NAG2-593

Tradeoffs in Implementing Primary-Backup Protocols*

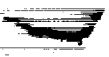
IN-81 CR 27081 P-18

Navin Budhiraja**
Keith Marzullo

TR 92-1307
October 1992

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

*This work is supported by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 and by grants from IBM and Siemens.
**Also supported by an IBM Graduate Fellowship.



Tradeoffs in Implementing Primary–Backup Protocols *

Navin Budhiraja † Keith Marzullo‡

Department of Computer Science, Cornell University
Ithaca NY 14853, USA
{navin,marzullo}@cs.cornell.edu

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 the servers fail, the service remains available. A common approach to structuring such 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].

In most such primary–backup protocols, when the primary receives a client request, it informs the backups about the request, and then responds to the client. Informally, this primary–backup protocol is *non–blocking* if the primary does not wait for an acknowledgement from the backups before it sends the response; otherwise, it is *blocking*. Most of the existing protocols are blocking as non–blocking protocols cannot be constructed for some kinds of failures.

However, we show that non–blocking protocols can be constructed for most of the process and communication failures that are expected to occur in the primary–backup systems of the future. Since non–blocking protocols can theoretically achieve the smallest possible response time, this papers analyzes these protocols under various system parameters. We analyze two kinds of non–blocking protocols—one in which processes use point–to–point communication to exchange messages, and the other in which processes use hardware broadcasts.

As expected, most of our simulations showed that non–blocking protocols have a smaller response time than blocking protocols. However, this was not the case in general. In particular, when processes used point–to–point communication, then under certain circumstances (which depended

*This work is supported by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593 and by grants from IBM Glendale Programming Laboratory, IBM T. J. Watson Research Center, Xerox Webster Research Center and Siemens.

†Also supported by an IBM Graduate Fellowship.

‡Alternate address: Department of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Dr. 0114, La Jolla, CA 92093–0114

both on the number of servers and clients and on the frequency of client requests), the non–blocking protocols lead to overly-high contention on the network, thereby causing their response time to be longer than the response time of blocking protocols. This undesirable behavior, however, did not occur when processes used broadcast communication.

Another surprising result was the following: in some cases using broadcast communication, the average response time of the non–blocking protocols was nearly independent of the number of servers. This implied that if failures are rare, then a smaller response time can be achieved by keeping the degree of replication (i.e., the number of servers) high! This is because larger the number of servers, less often the reintegration of new servers has to be done, which saves on the reintegration cost.

The above two observations might lead one to believe that broadcast communication should be preferred over point-to-point communication. However, as was pointed out in [2], broadcasting bears a price in interrupts to non–interested processes. This behavior was observed in one of our protocols where with a small degree of replication, the point–to–point protocol performed better than the corresponding broadcast-based protocol. One should, therefore, be careful in choosing the right kind of communication when designing such a system.[1]

The rest of the paper is organized as follows. Section 2 discusses non–blocking protocols in more detail. Section 3 describes our implementation, and Section 4 discusses the tradeoffs in failure–free runs between non–blocking and blocking protocols. Section 5 extends our results to runs with failures, and we conclude in Section 6.

2 Non–blocking protocols

In this section, we discuss the class of protocols that is the main focus of the paper—*non–blocking* primary–backup protocols. We also argue that these protocols can be constructed for future primary–backup systems.

In most primary–backup protocols, when the primary receives a client request, it processes the request, updates its state, informs the backups about the state update (so that the states of the backups remain consistent with the state of the primary), and finally responds to the client. Informally, such a protocol is non–blocking if the primary does not have to wait for an acknowledgement from the backups before it can respond to the request; otherwise, the protocol is *blocking*. Non–blocking protocols are interesting because they can achieve the smallest possible response time. However, these protocols can be constructed only for a restricted class of failures.

In this paper, we consider two kinds of failures in the implementation of non–blocking protocols: *crash failures* in which a server may halt prematurely, and *receive–omission failures* in which a

[1] One would expect that multicasts would prevent this undesirable behavior. Unfortunately, we were unable to run such an experiment as multicasts were not available to us.

server may crash or omit to receive some messages[2]. Crash failures are a reasonable assumption in a system where network partitions and message losses are highly unlikely. On the other hand, receive–omission failures are a reasonable assumption in a system where partitions are still unlikely, but it is possible that some servers may become overloaded and miss messages, or message buffers at these servers may become full and drop messages.

Even though the non–blocking protocols that we consider can achieve the smallest possible response time, it is not clear whether the above restricted set of failures are realistic in practical systems[3]. We, however, believe that most future primary–backup systems will only have to tolerate a restricted enough set of failures that allow our non–blocking protocols to be constructed. Given no extra constraints, a primary–backup system should have all servers on a single local area network. This is because the time required between the failure of a primary and the takeover by a backup is determined by the bandwidth between the primary and the backups. Using a single local area network now makes partitions that separate the servers unlikely. Other failures can still occur due to process crashes and message losses. However, the kinds of message losses that are expected to occur on this network are restricted and correspond to our receive–omission failure model. According to [2], as technology improves and newer, faster networks (such as FDDI) are used, there will be mainly the following two causes for message losses on a local area network:

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

This set of failures corresponds to our receive–omission failure model for which one can construct a non–blocking primary–backup protocol.

3 Description of the implementation

In the following section, we describe the protocols run by the clients and the servers. However, in order to simplify the presentation, many details have been omitted. Further details can be found in [4]. Furthermore, the protocols that we implement do not tolerate permanent message losses between the clients and the primary (although intermittent message losses are tolerated). We discussed in [4] how these protocols can be modified to tolerate permanent message failures. This simplification does not affect our results because all our conclusions are based on the behavior of the protocols when failures do not occur, and this behavior is the same for the protocols we implement and their modified versions.

The service that our protocols implement maintains a *fault–tolerant counter*. Clients request a counter value from the service, and on receiving such a request, the service responds by sending

[2]Non–blocking protocols can be constructed for some other kinds of failures as well. However, to get meaningful results for other failure models, we would need multiple independent links between servers, which were not available.

[3]In fact, most primary–backup protocols that have been mentioned in the literature (for example [9, 10]) are blocking and tolerate more severe failures.

the current counter value to the client and incrementing the counter. Informally, we require the counter service to satisfy the following three properties:

1. Two different requests cannot return the same counter value.
2. No counter value can be skipped.
3. Let request R_1 be sent by client c_1 and request R_2 be sent by client c_2, where c_1 may be the same as c_2. If the response to R_1 was received by c_1 before R_2 was sent by c_2, then R_2 cannot return a smaller counter value than the one returned by R_1.

This is clearly a simple service, yet it is a practical one; for example, such a service is a central component of a multicast transport protocol designed by Apple and Xerox [3]. Furthermore, even this simple counter service is not easy to implement because the counter has to be replicated across servers, and the various copies of the counter have to be kept consistent with each other. Our primary–backup protocol, therefore, was not simplified by using a simpler service semantics. Also, since we only analyze the tradeoffs inherent in primary–backup protocols (and not the tradeoffs due to a particular service), we believe our results are applicable to other service semantics as well.

Our implementation of the above counter service consists of one client protocol and three different protocols for the servers. Two of the server protocols are non–blocking and tolerate crash and receive–omission failures respectively. In order to compare the response time of these protocols with the response time of blocking protocols, the third protocol that we implement is blocking and tolerates transient network partitions. As we will discuss later, we consider the third protocol to be a canonical example of blocking protocols because most existing blocking protocols have a response time performance comparable to ours.

The rest of the section describes these protocols. We assume that there is a known upper bound on the message delivery time and that the servers have approximately synchronized clocks [6, 7, 8]. The implementations are written in C and run on RS/6000 550s with AIX Version 3. The machines were connected over a 16 Mbit token ring and messages were sent using UDP sockets.

3.1 The client protocol

The protocol that the clients run consist of two concurrent threads. The first thread keeps track of the current primary and the second thread initiates requests. In our implementation, clients keep track of the primary in a simple manner—whenever a new server (say s_i) becomes the primary, it sends a "s_i is primary" message to all the clients, and the clients then send subsequent requests to s_i.

The second thread initiates requests from the client. When a client wants a get the next counter value, it constructs a new request (say R), sends this request to the primary, and waits for a response. However, if a timeout occurs and the response has not been received (this happens if a

failure occurred), then the client sends another copy of R (possibly to a different primary). These retries continue until it receives a response.

As can be seen from the above implementation, a client may send multiple copies of the same request to the service. Our implementation, however, requires that the primary be able to distinguish two different requests from copies of the same request. The clients ensure this by attaching an unique identifier (for example a request counter and the client name) to the request R.

3.2 The server protocols

For each of the three implementations, any server runs the protocol given in Figure 1. The procedures **primary** and **backup** remain the same for all implementations, whereas the other two procedures change depending on the failure model. Since the **primary** and **backup** procedures remain the same, we describe these first, and then describe the other procedures for each of the implementations.

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

Figure 1: Protocol run by server s_i

3.2.1 The primary and backup procedures

The server s_i that is the current primary executes the procedure **primary**. It first informs the clients about the new primary by sending a "s_i is primary" message, and then enters into an infinite loop waiting to receive requests from clients. On receiving a request that is not the copy of an earlier request, s_i updates the counter value, **broadcasts** the updated counter (and some information regarding the identity of the request) to the backups, and then sends the response consisting of the new counter value to the client. Informally, the procedure **broadcast** ensures that all (non-crashed) backups receive a copy of the message that the primary sends. Furthermore, the implementation of **broadcast** (either using point-to-point messages or hardware broadcasts) determines whether the protocol is blocking or not, and hence depends on the particular protocol. We describe **broadcast** in detail below when we discuss each of the three protocols.

From the above description, if the primary s_i crashes *after* broadcasting the updated counter and *before* sending the response to some request R, then the client will resend R because it will not get the response to R. Consequently, the server that becomes the primary after s_i (say s_j) must be able to recognize R as a duplicate, because otherwise it is possible that some counter value will be skipped (in particular, the counter value that should have been sent by s_i to the client). The

new primary s_j recognizes the duplicate by keeping a record of the last request that is knows was received by the service from the client (this information is obtained from the broadcast that s_i had sent before crashing) and responds to the client by resending the response which should have been sent earlier by s_i.

All other servers that are not primaries execute the procedure **backup**. The backups receive the new counter values from the primary, and keep their local copies of the counter up to date with the primary copy. In addition, the backups also receive information from the **failure–detector** about the status (that is, crashed or not crashed) of other servers. The backups are ranked, and if the primary crashes, then the backup with the lowest rank takes over as the new primary.

The rest of the section now describes the procedures **broadcast**, **delivery–process** and **failure–detector** for each of the three protocols.

3.2.2 Procedures for the non–blocking protocol tolerating crash failures

Broadcast: For crash failures, this procedure sends a copy of the message to each backup (either using point–to–point messages or hardware broadcasts), and the procedure returns without waiting for any acknowledgements.

Delivery–process: This thread is in an infinite loop and waits to receive messages from the network. Upon receipt of a message, the message is delivered to the server.

Failure–detector: This thread periodically sends a " I am alive" message to all other servers. If such a message is not received from a server, then that server is declared faulty because in this model there can be no message losses. The **failure–detector** at the primary, however, is also responsible for reintegrating new servers. Informally, this is done by informing the existing backups of the new server, and then bringing up the new server with the current counter value. No new requests are processed during the reintegration.

3.2.3 Procedures for the non–blocking protocol tolerating receive–omission failures

Broadcast: This procedure is the same as crash failures and is, therefore, also non–blocking. One might worry that in this failure model, some server may omit to receive the message sent by the primary and later become the primary with an out-of-date counter value. This, however, is prevented by **delivery–process** and **failure–detector** as described below.

Delivery–process: When this thread receives a message from the sender of the broadcast, it first delivers the message and then relays the message to all other servers. If the thread receives a relayed message that it hasn't delivered before, then it delivers that message. As discussed below, the relayed messages will be used by the **failure–detector** to mask receive–omission failures.

This protocol is fairly expensive because for each request (and for each "I am alive" message as well), the protocol sends messages proportional to the square of the number of servers. However, we do not know of any non–blocking protocol tolerating receive–omission failures that sends fewer

messages. As we will see in the next section, these large number of messages play an important role in determining the response time of this protocol.

Failure–detector: As before, all servers periodically send "I am alive" messages to each other and the **failure–detector** at the primary reintegrates new servers when necessary. However, the **failure–detector** is also used by the servers to detect their omission to receive some message (either the new counter value from the primary, or a "I am alive" message from some other server). Again, we omit the details of how this is achieved, and just give an outline. As described earlier, **delivery–process** relays all the messages that it receives. Every server ensures that either it receives at least one copy of every message through a relay, or it detects that it omitted to receive all relays for some message, and halts itself. Thus no server can become the primary with an out-of-date counter value. This failure detection requires that less than half of the servers are faulty. We showed in [5] that this amount of replication is in fact necessary to achieve non–blocking protocols for receive–omission failures.

3.2.4 Procedures for the blocking protocol

We now give the above three procedures for a canonical blocking protocol. This protocol tolerates crashes and transient network partitions.

Broadcast: The procedure first sends the message to all the backups, and then waits for the messages to be acknowledged. Either all the backups acknowledge or a timeout occurs. If a timeout occurs, then another copy of the message is sent to the backups that did not acknowledge the first message. Such retries happen for some fixed number of times. If some backup s_j still does not acknowledge, then it is assumed that s_j has crashed.

Delivery–process: As before, this thread delivers any message that is received. In addition, an acknowledgement is also sent to the sender of the message. Note that this protocol is more efficient in the number of messages sent as compared to the protocol for receive–omission failures.

Failure–detector: Similar to crash failures, except that the "I am alive" messages are acknowledged.

We believe that the behavior of this protocol in a failure-free run can be used to compare most of the blocking primary-backup protocols in the literature with our non–blocking protocols. For example, in the Harp protocol [9], even though the primary only waits for a majority of responses from the backups before proceeding (and thus can be faster than ours), we can still validly compare Harp to our nonblocking protocols. We do this by comparing the response time of our $2k+1$–replica non-blocking protocol with the response time of our k–replica blocking protocol.

4 Tradeoffs between non–blocking and blocking protocols

In this section, we analyze the tradeoffs in failure–free runs between non–blocking and blocking protocols with respect to the response time. In the next section, we discuss how these results can be extended to runs in which failures do occur.

Initially, one would expect that non–blocking protocols will have a better response time because the primary does not have to wait for an acknowledgement from the backups before responding to the clients. However, this may not always be true because, as we saw, some non–blocking protocols can be very message inefficient. On a shared communications medium (like a token ring), these messages may interfere with future requests, thereby increasing the average response time of the protocol. As it turns out, the response time depends on a number of parameters— the degree of replication, the number of clients, the interval between successive client requests, whether the servers use point–to–point communication or hardware broadcasts, etc. The following sections, therefore, analyze the response times for some of these parameters, first for point–to–point communication, and then for hardware broadcasts. Due to space limitations, we only present a subset of our experiments. Our conclusions in the paper, however, are consistent with the experiments that we have omitted.

4.1 Communication using point–to–point messages

In the following sections, we plot the average response time of a protocol versus the degree of replication, while keeping other parameters constant. The maximum degree of replication that we consider is 5 and the maximum number of clients that we consider is 2. This is because we had only a limited number of machines on which we could run our tests. However, for each figure we discuss how we expect the results to extend to higher degrees of replication and to more clients. Also, in these figures, we only show the response time for odd values of n for receive–omission failures, where n is the degree of replication. This is because, as argued in Section 3.2.3, the receive–omission protocol can only tolerate up to $\lceil \frac{n}{2} \rceil - 1$ server failures. Finally, in order to control the frequency of requests that are sent by the clients, we have added another parameter to the client protocol. We call this the *compute–time*, and it corresponds to the amount of time a client waits after receiving the response to a request, before it sends out the next request.

Figure 2 shows the average response time (in the absence of failures) for the three protocols when there was a single client with a compute–time of 10 ms. As a comparison, the response time in the absence of any replication (*i.e.* $n = 1$) was 3.27 ms. As can be seen from the figure, the non–blocking protocols perform better However, surprisingly this was no longer true in Figure 3 where we have added one more client to the system.

The above behavior occurs because of the following reason. As argued earlier, the receive–omission protocol generates a large number of messages. On a shared medium like the token ring, these messages can interfere with future requests sent by the clients, thereby increasing the average

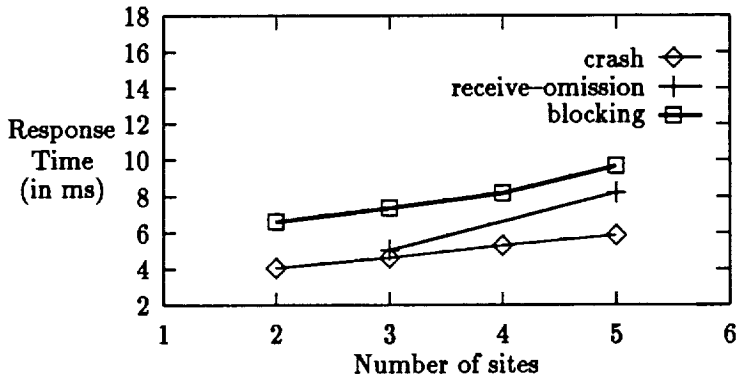


Figure 2: Av. response time v/s degree of replication. Number of clients = 1 and compute–time = 10 ms

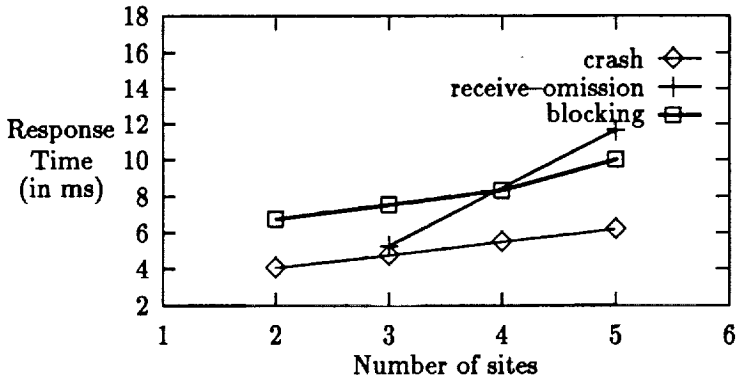


Figure 3: Av. response time v/s degree of replication. Number of clients = 2 and compute–time = 10 ms

response time. In Figure 3, the interference was greater because more requests were sent per unit time. In particular, the interference was large enough to make the response time of the non–blocking receive–omission protocol greater than the response time of the blocking protocol. In fact, Figure 4 shows a run in which we have increased the compute–time to 30 ms. In this case, since less requests are being sent per unit time, interference is less likely, and the average response time becomes smaller again.

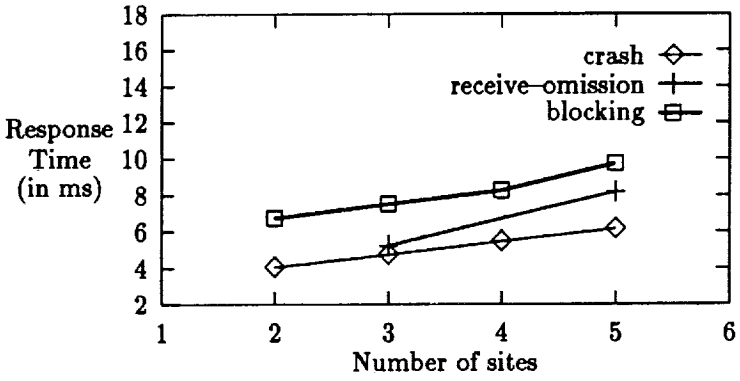


Figure 4: Av. response time v/s degree of replication. Number of clients = 2 and compute–time = 30 ms

Another way to reduce the effect of the interference is as follows. We changed the service slightly so that the clients could send *query* requests in which the clients just queried about the current value of the counter. When the primary receives such a query request, it does not have to inform the backups about any update (there is none) and can immediately respond to the client. Thus, if the clients send a large number of query requests, then the average number of messages sent per request are reduced, reducing the interference. The result is shown in Figure 5. As in Figure 3, there are two clients, each with a compute–time of 10 ms. However, in Figure 5, half of the requests sent by the clients are query requests. As can be seen, the average response time is now smaller.

The results in this section can be extrapolated for higher degrees of replication and more clients. We can expect the graphs of the protocol for crash failures and the blocking protocol to remain essentially the same, increasing more or less linearly with the number of servers, the increase being greater if the number of clients are more. Furthermore, the average response time of the crash failure protocol should always remain less than the response time of the blocking protocol, as the crash failure protocol is very message efficient.

On the other hand, if we add more servers in Figure 2, then we can expect the response time of the receive–omission protocol to eventually become greater than that of the blocking protocol because more servers imply more messages, which then implies more interference. Furthermore,

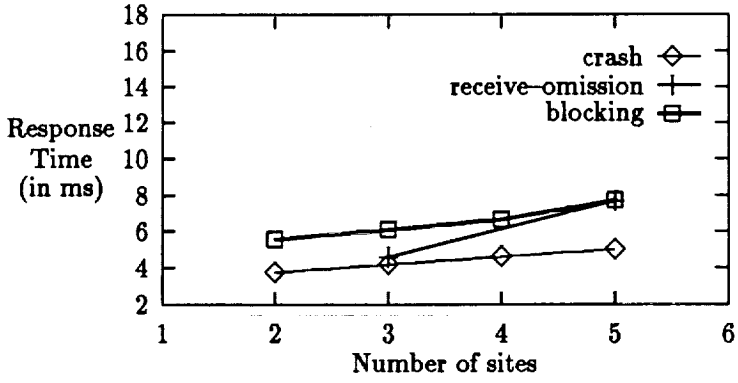


Figure 5: Av. response time v/s degree of replication with query requests. Number of clients = 2 and compute–time = 10 ms

larger the number of clients, earlier the two graphs will intersect. However, if we lower the frequency of the requests (as in Figure 4), then we can expect the interference to become smaller again, which will move the intersection point to higher degrees of replication.

4.2 Communication using hardware broadcasts

In the previous section, all communication between servers was through point-to-point messages. However, the token ring allows more efficient communication using hardware broadcasts. We therefore reimplemented all our protocols so that they use broadcast communication wherever possible.

The new experiments corresponding to the parameters in Figures 2 and 3 are shown in Figures 6 and 7 respectively. In both the figures, the graphs of the two non–blocking protocols are almost indistinguishable. Also, as expected, the response time of most of the protocols is smaller than the corresponding response time for point–to–point communication because less messages were being generated (one message instead of n messages in most cases). A smaller number of messages means that there is less interference. The only exception was the crash failure protocol with $n = 2$. This was because only one message was being sent (to the lone backup) per request, both in the point–to–point and the broadcast case. However, when this message was sent using a hardware broadcast, the primary received its own message, which slowed down the subsequent response. However, what surprised us more was that the non–blocking protocol for receive–omission failures gave a flat curve—the response time was independent of the number of servers. We expected such a behavior for the crash failure protocol because in this case, the primary sends just one broadcast message per request, independent of the number of servers. In the receive–omission protocol, however, each backup relays the primary's message to all the other backups, and therefore the total number of

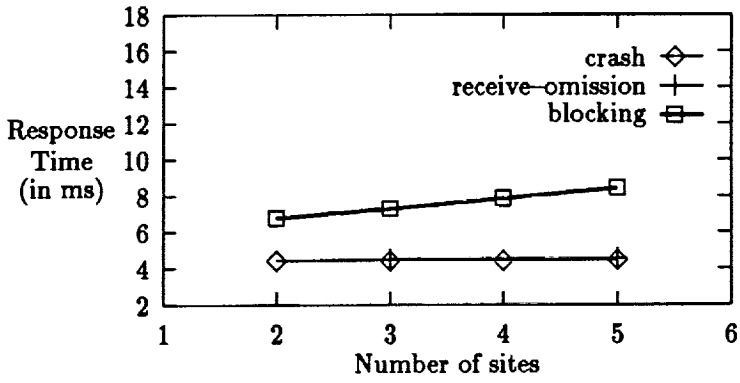


Figure 6: Av. response time v/s degree of replication. Number of clients = 1 and compute–time = 10 ms

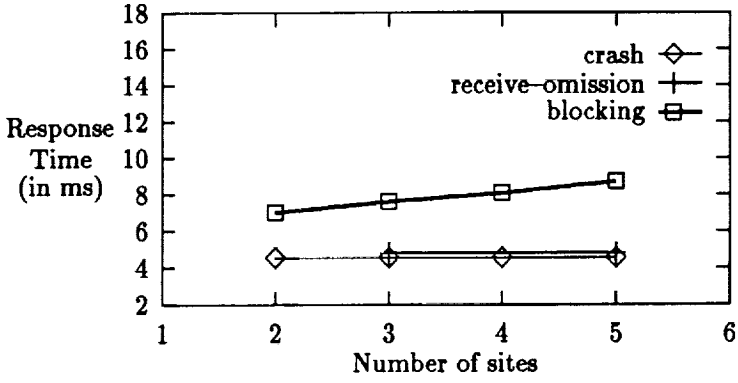


Figure 7: Av. response time v/s degree of replication. Number of clients = 2 and compute–time = 10 ms

relays depend on the number of backups. Thus, more relays should have caused more interference, resulting in higher response times.

The next experiment (shown in Figure 8) showed us the reason for the response time being independent of the degree of replication. We reduced the compute–time to 2 ms, thereby increasing the possibility of interference. We now saw a positively sloped curve for receive–omission failures, which we were expecting earlier. In the earlier figures, the number of messages generated by the backups were not sufficient to interfere with future requests, as a small number of requests were being sent per unit time. On the other hand, when we increased the frequency of the requests, the interference became large enough to increase the response times.

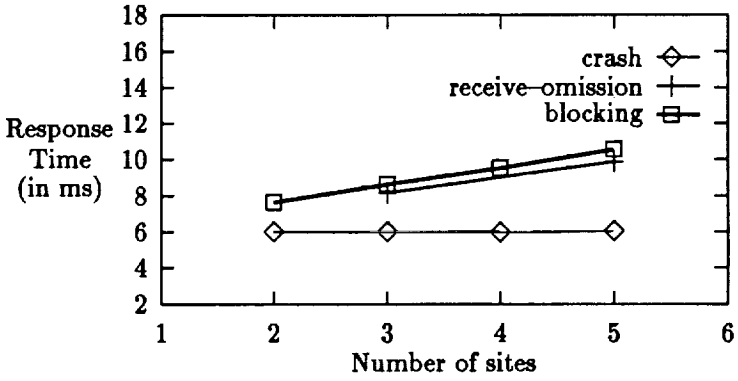


Figure 8: Av. response time v/s degree of replication. Number of clients = 2 and compute–time = 2 ms

Note, however, that the response time of the blocking protocol was never independent of the number of servers. As described in the implementation of the **broadcast** procedure for the blocking protocol, the primary waits for the backups to acknowledge before responding to the client. Thus, the number of messages that the primary waits for increases with the number of servers, causing the response time to increase as well.

We now extrapolate our results to higher degrees of replication and to more clients. Keeping other parameters constant, we should expect the response time of the crash failure protocol to be always independent of the number of servers. This should be the case, because as discussed before, the primary always sends one message per request. However, the exact value of this response time will clearly depend on the number of clients and the frequency of requests. On the other hand, for reasons described in the previous paragraph, the response time of the blocking protocol should increase more or less linearly with the number of servers.

The situation is a little more complicated for receive–omission failures, as our experiments obtained both flat and positively sloped graphs for this protocol. Combining with the results in

Figure 8, one can expect the response times in Figures 6 and 7 to eventually start increasing, possibly as a step function, as the number of servers increase. A similar behavior should be seen if we increase the number of clients. However, unlike the point–to–point case, the response time of the receive–omission protocol should always be less than the response time of the blocking protocol. This is because the former never sends more messages than the latter. However, if our blocking protocol had been like the Harp protocols in which only a majority of acknowledgements are required, then there may be a crossover point after which the blocking protocol performs better.

5 Extending the results to runs with failures

We did not consider failures in the previous section because the results in the case of failures will depend on the underlying system and the particular service being implemented. For example, the amount of time it takes to reintegrate a new server usually depends on the service semantics and the amount of information that has to be sent to the new server[4]. Similarly, the *failover time* (*i.e.* the amount of time that there can be no primary) depends on the maximum message delivery time. However, we can still draw some useful conclusions about runs with failures from our results.

If failures are rare and the non–blocking and blocking protocols have similar failover and reintegration times, then it is easy to see that the results in the previous sections should still be qualitatively true. These results can also be used in deciding the right degree of replication to be used when implementing these protocols. If servers use point–to–point communication, then we saw that the average response time of all protocols (non–blocking and blocking) rises with the number of servers. Therefore, if failures are infrequent (which is usually the case), and reintegration does not cost too much (which was also true in our counter service), then in designing a primary–backup service that uses point–to–point communication, one should keep the degree of replication as low as possible, and reintegrate new servers only when the older servers have crashed. This should result in a smaller average response time since the degree of replication is always kept low.

On the other hand, if communication can be done using hardware broadcasts, then we saw that the response time of non–blocking protocols can be independent of the number of servers. In this case, it may be possible to keep the degree of replication for these protocols high (since this does not increase the response time), and save on some reintegration cost as well, further reducing the response time. We can achieve this by not doing any reintegration until a large number of servers have crashed, and then reintegrating more than one server at a time, thereby saving on the total reintegration time. This clearly assumes that servers can be reintegrated in parallel without too much additional expense.

[4]In our case, reintegration took almost no time because just the current counter value and some information regarding the last request sent by each client needed to be sent to the new server.

6 Discussion

In this paper, we looked at a subclass of primary–backup protocols, called non–blocking protocols. These non–blocking protocols are interesting as they have the potential of achieving relatively small response times, as the primary does not have to wait for any acknowledgements from the backups before responding to the client. However, surprisingly, we saw that even though a small response time could be achieved in many cases, it was not true in general. The actual response time depended on many other system parameters—- the underlying communications pattern, the number of servers and clients, and the frequency of client requests.

In particular, we found that if servers used point–to–point communication over a shared medium, and if the number of clients or servers was large enough, then the non–blocking receive–omission protocol could have a larger average response time than the blocking protocol. This was primarily due to the large number of messages sent by this protocol, leading to contention on the network. If we could reduce this contention by keeping the number of clients or servers small, or by changing the frequency or mix of client requests, then the protocol performed much better.

On the other hand, if servers used hardware broadcasts to communicate, then the contention was removed to a large extent, and the non–blocking protocols performed better than the blocking protocol in all cases. In fact, for small degrees of replication, the response times of the non–blocking protocols even became independent of the number of servers.

As we said earlier, non–blocking protocols can only be built for restricted classes of failures. However, we expect that these restricted classes of failures will be the dominant ones for future systems, and our results show that in such systems these protocols should be preferred in many cases as they can achieve a smaller response time than the traditional blocking protocols. On the other hand, since some of these non–blocking protocols can be fairly message inefficient, one needs to take many system parameters into consideration before designing such a primary–backup service.

References

[1] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 627–644, October 1976.

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

[3] Susan M. Armstrong, Alan O. Freier, and Keith Marzullo. Multicast transport protocol. Internet RFC 1301, Feb 1992.

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

[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 Applicaions*, pages 187–198, Mondello, Italy, September 1992.

[6] Falviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[7] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

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

[9] 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.

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