-backup protocol schema that will be used to derive the protocols for all the failure models. This schema is based on the *properties* of two key primitives, **broadcast** and **deliver**, that sites use to exchange messages. We show that the schema satisfies Pb1—Pb5 by only using these properties independent of the particular failure model. Each failure model—crash, crash+link, send–omission, receive–omission and general–omission—is handled with a different implementation of **broadcast** and **deliver**, and in all but one case optimal protocols are constructed.

The protocols for crash and crash+link failures show that all the corresponding lower bounds are tight. The protocol for general-omission failures uses a translation technique similar to [8], and demonstrates that our lower bounds for general-omission failures are tight, except for the bound on blocking time when $f = 1$. However, for this special case we have derived a different protocol (not described in this paper) having optimal blocking time . In all failure free runs of this protocol, the site that receives the request (*i.e.* the primary) is not the site that sends the response to the client. We show that this behavior is necessary in this paper.

We do not show the protocols for send-omission and receive-omission failures in this paper because they are similar to the protocol for general-omission failures. These protocols establish that the bounds for send-omission failures are tight. For receive-omission failures, the lower bound on blocking time when $n_s > 2f$ and the lower bound on failover time are also tight. However, our protocol does not have optimal replication, as it requires $n_s > 2f$ (rather than $n_s > \lfloor \frac{3f}{2} \rfloor$).

Finally, in [5] we proved that all receive-omission protocols having $\lfloor \frac{3f}{2} \rfloor < n_s \leq 2f$ necessarily exhibit a scenario in which a non-faulty primary is forced to relinquish control to a faulty backup. In Sect. 9, we describe such a protocol: it uses two sites and tolerates a single receive-omission failure. In addition, this protocol is δ-blocking and so it demonstrates that our lower bound on blocking time is tight for $n_s \leq 2f$ and $f = 1$. As in the protocol for general omission when $f = 1$, it is the backup that sends responses to clients. This behavior is shown to be necessary for an important class of protocols.

6 Protocols for the Clients and the (k, Δ)–bofo server

Property Pb4 requires that the primary–backup service behave like some (k, Δ)–bofo server. Figure 1 gives such a canonical (k, Δ)–bofo server (say s), and Fig. 2 gives the protocol for client i interacting with s. As with any other bofo server, a client will not receive the response to a request if either the request to s or the response from s is lost.

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

procedure initialize()
  state := ε

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

Fig. 1. Protocol run by a single (k, Δ) bofo–server s

In Fig. 2, *response–time* corresponds to the amount of time the client has to wait in order to get the response from s, which is just the round trip message delay. The exact value for *response–time* depends on the failure model being assumed.

7 The Primary–Backup Protocol Schema

We first make the simplying assumption that the links between the clients and the sites are non–faulty and there are no omission failures between the clients and the sites (i.e. only the links between sites can be faulty for crash+link failures,

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

Fig. 2. Protocol run by client i interacting with server s

and omission failures can occur only between sites for omission failures). We show in Sect. 7.1 how this assumption can be removed.

In order to emulate the server s (and consequently satisfy property Pb4), our primary–backup protocol consists of n_s sites $\{s_1, \ldots, s_{n_s}\}$, each of which runs the protocol in Fig. 3. The protocol for the clients remains the same.

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

Fig. 3. Protocol run by site s_i to emulate server s

The procedures **primary** and **backup** (shown in Fig. 4) are the same for all the failure models. On the other hand, the implementation of the procedures **initialize**, **broadcast**(used in Fig. 4), **delivery–process** and **failure–detector** change depending on the particular failure model. However, we ensure that these different implementations always satisfy a set of properties, called B1—B11 below. We extracted these properties in order to make our proofs modular. In particular we proved that, independent of the failure model, the protocol in Figs. 3 and 4 satisfies Pb1–Pb5, as long as the remaining procedures satisfy B1—B11. As a result, we could then prove Pb1–Pb5 for any other failure model by just ensuring that the implementation of **broadcast**, **delivery–process** and **failure–detector** for that failure model satisfied B1–B11.

```
procedure primary(j)
  cobegin
    || inform–clients("Dest = s_j")
    || broadcast((mylastlog, s_j, last(state_j)), j) /* to all sites */
      do forever
        when received request from client c
          response := Π(state_j, request)
          state_j := state_j ∘ response
          broadcast((log, s_j, response), j)
          send response to client c
      od
  coend

procedure backup(k)
  do forever
    ((tag, s_j, r), j) := Deq(Rqueue_k)
    /* assume that dequeueing an empty queue
    does not return any sensible value of tag */

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

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

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

Fig. 4. The procedures **primary** and **backup**

We now give the properties B1—B11. In these properties, d, C and τ are some constants whose values depend on the failure model. Intuitively, d corresponds to the amount of time that can elapse from the time a message is broadcast to the time it is dequeued by the receiver, C corresponds to the blocking time and τ corresponds to the interval between successive "I am alive" messages that sites send to each other (as we will see in the implementation of **failure–detector**).

When we say that a site "halts", we mean that either the site has crashed or has stopped executing the protocol by executing a **stop**. The array of booleans $Faulty_k$ indicates which servers s_k believes has halted: $Faulty_k[s_j]$ being true implies s_k believes that s_j has halted. Finally, we define a broadcast by a site to be *successful* if the site does not halt during the execution of **broadcast**.

The properties can be subdivided according to the procedures to which they relate:
Properties of **broadcast** and **delivery–process**:

B1: If s_j initiates a broadcast b' after broadcast b, then no site dequeues b' before b.
B2: If s_j initiates a broadcast b at time t, then no site dequeues b after time $t + d$.
B3: If s_j initiates a broadcast at time t and does not halt by time $t + C$, then the broadcast is successful. Furthermore, no broadcast takes longer than C to complete.

Properties of **failure–detector**:

B4: If $Faulty_j[s_k]$ becomes true, then it continues to be true, unless s_j halts.
B5: The value of $Faulty_j[s_k]$ can only change at time $t = l\tau + d$ for some integer $l \geq 0$.
B6: If $Faulty_j[s_k] = true$ at time t then s_k has halted by time t.
B7: If s_j has not halted by time t_1, and s_i, $i < j$ has halted by time t_2 where $t_1 = t_2 + \tau + d$, then $Faulty_j[s_i] = true$ by time t_1.

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

B8: No correct site halts in procedures **initialize, broadcast, delivery–process** or **failure–detector**.
B9: If s_j initiates a successful broadcast at time t, then for all non–halted sites s_k, $k > j$, $Faulty_k[s_j] = false$ through time $\lceil \frac{t}{\tau} \rceil \tau + d$.
B10: If s_j initiates a successful broadcast b, then for every non–halted site s_k: $(Faulty_k[s_j] = true) \Rightarrow$ (s_k has dequeued b).
B11: If s_j initiates a broadcast b at time t and $s_k, k > j$ broadcasts b', then either no site dequeues b after b', or $Faulty_k[s_j] = false$ through time $t + d$.

7.1 Outline of the Proof of Correctness

We now informally argue that the protocol in Figs. 3 and 4 satisfies Pb1–Pb5 as long as the procedures **initialize**, **broadcast**, **delivery–process** and **failure–detector** satisfy B1—B11.

Define: $Prmy_{s_j}$ at time $t \equiv s_j$ has not halted by time t

$$\wedge \forall k < j : Faulty_j[s_k] = true \text{ at time } t.$$

From the above definition, Pb1 can now be seen from B6 and the backup protocol in Fig. 4. Pb2 trivially follows from Fig. 2. Pb3 follows from Fig. 4 as no request is sent to a site s_j before s_j becomes the primary. Also, Pb5 holds (from B8 and Fig. 4) as a correct primary continues to be the primary. We now show Pb4.

In order to show Pb4, we need to show two things–the state of the new primary is consistent with the state of old primary; and all outages are bounded. We first show that the states are consistent.

Starting at the top of Fig. 4: when a site s_j becomes the primary, it first informs the clients of its identity by calling **inform–clients**. For now, ignore the broadcast of (`mylastlog`,s_j,–) by primary s_j.

Whenever s_j gets a request from a client, it computes the response, changes state, broadcasts the log to the backups and sends the response back to the client. It can be seen from Fig. 4 that if primary s_j sends a response r to the client, then s_j must have executed a successful broadcast of (`log`, s_j, r). This fact and properties B1,B2,B9 and B10 imply that (`log`, s_j, r) must also have been dequeued by any backup s_k before s_k becomes the primary. Thus, the state of s_k will continue to be consistent with the state of s_j iff the states were consistent when s_j became the primary. We show this as follows.

Informally, the states of s_j and s_k could be inconsistent when s_j becomes the primary for the following reason. Consider a scenario in which some primary s_i crashes during the broadcast of (`log`,s_i, r) for some r. It is possible that s_k received (`log`, s_i, r) and s_j did not. As a result, the states of s_j and s_k now differ. It is for this reason that s_j broadcasts (`mylastlog`, s_j, r') where $r' = last(state_j)$ on becoming the primary. On receiving this, s_k sees that $r' \neq last(state_k) = r$ and removes r from its state. As a result, $state_j$ and $state_k$ become equal. Similarly, s_k would add r to its state had s_j, and not s_k, received (`log`, s_i, r).

In the scenario described in the last paragraph, response r is never sent to the client (*i.e.* there is a service outage). We now show that such outages are bounded. s_i did not send the response, and so by B3, must have halted by time t (say). Now from B7 either s_{i+1} halts or becomes the primary by time $t + \tau + \delta$. Since no correct site halts (by B8 and Fig 4), and the number of faulty sites are bounded by f, there eventually will be a time when there is a correct primary and no more outages occur.

From B3, the protocol C–blocking. Furthermore, it can be shown from B7, B8 and Fig. 4 that the failover time of the protocol is $f(d + \tau)$ for arbitrarily small and positive τ.

However, the **primary** procedure in Fig. 4 does not work if there are message losses between the clients and the sites (due to link or omission failures). For example, a non–faulty primary might omit to receive all requests from a client due to a failure, violating Pb4. Similarly, **inform–client**s might omit to inform some of the clients. However, it is relatively easy to account for these failures when clients are non-faulty. Assume that there is an upper bound (say G) between any two requests from a client and that requests carry sequence numbers. If the primary does not receive any requests from a client during an interval of length G or if the primary receives some request with a sequence number gap, then the primary halts. Similarly, the primary can detect that a response was lost by having clients acknowledge responses. If such an acknowledgement is not received, then again the primary halts. Properties Pb1–Pb5 can again be shown to be true if we make the above modification in Figs. 2 and 4.

8 Implementation for the various Failure Models

In this section, we show how to implement B1—B11 for the various failure models.

8.1 Crash Failures

The procedures implementing B1—B11 for crash failures are given in Fig. 5. Whenever we say that a site "delivered M", we mean that the procedure **deliver** has been called with M. **Enq** adds an element to the head of a queue and **Deq** dequeues an element from the tail.

---

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

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

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

procedure delivery–process(k)
  do forever
    if received M **then deliver**(M, k)
  od

procedure failure–detector(k)
  cobegin
    || **for** $i := 0$ to ∞
        when *current–time* $= i\tau$: **send** $(\texttt{alive}, s_k, i\tau)$ **to** all sites
    || **for** $i := 0$ to ∞
        when *current–time* $= i\tau + d$:
          $\forall j$: **if** not delivered $(\texttt{alive}, s_j, i\tau)$ **then** $Faulty_k[s_j] := true$
  coend

Fig. 5. Procedures for crash failures

We now informally argue that B1–B11 hold for this implementation if $d = \delta$ and $C = 0$. B1 holds as channels are FIFO and, B2 holds as $d = \delta$ and the maximum message delivery time is also δ. B3, B4 and B5 can be seen trivially. B6 and B7 can be seen from **failure–detector** as there are no message losses and message delivery time is atmost δ. B8 holds trivially. It can be shown that if s_j halts at time t, then no site sets $Faulty[s_j]$ to true before time $t + \delta$. B9, B10 and B11 now follow.

The procedures in Fig. 5 require $n_s > f$, and so the lower bound on the degree of replication is tight. Since $C = 0$ and $d = \delta$, from Sect. 7.1, the lower bounds on blocking time and failover time are tight as well.

8.2 Crash+Link Failures

The procedures in Sect. 8.1 do not work if links can fail. For example, if s_j sends a message to s_k then the message might not reach s_k due to a link failure (which will violate B6 and B10). We therefore replace the implementation in Fig. 5. with the one in Fig. 6, except that **deliver** is the same as before. For this implementation, $d = 2\delta$ and $C = 0$. These procedures use **fifo–broadcast** and **fifo–deliver** in Fig. 7 which ensure that intermittent link failures become permanent failures: if s_j fifo–broadcasts a message m to s_k and s_k omits to fifo–deliver m, then s_k will not fifo–deliver any subsequent message from s_j.

It can be shown (proof omitted) that this new implementation again satisfies B1–B11 if $n_s > f + 1$. Informally, this is true because of the following reason. Whenever s_j initiates a broadcast of M at time t, it sends M to all sites, and the sites then relay M to all other sites. Since $n_s > f+1$, there is always at least one non-faulty path between any two non-crashed sites, where a path consists of zero or one intermediate sites. Therefore, if s_j does not crash during the broadcast, then all non-crashed sites will deliver M by time $t + 2\delta$. Furthermore B1 will be satisfied because of the FIFO properties of fifo–broadcast and fifo–deliver.

This crash+link protocol requires $n_s > f + 1$, is 0–blocking (since $C = 0$), and has a failover time of $f(2\delta + \tau)$ (since $d = 2\delta$). Thus, all lower bounds for crash+link failures are tight.

8.3 General–Omission Failures

The implementation of the procedures for general-omission failures is given in Figs. 8 and 9, except **delivery–process** which is the same as Fig. 6. Whenever, we say that a site "fifo–delivered M", we mean that the procedure **fifo–deliver** was called with M. These procedures were developed using a technique similar to [8] (although modified to work in our non-round-based model) which requires $n_s > 2f$ and $d = 2\delta$.

```
procedure initialize(k)
  state_k := Rqueue_k := Dqueue_k := ε
  ∀i : Faulty_k[s_i] := false
  last-sent_k := ∀j :expected_k[j] := 0

procedure broadcast(M, k)
  time :=current-time
  fifo-broadcast(init, M, s_k, time)

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

procedure failure-detector(k)
  A_j^i = (alive, s_j, iτ)
  cobegin
    || for i := 0 to ∞
         when current-time = iτ: fifo-broadcast(init, A_k^i, s_k, iτ)
    || for i := 0 to ∞
         when current-time = iτ + d:
           ∀j : if not delivered A_j^i then Faulty_k[s_j] := true
  coend
```

Fig. 6. Procedures for crash+link failures

```
procedure fifo-broadcast(tag, M, s_k, t)
  send (tag, M, s_k, t,last-sent_k) to all
  last-sent_k :=last-sent_k + 1

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

procedure fifo-delivery-process (k)
  do forever
    if received (tag, M, s_j, t, last_j) then
      if (last_j ≠expected_k[j]) then skip
      else
        expected_k[j] := expected_k[j] + 1
        fifo-deliver (tag, M, s_j, t)
  od
```

Fig. 7. Procedures for crash+link failures

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

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

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

Fig. 8. Procedures for general-omission failures

We now briefly argue that these procedures satisfy B1—B11. The detailed proof is omitted from this paper. Had we used the implementation of broadcast in Fig. 6, B10 (in particular) would be violated because a faulty primary s_j might omit to send the logs to the backups. Therefore, in Fig. 8, s_j stops in the broadcast of a response (say r) if less than $n_s - f$ sites fifo-deliver and subsequently fifo-broadcast r. However, even if s_j does not stop in the broadcast, a faulty (but non-crashed) site s_k might still omit to deliver r, due to a receive-omission failure, and later become the primary were s_j to fail. To prevent this, s_k ensures (in procedure **failure-detector**) that it fifo-delivers some message (say m') from at least one of the above $n_s - f$ sites that had earlier fifo-broadcast r. If s_k does not receive such an m', then s_k stops. Now, if s_k omitted to fifo-deliver r, then by the properties of fifo-broadcast and fifo-deliver, s_k cannot fifo-deliver m' and would stop (and, therefore, cannot become the primary). Property B6 is similarly satisfied by ensuring that sites detect their own failure to send or receive `alive` messages and therefore stop.

These procedures require $n_s > 2f$, $d = 2\delta$ and $C = 2\delta$. Furthermore, we have developed a protocol for $f = 1$ (omitted in this paper) that is δ–blocking. Thus, we establish that all lower bounds for general-omission failures are tight.

As mentioned earlier, the δ–blocking protocol for $f = 1$ has scenarios in which the site that receives the request is not the site that responds to the clients. This is in fact necessary. Define a protocol to be "pass the buck" if in any failure-free run of the protocol, the site that receives a request is not the site that sends the corresponding response.

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

Fig. 9. Procedures for general-omission failures

Theorem 1. *Any C-blocking protocol, where $C < 2\delta$, for send-omission failures is "pass the buck".*

Proof. Omitted in this paper. See [4]. □

8.4 Other Failure Models

The implementations of the procedures for send-omission and receive-omission failures are similar to those for general-omission failures and so are omitted from this paper. For receive-omission failures, the lower bound on the degree of replication and the lower bound on blocking time when $n_s \leq 2f$ and $f > 1$ are *not* tight. Finding optimal protocols remains an open problem. However, the lower bound on failover time for receive-omission failures, and all lower bounds for send-omission failures are tight.

9 A Surprising Protocol

We now describe a δ–blocking protocol tolerating receive-omission failures for the special case of $n_s = 2$ and $f = 1$. This protocol is complex, and so we omit the detailed description and only outline the protocol's operation here. This protocol shows that our lower bound on blocking time when $n_s \leq 2f$ and $f = 1$ is tight. The protocol has the odd (yet necessary as shown in [5]) property that a non-faulty primary is forced to relinquish to a faulty backup. Furthermore, the protocol is "pass the buck". We, however, show that most δ–blocking protocols tolerating receive omission failures have to be "pass the buck".

Informally, let Γ be the maximum time between any two successive client requests (possibly from different clients), and let D be such that if some site s becomes the primary at time t_0 and remains the primary through time $t \geq t_0+D$ when a client i sends a request, then $Dest_i = s$ at time t. We write $D < \Gamma$ to mean that D is bounded and Γ is either unbounded or bounded and greater than D. Then

Theorem 2. *Any C-blocking protocol, where $C < 2\delta$, for receive-omission failures with $n_s \leq 2f$ and $D < \Gamma$ is "pass the buck".*

Proof. Omitted from this paper. □

Whether a protocol has to be "pass the buck" when the relation $D < \Gamma$ does not hold is an open question.

We now describe the protocol. There are two sites s_0 and s_1. They communicate with each other using **fifo–broadcast** and **fifo–deliver** shown in Fig. 7. Henceforth, when we say that a site sends a message to the other, we will mean that the message is sent with **fifo-broadcast** and other site receives it with **fifo-deliver**.

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

Let s_0 initially be the primary. Whenever s_0 receives a request from the client, it computes a response r, changes state, and sends (`green`,r) to s_1. Upon receiving this message, s_1 updates its state, acknowledges to s_0, and then sends r to the client. Because it is the backup that responds to the client, the protocol is δ–blocking. Site s_0 processes a new request only after receiving the acknowledgement from s_1 for the previous request. Finally, s_0 periodically sends `alive` messages to s_1, and s_1 acknowledges these messages.

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

Case 1: s_1 has crashed. As a result, s_0 does not receive the acknowledgement to the `alive` message. s_0 continues as the primary. From then on, whenever s_0 receives a request from the client, it computes the response r, sends (`red`,r) to s_1, and then sends the response back to the client. Also, s_0 continues to send `alive` messages. Since s_0 is correct, it can continue like this forever.

Case 2: s_1 is faulty and omitted to receive (`green`,r). By the property of fifo–broadcast and fifo–deliver, s_1 will not receive the `alive` messages that s_0 sends.

s_1 concludes that s_0 has crashed, sends ("s_1 is primary") to s_0 and becomes the primary. After that, it behaves like s_0 in case 1 above (including sending `alive` messages to s_0). Since s_0 is correct, it receives ("s_1 is primary") (as opposed to case 1) and so it becomes the backup. Also, since s_0 is correct it will not omit to receive (**red**,r) messages that s_1 sends and so s_0 keeps its state consistent with s_1. Subsequently, if s_0 stops receiving `alive` messages from s_1, then s_1 has crashed and s_0 becomes the primary once again.

Case 3: s_0 is faulty. Since s_1 is correct, it receives the `alive` message from s_0, sends the corresponding acknowledgement and remains the backup (as opposed to case 2). However, by the property of fifo–broadcast and fifo–receive, s_0 will not receive this acknowledge to the `alive` message (or the ("s_1 is primary") message), and so it behaves as in case 1 and continues as the primary. Similar to case 2, s_1 receives all (**red**,r) messages that s_0 sends and so its state is consistent with s_0. Finally, s_1 becomes the primary if it stops receiving `alive` messages from s_0.

Case 2 in the protocol is the odd scenario in which the correct primary s_0 is being forced to relinquish to s_1, known to be faulty. However, this scenario is not something peculiar to our protocol. We showed in [5] that relinquishing to a faulty backup is necessary when $n_s \leq 2f$.

10 Discussion

In [5], we present lower bounds for primary–backup protocols which constrain the degree of replication, the failover time, and the amount of time it can take to respond to a client request. In this paper, we derive matching protocols and show that all except two of these lower bounds are tight. Furthermore, we show that in some cases the optimal response time can only be obtained if the site that receives the request is not site that sends the response to the clients.

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 as these protocols were developed for an asynchronous system [6,9]. We are currently studying possible characterizations for a primary-backup protocol in an asynchronous system and expect to extend our results to this setting.

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. 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.

3. Anupam Bhide, E.N. Elnozahy, and Stephen P. Morgan. A highly available network file server. In *USENIX*, pages 199–205, 1991.
4. Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary–backup protocols. Technical report, Cornell University, Ithaca, N.Y., 1992. In preparation.
5. 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*, 1992. To Appear.
6. Timothy Mann, Andy Hisgen, and Garret Swart. An algorithm for data replication. Technical Report 46, Digital Systems Research Center, 1989.
7. Gil Neiger and Sam Toueg. Substituting for real time and common knowledge in asynchronous distributed systems. In *Sixth ACM Symposium on Principles of Distributed Computing*, pages 281–293, Vancouver, Canada, August 1987. ACM SIGOPS-SIGACT.
8. 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.
9. 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.
10. 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.