Highly-Available Services Using the Primary-Backup Approach

Navin Budhiraja 
Keith Marzullo
Department of Computer Science
Cornell University
Ithaca, NY 14850

Suppose one wishes to build a service that is to be highly available: that is, the service must continue to be available even if a subset of processors or communication links fail. There are two basic approaches one can use to build such a service.

The first approach is commonly called primary-backup or primary-copy [1]. In this approach, one of the servers is designated as the primary and all the others are designated 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. The virtues of this approach are often stated to be its simplicity and it not being profligate in use of computing resources.

The other approach is commonly called state machine or active replication [14]. All (correct) servers receive the same requests in the same order, and all (correct) servers send their responses to the requesting client. The client, on receiving these responses, uses a voter to compute the correct response. This approach is based on atomic multicast protocols which have been efficiently implemented in practical systems [3]. The virtues of the state machine approach are often stated to be its symmetry and its ability to mask failures completely.

There has been a tremendous amount of work done on the formal aspects of active replication, which has resulted in a good understanding of its tradeoffs and limitations. The same, however, can not be said for the primary-backup approach. Outside of small investigations [2, 8, 11], there has been no formal study of the limits and tradeoffs of a primary-backup system. Over the last year we have been working on filling this gap. In particular, we have answered the following questions for synchronous primary-backup systems:

- What is the minimum degree of replication (i.e. the total number of servers) required to tolerate various kinds for processor and link failures?
- How much time can it take for a service to respond to a request (even after assuming negligible request processing time) in the absence of failures?
- What is the the longest duration the service can be unavailable when an a priori bounded number of failures can occur?

This position paper summarizes our work. We have derived lower bounds [6] and the corresponding optimal protocols [5] for the above three parameters for synchronous primary-backup systems. We are now comparing our results with similar results for active replication in order to determine whether the common folklore on the virtues of the two approaches can be shown formally. We have also extended some of our results to asynchronous primary-backup systems [4].

Finally, we have implemented an important subclass of primary-backup protocols that we call 0-blocking. These protocols are interesting because they introduce no additional protocol related delay into a failure-free service request. Through implementing these protocols we hope to determine the appropriateness of our theoretical system model and uncover other practical advantages or limitations of the primary-backup approach.

Lower bounds for synchronous primary-backup systems

In order to derive lower bounds, one has to specify precisely what constitutes a primary-backup system. We used the following four properties which most existing implementations satisfy. The first three properties specify how a client interacts with the service.

Pb1: There exist predicates $Prm_y$, on the state of each server $s$. At any time, there is at most one server $s$ whose state satisfies $Prm_y$. 


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

Pb3: If a request arrives at a server that is not the current primary, then the request is discarded.

In a primary-backup system, a request can be lost if it is sent to a faulty primary. However, periods during which requests are lost 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 \textit{bofo} (bounded outage finitely often). We say that an outage occurs at time $t$ if some client makes a request at time $t$ but does not receive a response (for simplicity, we assume that every request elicits a response). In a $(k, \Delta)$-bofo server, all outages occur in at most $k$ intervals of time, with each interval having a duration of at most $\Delta$.

The final property requires that the primary-backup service behave as if it were a single bofo server (for some values of $k$ and $\Delta$).

Pb4: If the number of failures is \textit{a priori} bounded, then there exist fixed values for $k$ and $\Delta$ such that the service implements a single $(k, \Delta)$-bofo server.

The lower bound results we describe depend on the type of server or link failures to be tolerated. We consider the following types of failures: crash failures in which a server may halt prematurely, crash+link failures in which a server may crash or a link may lose messages, receive-omission failures in which a server may crash or omit to receive some messages (perhaps due to buffer overflows at the receiver), send-omission failures in which a server may crash or omit to send some messages (perhaps because the sender is overloaded and omits to send a message on time), general-omission failures in which a server may fail by send-omission, receive-omission, or both. We further assume that at most $f$ components can be concurrently faulty (for crash+link failures, both links and servers can be faulty).

Table 1 shows the minimum number of servers ($n$) needed to tolerate $f$ failures of various types. Thus, it can be seen that for many kinds of failures, more than $f+1$ servers are required to tolerate $f$ failures. We have also derived the corresponding optimal protocols which prove that all these lower bounds, except the one for receive-omission failures, are tight. For receive-omission failures we only have a protocol when $n > 2f$. Finding an optimal protocol remains an open problem.

<table>
<thead>
<tr>
<th>Failure type</th>
<th>Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>$n &gt; f$</td>
</tr>
<tr>
<td>Crash+Link</td>
<td>$n &gt; f+1$</td>
</tr>
<tr>
<td>Send-Omission</td>
<td>$n &gt; f$</td>
</tr>
<tr>
<td>Receive-Omission</td>
<td>$n &gt; \frac{3f}{2}$</td>
</tr>
<tr>
<td>General-Omission</td>
<td>$n &gt; 2f$</td>
</tr>
</tbody>
</table>

Table 1: Lower bounds for the degree of replication

Table 2 presents the results on blocking times. Informally, a protocol is $C$-blocking if in all failure-free runs, the interval between the times when the service receives a request and when the service sends the associated response is no longer than $C$ (assuming that it takes negligible time for the service to compute the response to a request). Table 2 presents the lower bounds for $C$.

<table>
<thead>
<tr>
<th>Failure type</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>0</td>
</tr>
<tr>
<td>Crash+Link</td>
<td>0</td>
</tr>
<tr>
<td>Send-Omission</td>
<td>$\delta (f = 1)$ $+$ $2\delta (f &gt; 1)$</td>
</tr>
<tr>
<td>Receive-Omission</td>
<td>$\delta (n \leq 2f, f = 1)$ $+$ $2\delta (n \leq 2f, f &gt; 1)$ $+$ $\delta (n &gt; 2f)$</td>
</tr>
<tr>
<td>General-Omission</td>
<td>$\delta (f = 1)$ $+$ $2\delta (f &gt; 1)$</td>
</tr>
</tbody>
</table>

Table 2: Lower bounds for $C$

Again we have shown that all our lower bounds, except the one for receive-omission failures when $n \leq 2f$ and $f > 1$, are tight. One surprising observation was that in all $\delta$-blocking protocols, it is the backup and not the primary which responds to the clients. Unfortunately, this tradeoff is unavoidable.

Finally, Table 3 gives the lower bounds for the \textit{failover time}—the longest duration (over all possible runs) for which there is no primary. However, these lower bounds only hold for systems that satisfy the following additional (and reasonable) property.

Pb5: A correct server that is the primary remains so as long as there are no failures.

Pb5 is necessary for the lower bounds shown in Table 3 as there are protocols that do not satisfy Pb5 and violate these bounds.
Table 3: Lower bounds for failover times.

<table>
<thead>
<tr>
<th>Failure type</th>
<th>Failover time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>$f_b$</td>
</tr>
<tr>
<td>Crash+Link</td>
<td>$2f_b$</td>
</tr>
<tr>
<td>Send-Omission</td>
<td>$2f_b$</td>
</tr>
<tr>
<td>Receive-Omission</td>
<td>$2f_b$</td>
</tr>
<tr>
<td>General-Omission</td>
<td>$2f_b$</td>
</tr>
</tbody>
</table>

Because of the above result, we need to sufficiently strengthen the asynchronous model in order to build fault-tolerant services. One possibility is using failure detectors [7]. We have shown that given such a failure-detector, one can build almost any kind of service in an asynchronous system with crash failures. However, since message delivery time is still unbounded, the only metric that seems useful in order to specify optimality of asynchronous protocols is the degree of replication. For this metric, we have shown that most services can only tolerate a minority of the servers to be faulty.

Asynchronous primary–backup systems

An asynchronous system is one in which there is no upper bound on processor speeds and message transmission delays. However, our specification (for example, Pb4) does not extend to asynchronous systems as we cannot put a bound on the amount of time the service can be unavailable. We have, however, shown in [4] that it is impossible to build many kinds of replicated services in an asynchronous system, independent of the underlying structure (whether primary–backup or active replication), and we need to sufficiently augment the asynchronous model in order to build such services. The reason for this is the following.

One would expect a fault–tolerant service to at least satisfy the following two properties: safety—any response that the service sends is "consistent" with the responses sent earlier; and liveness—if a correct client initiates a request, then the client receives the response within a finite (but possibly unbounded) amount of time. Note that in a primary–backup system, since a request can be lost if sent to a faulty primary, liveness can potentially be achieved by some kind of stub at the client that resends a copy of the request to the new primary. Services that ensure both safety and liveness can be built in a synchronous system with bounded number of failures. Also, services that ensure safety can be built in an asynchronous system [12, 13]. However, we have shown that both safety and liveness cannot be ensured for many kinds of non–trivial services in an asynchronous system. We demonstrate this by showing that a safe and live implementation of a particular service (a service implementing a fifo queue) can be used to solve consensus. The result then follows from the impossibility of consensus in an asynchronous system [9].

Implementing primary–backup protocols

In order to derive the lower bounds for primary–backup protocols, we had to make some simplifying assumptions about the system model. We have therefore implemented some of these protocols in order to see how realistic these assumptions are. In particular, we have implemented the 0–blocking protocols for crash failures, crash+link failures and receive–omission failures (which only exits when $n > 2f$).

Through this implementation we are measuring the response time of the service for the various failure models as the following parameters vary—the degree of replication, the request rate, the number of clients, and the relative mix of update and non-update requests. Our preliminary results show that the average response time increases as the number of servers increase, even in the absence of failures. And as expected, the increase is larger if the request rate or the number of clients is larger (we haven't yet changed the relative mix of update and non–update requests). However, for crash failures this increase is very small and is almost linear in the number of servers. This is because for crash failures the number of messages sent per request increases linearly with the number of servers. On the other hand, for crash+link and receive–omission failures the response time increases almost quadratically, as the number of messages sent per request is now quadratic in the number of servers.

Since the number of messages sent was so dominant in determining the response time, we implemented a modified protocol that requires $n > f$ servers, sends only a linear number of messages, and tolerates link or receive–omission failures. However, this protocol only works under the weakened assumption that (link or receive–omission) failures are transient and so our
protocols does not violate the lower bounds in Table 1. Unfortunately, this protocol is 26-blocking. However, we were surprised to see that in most cases this 26-blocking protocol sending linear number of messages has a smaller response time than the 0-blocking protocols sending quadratic number of messages. This shows that the number of messages sent, and not the blocking time, can be a more dominant factor in determining the response time.

Another tradeoff that we hope to understand from our experiments is the following. If the probability of concurrent failures is large, then we have to keep the degree of replication high (as dictated by Table 1). However, this hurts failure-free performance (as our experiments show). On the other hand, if concurrent failures are unlikely, then we can keep the number of servers small, and integrate new servers whenever there are failures. This improves failure-free performance, possibly at the expense of greater delay during failures. We expect that with a moderate cost of reintegration, a small degree of replication will be best.

Acknowledgements

Our research is in collaboration with Prasad Jayanti, Fred B. Schneider and Sam Toueg.

References