Implementing Adaptive Fault-Tolerant Services for Hybrid Faults[*]

Li Gong and Jack Goldberg

SRI International
Computer Science Laboratory
Menlo Park, California 94025 U.S.A.
{gong,goldberg}@csl.sri.com

March 22, 1994
(Revised September 30, 1994)

Abstract. The two major approaches to building fault-tolerant services are commonly known as the Primary-Backup approach (PB) and the State-Machine approach (SM). PB can tolerate crash and omission faults and runs more economically than SM, but SM can tolerate more serious faults, including arbitrary or Byzantine faults. Instead of selecting one or the other approach, thus either incurring a high running cost or risking the service becoming incorrect when unexpected faults occur, we advocate the approach of adaptive fault tolerance. We present algorithms that intelligently adapt between PB and SM, thus retaining (almost) the best of both worlds. Our adaptive approach is modular in that any PB or SM protocol can be used, and is also practical in that it can be easily incorporated into some existing systems.

Index Terms. Fault tolerance, primary backup, state machine, manifest faults, Byzantine faults, hybrid faults, adaptivity, distributed systems services, algorithm complexity.

1 Introduction

In building fault tolerance services in a distributed system, there are two major approaches, namely, the Primary-Backup approach (PB) (e.g., [1, 9]) and the State-Machine approach (SM) (e.g., [19, 16]). Each approach has its distinctive advantages. To tolerate simple faults such as crash and omission, PB protocols are generally significantly cheaper than SM protocols in terms of the numbers of processors, messages, and rounds (which directly affects the service response time). PB protocols are also much simpler than SM protocols, and thus the efforts of debugging or formal verification of PB protocols are also easier. On the other hand, in choosing to run a PB protocol instead of a SM protocol, one risks providing incorrect service functions or values, which may cause the overall system to fail, in the face of more serious *types* of faults such as arbitrary (Byzantine)

[*]This work was supported in part by the U.S. Air Force under USAF contract F30602-92-C-0098.

faults[1]. Therefore, it is common practice for critical applications to run a SM protocol, possibly using Byzantine agreement [13]. The high cost of running such a protocol is compensated by the belief that all possible faults (up to a certain number) are adequately tolerated.

Instead of being forced to make a design choice between using SM or PB, thus either incurring a high running cost or risking system failure when unexpected faults occur, we advocate an approach of adaptive fault tolerance [12]. Given that in many situations Byzantine or other nontrivial faults occur only relatively infrequently, we develop intelligent adaptive algorithms, using PB and SM protocols as building blocks, that runs typically at a cost close to that of a PB protocol and switches to a more expensive SM protocol only as complicated faults (which cannot be tolerated by a PB protocol) occur. This adaptive approach thus has the potential to retain the best of both worlds. In addition, our adaptive approach is modular in that any PB or SM protocol can be used.

For noncritical applications, our approach may be seen as a way of extending PB to cover more complex faults at low additional cost. For critical applications, our approach may be seen as a way of allowing some of the processing resources required for SM to be used for other services when full SM functionality is not needed. For example, when only manifest faults occur, an adaptive algorithm runs in the PB mode and can thus tolerate a maximum number of such benign faults. The adaptation can also be viewed as between an optimistic algorithm and a pessimistic one where the former is the default mode of operation and the latter is invoked only when necessary.

In the rest of this paper, we first outline a general strategy of adaptation for handling hybrid faults. We then present two adaptive fault tolerance algorithms, analyze their correctness and complexity, and compare them with nonadaptive approaches. We conclude with a summary and suggestions for future research.

2 An Adaptation Strategy

System functions can be concentrated in some central location or distributed around a network, and the software for these functions can consist of modules on separate processors or can be more closely integrated. Conceptually, however, a fault-tolerant service generally contains some or all of the following functions: processing of requests, fault forecast, fault detection, fault masking, fault diagnosis, fault removal, repair, and reintegration of repaired components.

To explain our general adaptation strategy, suppose that in the course of operations, faults of two types, A and B, may occur. Also suppose that type A faults occur more frequently and are less expensive to tolerate than type B faults. If both types of faults must be tolerated, the traditional approach has been to assume the worst and constantly run an (expensive) algorithm that can handle both types of faults.

We observe that detecting a fault is in general less expensive than tolerating it. Based on this premise, our strategy is to run, as a default, an algorithm that can tolerate type A faults and can also reliably detect the occurrences of type B faults. When type B faults occur, the default

[1]Any given system configuration can tolerate only up to a certain *number* of faults. The emphasis here is on the distinction that a PB protocol cannot tolerate Byzantine faults.

algorithm switches to a more expensive one that can tolerate both type A and type B faults. Some decision procedure then decides when to switch back to the default algorithm. For example, when the occurrences of type B faults are bursty according to a fault forecast, it may be wise to continue running the expensive algorithm for some period of time.

If the difference in the cost of tolerating the two types of faults is significant, such as in the case of simple crash faults versus Byzantine faults, then an adaptive strategy gains a great deal by reducing the average running cost. The strategy is at its best if (1) the cost of adding the extra fault detection mechanism to an algorithm that tolerates type A faults is negligible (so that the service efficiency is near optimal when type B faults do not occur) and (2) the default algorithm or the fault detection algorithm forms the initial segment of the more expensive algorithm (so that nothing is lost when type B faults do occur). The next section gives adaptive algorithms that exhibit such desirable behaviours. The strategy can be extended to handle faults of multiple types, in this case a more elaborate fault diagnosis mechanism (especially the online variety) is needed to determine the exact types of fault in order to direct the adaptation.

3 Two Adaptive Fault Tolerance Algorithms

Two algorithms are described that adapt between the primary-backup approach and the state-machine approach. But first, we need to explain the assumptions we make about the execution environment of our adaptive algorithms.

3.1 System Model

The environment we assume is the following. Clients send their requests to the servers who process the requests and respond, all by exchanging messages. For simplicity, we assume that the communication channel between a client and any server is reliable and FIFO, and we aim to tolerate faulty servers but not faulty clients. We also assume that the servers are deterministic – because in the state-machine approach it is usually undesirable to allow nondeterministic behaviours in the (correct) servers. Moreover, we assume that the system is synchronous, and thus we can use a model of computation based on rounds. The reason for this limitation is that it is impossible to guarantee both safety and liveness in asynchronous systems [6, p.19].

Following the literature, we classify faults into three categories [14]:

- Manifest fault – one that produces detectably missing values (e.g., crash and omission faults) or that produces a value that all nonfaulty recipients can detect as bad (e.g., it fails checksum or format or typing tests).
- Symmetric fault – one that delivers the same wrong value to every nonfaulty receiver.[2]

[2] It will become clear later that we can weaken this definition to that all nonfaulty recipients receive some wrong values, although they may not receive the same wrong value.

- Asymmetric fault – an arbitrary fault with no constraints, also known as Byzantine fault.

We assume that the reader is familiar with both primary-backup and state-machine approaches, and omit non-essential details of the algorithms. Briefly, in PB, one and at most one server is designated as the primary at any time. A client sends a request to the primary, who processes it and then broadcasts the necessary state change to all backup servers. In a nonblocking PB protocol, the primary server responds to the client before receiving acknowledgements to its broadcast whereas in a blocking protocol, the primary blocks until all backups have acknowledged or after a timeout period. The schema for a server consists of three modules for: (1) deciding whether it is a primary or a backup, (2) processing requests, and (3) fault detection and recovery [6, p.56]. It is apparent that the PB approach can tolerate only manifest faults. For example, an incorrect primary can broadcast an incorrect state change and backup servers cannot detect this fact because they do not know the client's service request.

In a SM protocol, a client broadcasts its request to all servers, and then takes a vote on the responses it receives. Therefore, the client will decide on the correct response if a majority of the servers are nonfaulty. For correctness, all nonfaulty servers must process requests (possibly from multiple clients) in the same order. This requirement is called replica coordination [16] and is not necessary in a PB protocol. Satisfying this coordination requirement is quite expensive – for example, a Byzantine agreement protocol is a typical solution. With this heavy cost in resources and performance, the SM approach gains the ability to tolerate symmetric as well as asymmetric faults, in addition to manifest faults.

In our adaptive algorithms given below, the high-cost Byzantine agreement machinery is used only when needed, thus we call these algorithms Byzantine-On-Demand or BOD.

3.2 Manifest versus Symmetric Faults

Our first adaptive algorithm BOD-1 is to tolerate both manifest and symmetric faults, but not asymmetric faults. For the moment, we assume that links connecting servers are nonfaulty.

Given any PB protocol (blocking or nonblocking), we need only make a few simple changes to make it adaptive. We assume the servers have implemented a Byzantine agreement (BA) protocol – for agreeing on the next client request for processing, or replica coordination – that can handle symmetric faults. Strictly speaking, any protocol that can mask symmetric faults is sufficient for BOD-1, but for convenient discussion, we always refer to a BA protocol.

The basic idea is to let the backup servers participate in the service passively as in the PB protocol, except that they now also receive the original request from the client and watch the primary for any inconsistency (compared with themselves). If they detect an inconsistency, they report an error to the client (who will then wait for further action on the part of the servers) and initiate a Byzantine agreement protocol among the servers to mask the error.

Notice that since we are using a primary-backup approach as default, it is important, from the viewpoint of providing a correct service to the client, to detect non-manifest faults only in the primary, from whom the client takes a response. Nonmanifest faults in backups will lower the overall degree of fault tolerance (for additional faults) but can be safely ignored for the meantime because once such a faulty backup becomes a primary, its erratic behaviour will be immediately detected. It is this property that makes the adaptive algorithm so cost-effective.

Nevertheless, it may not be desirable to allow a significant proportion of backups to become faulty because the chance of detecting faults in the primary and the ability to replace it will be reduced. Faulty backup servers can be dealt with by an additional fault diagnosis and removal mechanism. For example, if only one backup disagrees with the primary, then this backup must be faulty (on the assumption that the majority is nonfaulty) and can immediately be removed and repaired.

The outline of the BOD-1 algorithm is in Table 1. We use r to denote a request, a(r) for the response, and s-c for information regarding state change.

```
Round 0. Client:  Broadcast request r to all servers.
         Primary: Wait for request from client.
         Backup:  Wait for request from client.

Round 1. Client:  Wait for response from primary.
         Primary: Broadcast (r, a(r), s-c) to all backups.
                  Respond a(r) to client.
         Backup:  Queue r. Wait for message from primary.

Round 2. Client:  Wait for error report from backup.
         Primary: Wait for error report from backup.
         Backup:  Verify the correctness of a(r).
                  If error, broadcast ERROR to client and all servers,
                  and start BA protocol (to agree on which client request to process).
                  Wait for error report from other backups.

Round 3. Client:  If receive ERROR from a majority of servers,
                  wait for the BA protocol to complete; then vote on the responses.
                  Otherwise, accept a(r).
         Primary: If receive ERROR from a majority of servers, switch to the BA protocol,
                  process request and respond to client.
                  Return to Round 1.
         Backup:  If receive ERROR from a majority of servers, switch to or continue with
                  the BA protocol, process request and respond to client.
                  Otherwise, terminate BA protocol it started earlier, if any.
                  Return to Round 1.
```

Table 1: Byzantine-On-Demand Algorithm BOD-1

A few points need to be clarified about the algorithm. For simplicity, we have omitted some details of the PB protocol, especially its failure detector and the handling of manifest faults. The mechanism for detecting (and masking) symmetric fault in BOD-1 is *in addition to* the failure detector of the PB protocol. In theory, either detection mechanism can take precedence over the other. For example, if manifest and symmetric faults occur concurrently, we can deal with the manifest faults first (e.g., arranging a new primary) and the symmetric faults later. Or we can mask the symmetric faults first and handle the manifest faults later. The latter scheme has the advantage that the client receives responses earlier than in the former scheme. Variations are possible. For example, if both fault detection mechanisms detect faults related to the primary, a symmetric fault can be assumed and the BA protocol initiated.

We have used a nonblocking PB protocol in BOD-1 such that the primary responds to the client without waiting for acknowledgement from the backups. In such a case, it is important that the primary's response to the client and its broadcast to the backups must be in the same round because otherwise, a fault may occur when the primary is responding to the client so that the client receives an incorrect response while all backups receive the correct response (and thus do not complain). Obviously, any blocking PB protocols will also work in this framework and our algorithm BOD-1 needs only minor modifications (which we do not go into here).

In addition, we have assumed that the client always expects to get the response a(r) from the primary. Some PB protocols are "pass-the-buck" in that the response always comes from a different server [6, p.100]. Our framework can also accommodate these protocols.

If the primary has failed manifestly, the identity of the new primary is decided according to the PB protocol and is conveyed to the client. When the identity of the primary is in dispute, a non-manifest fault has occurred and a BA protocol can be used to reach an agreement. It is not difficult to see how this additional BA protocol can be added, so we will not discuss this issue in more detail.

A backup server checks for two new errors in **Round 2**. One is that the primary processes a client's request not according to a FIFO order. The other is that the primary's response is wrong. A backup server can check the first error by looking at its local queue of requests, and can check the second by taking the request the primary broadcast, processing it, and comparing its own result with the one sent by the primary. A discrepancy signifies that a symmetric fault has occurred. Note that when a server broadcasts an error message, it can use the *same* message to carry out the next step of the BA protocol, instead of waiting till **Round 3**. We do not discuss such optimizations.

In BOD-1, a server does a majority voting on error messages before deciding to switch to the BA protocol. This is because of the assumption that the links between the servers are nonfaulty, thus a symmetric error will be detected and reported by all nonfaulty servers. Without this assumption, a server may need to switch to the BA protocol even when receiving just one error report.

A backup server that has detected an error in **Round 2** immediately starts the BA protocol. This is the earliest possible time to convert to running the BA protocol. Under the assumption that links connecting the servers are nonfaulty, if the primary is faulty, then all nonfaulty backups will have started running the BA protocol in **Round 2**. Thus it may appear to contribute little for the primary and the faulty backups to start catching up to running the BA protocol in **Round 3**. However, a server experiencing a transient fault in **Round 2** may have recovered by **Round 3** and thus could be quite useful in the vote. If the primary is not faulty, then a backup server will receive error report from only a minority of servers in **Round 3**, and those who have started running the BA protocol earlier should terminate it. Without this assumption of nonfaulty links, a server need not vote on error report and must continue to complete the BA protocol.

In BOD-1, at the end of the protocol, servers return to start from the beginning, in the default PB mode. This can be changed easily. For example, if the occurrence of non-manifest faults has been frequent for a period of time, the algorithm can stay in the SM mode for a while before returning to the PB mode. Here, fault forecast and heuristic methods can be useful.

Finally, the adaptive algorithm imposes no ordering among the processing of requests from multiple clients. The primary is free to choose the next request to process, as long as the order among requests from the same client is FIFO. The backups simply follow the primary's lead. This arrangement satisfies the Replica Coordination requirement [16], and is crucial for keeping the cost down. This is in the same spirit of the coordinator-cohort scheme [3, 5]. Any additional ordering can be enforced with other methods, which are beyond the scope of this paper.

Proof of correctness. We give a proof outline by enumerating all cases. (1) If there is no fault, then the protocol terminates essentially as the PB protocol. (2) If the primary and backups exhibit only manifest fault, then the PB protocol's fault detection and recovery mechanism handles these faults. (3) If the primary exhibits a symmetric fault, then some nonfaulty backups (or all nonfaulty backups, if there are no link faults) will detect the fault (by the additional fault detection mechanism in BOD-1) and report to the client. A subsequent BA protocol will mask this fault and the client can obtain the correct response by simple majority voting [16]. (4) If the primary is nonfaulty, then at most a minority of backup servers will report error (note that no protocol can tolerate a majority of servers being non-manifestly faulty), and these error messages are false alarms and rightly ignored. □

Analysis of complexity. Compared with the PB approach, when there are no faults, BOD-1 requires m extra messages in the client's initial broadcast, m being the number of backup servers. BOD-1 also uses one more round than a nonblocking PB protocol, the same number of round as a "pass-the-buck" PB protocol, but one fewer round than a blocking PB protocol. The primary's broadcast message in Round 2 is slightly longer (it contains both the request and response), and the backups will all have to process the request individually (thus consume more CPU cycles). Therefore, BOD-1 is slightly more expensive than a typical PB protocol, and this is quite reasonably compensated by the fact that symmetric faults can now be detected and masked.

If there are only manifest faults, then a PB protocol requires more rounds to recover. Thus the overall response time for BOD-1 is no worse than the PB protocol it uses as default, and the only additional expense in BOD-1 is the client's initial broadcast.

When symmetric faults occur, all nonfaulty servers in BOD-1 convert to running a BA protocol (or whatever algorithm that can tolerate this type of fault) in **Round 2**, assuming no link faults, but with some overhead. Compared with the state-machine approach, BOD-1 uses one more round because in SM the BA protocol can start in **Round 1**. However, the client's broadcast and that of the primary in the first two rounds of BOD-1 are not wasted – they can be used as part of the early rounds of the BA protocol – and the only extra messages are those reporting errors to the client. On the other hand, if the state-machine approach is used as default, then even when no fault or only manifest faults occur, the running cost is significantly higher than that of BOD-1. □

In Table 2 below, we compare the algorithm complexity of blocking PB (denoted as bPB, and including "pass-the-buck" protocols), State Machine (SM), and BOD-1. We give only the difference between the complexity of BOD-1 and that of other algorithms because the absolute complexity varies depending on the PB or SM protocol we use as building blocks. For brevity, we do not include nonblocking PB protocols because their running cost differs from "pass-the-buck" protocols only in that nonblocking protocols use one fewer round. Suppose there are a total of m backup servers.

	bPB		BOD-1		SM	
	msgs	rounds	msgs	rounds	msgs	rounds
fault-free	msg(bPB)	r(bPB)	msg(bPB)+m	r(bPB)	msg(SM)	r(SM)
manifest	msg(bPB)	r(bPB)	msg(bPB)+m	r(bPB)	msg(SM)	r(SM)
symmetric			msg(SM)+m	r(SM)+1	msg(SM)	r(SM)

Table 2: Complexity Comparison

We can clearly see that when no fault occurs, BOD-1 uses one more broadcast (from client to all the servers) and one more round than the cheapest nonblocking PB protocol. However, nonblocking protocols need additional mechanisms for error recovery, such as checkpointing, and cannot handle send-omission and general-omission faults. Also, in a fault-free run, some blocking protocols (such as "pass-the-buck" protocols) use just one more round than nonblocking protocols, while other blocking protocols use more rounds. When only manifest faults occur, BOD-1 uses one more broadcast than either nonblocking or blocking Primary-Backup, but no more rounds. When symmetric faults occur, BOD-1 is as slightly more expensive than the SM protocol.

Since SM is much more expensive than PB, but non-manifest faults occur relatively rarely, the adaptive algorithm BOD-1 is superior than the PB approach in that non-manifest faults can now be tolerated and also superior than the SM approach in that the average running cost is greatly reduced. It should be pointed out that in our discussion we use Byzantine agreement protocol to tolerate symmetric faults, so the comparison in cost in Table 2 may be a little unfair because a cheaper protocol may also tolerate such faults. However, it is intuitive that any method that can tolerate symmetric faults will likely be significantly more expensive than the PB approach, and thus adaptive algorithms similar to ours will likely be beneficial.

3.3 Manifest versus Asymmetric Faults

BOD-1 cannot handle asymmetric faults. For example, the primary can send correct responses to all backup servers but send a wrong one to the client. Therefore, the client cannot rely on the primary's response as before, and it is not enough for backup servers to watch the primary.

Our second algorithm BOD-2 is a simple modification of BOD-1 and can tolerate manifest and asymmetric faults. As before, we assume the availability of a PB protocol (blocking or nonblocking) and a Byzantine agreement (BA) protocol.

As can be seen in Table 3, there are only two major differences between BOD-1 and BOD-2. First, when a backup server is satisfied with the primary's response, instead of remaining silent, it sends the response back to the client as well. Second, the client votes on all the responses and reports an error (to initiate the BA protocol) if the primary's response is not the majority vote of all responses from the servers. Note also that it is no longer meaningful to vote on error reports since faults can be arbitrary.

Round 0.	*Client*:	Broadcast request r to all servers.
	Primary:	Wait for request from client.
	Backup:	Wait for request from client.
Round 1.	*Client*:	Wait for response from primary.
	Primary:	Broadcast (r, a(r), s-c) to all backups. Respond a(r) to client.
	Backup:	Queue r. Wait for message from primary.
Round 2.	*Client*:	Wait for a(r) or error report from backup.
	Primary:	Wait for error report from backup.
	Backup:	Verify the correctness of a(r). If correct, respond a(r) to client.
		If error, broadcast ERROR to client and all servers,
		and start BA protocol (to agree on which client request to process).
		Wait for error report from other backups.
Round 3.	*Client*:	If receive an error report,
		wait for the BA protocol to complete; then vote on the responses.
		If no error is reported but primary's a(r) is not the majority vote
		of the a(r)'s, broadcast to all servers to initiate BA protocol.
		Otherwise, accept a(r).
	Primary:	If receive an error report, switch to the BA protocol, process request,
		respond to client, and return to **Round 1**. Otherwise, go to **Round 4**.
	Backup:	If receive an error report, switch to or continue with the BA protocol,
		process request, respond to client, and return to **Round 1**.
		Otherwise, go to **Round 4**.
Round 4.	*Primary*:	Wait to see if client report error. If error, switch to BA protocol.
		Otherwise, return to **Round 1**.
	Backup:	Wait to see if client report error. If error, switch to BA protocol.
		Otherwise, return to **Round 1**.

Table 3: Byzantine-On-Demand Algorithm BOD-2

The correctness argument for BOD-2 is similar to that of BOD-1 – any non-manifest fault is detected and a Byzantine agreement protocol is initiated to mask it.

The complexity of BOD-2 is higher than BOD-1. In a fault-free run, an extra m messages will be needed in Round 2 (for the "backup" servers to respond to the clients). When only manifest faults occur, it is still much cheaper than a full-fledged state-machine approach in that backup servers simply follow the lead by the primary in deciding the next request to process. This arrangement eliminates the need for extra effort to satisfy the Replica Coordination requirement [16]. When asymmetric faults occur, BOD-2 may use one more round than BOD-1, for example, when the backup servers have to wait till **Round 4** to decide whether to switch to the BA protocol. However, if the client does not report error in **Round 3**, it can complete the protocol after **Round 3**, earlier than the servers.

4 Related Work

Our work is undertaken within the general framework outlined in [12] and can be viewed as a realization of some of the principles of adaptive fault tolerance. We have made heavy use of materials on primary-backup protocols [9, 8, 6] and the state-machine approach [16]. In particular, our adaptive algorithms use those protocols as building blocks, in a modular fashion.

Previous work on handling hybrid faults appears to focus on extending protocols for Byzantine agreement so that they can tolerate a higher number of benign or hybrid faults (e.g., [14, 15, 17]) than a standard Byzantine agreement protocol. These algorithms typically can tolerate as many Byzantine faults as possible (bounded by one third of the number of processors [13]). However, when other non-manifest faults do not occur, the algorithm by Thambidurai and Park [17] cannot tolerate many manifest faults whereas our algorithms can tolerate a maximum number of manifest faults because they will be running a Primary-Backup protocol. The algorithm by Lincoln and Rushby [14] can tolerate a maximum number of manifest faults but, like the algorithm by Thambidurai and Park [17], it is non-adaptive in that the number of rounds of each execution of the protocol is decided in advance so the complexity of the protocol does not decrease when no or fewer faults occur. Moreover, the complexity of such algorithms is typically comparable to that of Byzantine agreement because they aim to tolerate arbitrary faults, including symmetric and asymmetric faults, all the time. In contrast, our adaptive algorithms are much less expensive because for most of the time they merely attempt to detect arbitrary faults, and activate the heavy machinery to tolerate arbitrary faults only *as they occur.*

Our adaptation strategy is in flavour similar to early-stopping protocols (e.g., [10, 7, 2]). The complexity (i.e., numbers of messages and rounds) of these protocols is proportional to the number of actual faults occurring instead of the maximum number of faults that can be tolerated. In other words, the protocols terminate earlier if fewer faults occur. This line of work has largely been focused on Byzantine-agreement-type problems, so the protocols are adaptive only to the extent of the number of actual faults, not distinguishing the types of faults. Therefore, they are usually much more expensive than our algorithms, which take advantage of the common observation that in most applications Byzantine faults occur only infrequently. Nevertheless, our algorithms can use early-stopping Byzantine agreement protocols or those above for hybrid faults to further increase efficiency.

Garay and Perry [11] recently proposed a continuum of failure models with crash-only faults and

Byzantine faults at the extremes. This can be taken as a combination of early stopping and dealing with hybrid faults. Although their model is flexible in that a design does not have to choose one of the two extremes, their method again concentrates on solving agreement-type problems. Our algorithms, on the other hand, are aimed at building general fault-tolerant services. In particular, the algorithms adapt between two different approaches, namely primary-backup and state-machine, and utilize to the maximum the efficiency of a primary-backup protocol.

There have been efforts to evaluate the relative merits of various fault tolerance techniques for different applications (e.g., [18]), especially following the recent precise formulation and analysis of the widely used primary-backup approach [9, 8, 6]. Our work provides some insight in that one can adaptively use these different approaches and retain (almost) the best of both worlds.

Finally, since our adaptation is modular in that it uses existing primary-backup and Byzantine agreement protocols as building blocks, our algorithms are conceptually simple. An important benefit is that the correctness proof is much simpler than that for earlier protocols for handling hybrid faults. We need only show that the adaptation correctly detects and adapts to the occurrence of certain types of faults. The potential reduction in the effort of formal verification is then significant.

5 Summary and Future Work

We have shown how to apply the principles of adaptive fault tolerance to handle hybrid faults. We have presented algorithms that intelligently adapt between the Primary-Backup (PB) approach and the State-Machine (SM) approach. Such an adaptive algorithm runs a default PB protocol but also attempts to detect the occurrence of non-manifest faults. When these fault occur, a Byzantine agreement type protocol is activated to mask the faults. Given that in practice manifest faults (e.g., crash and omission faults) are the most common ones, our adaptive approach is more cost-effective than the traditional "one or the other" approach because it retains (almost) the best of both worlds. Our approach is modular in that any specific PB or SM protocol can be plugged in. This keeps our algorithms conceptually simple, and it also significantly reduces the complexity of the correctness argument and the effort of formal verification.

There are many directions for future research. One is to investigate optimizations of our algorithms. For example, given an assumption of the number of possible faults in a period of time (such as predicted by the fault forecast component), some messages in BOD-1 may not need to be broadcast to all backup servers. This is because non-manifest faults are symmetric, so only one nonfaulty server is needed to detect a fault, depending on the assumptions of link faults. Some simulation may also provide fresh insights.

Another is to examine other possible adaptations, such as between symmetric and asymmetric faults, and between various manifest faults. As we already mentioned, other mechanisms for fault diagnosis and fault removal can be integrated. They can run parallel to our algorithms. To reintegrate a repaired component, if only symmetric faults are possible, algorithms can be developed so that the repaired server obtains state information from a small number of existing servers. If asymmetric faults are possible, then those methods mentioned in [16] can be used. Other aspects of adaptation, such as fault transparency to clients and the cost to repair damaged servers, may also be worth investigating.

Our adaptive algorithms not only offer some theoretical insight into the relationship between the primary-backup and state-machine approaches to implementing fault-tolerant services, they also appear to be very practical. For example, given the existing support for process groups and virtual synchrony in the ISIS/Horus system [4], it should not be difficult to add an adaptation facility so that non-manifest faults can be tolerated as needed.

References

[1] P.A. Alsberg and J.D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, October 1976.

[2] P. Berman, J.A. Garay, and K.J. Perry. Optimal Early Stopping in Distributed Consensus. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 221–237, Haifa, Israel, November 1992. Springer-Verlag.

[3] K.P. Birman. Replication and Availability in the ISIS System. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, volume 19(5) of *ACM Operating Systems Review*, pages 79–86, December 1985.

[4] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53/103, December 1993.

[5] K.P. Birman, T.A. Joseph, T. Raeuchle, and A. El Abadi. Implementing Fault-tolerant Distributed Objects. *IEEE Transactions on Software Engineering*, 6(11):502–508, June 1985.

[6] N. Budhiraja. *The Primary-Backup Approach: Lower and Upper Bounds.* Ph.d. dissertation, Cornell University, Ithaca, New York, June 1993.

[7] N. Budhiraja, A. Gopal, and S. Toueg. Early Stopping Distributed Bidding and Applications. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, volume 486 of *Lecture Notes in Computer Science*, pages 304–320, Haifa, Israel, September 1990. Springer-Verlag.

[8] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Optimal Primary-Backup Protocols. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 362–378, Haifa, Israel, November 1992. Springer-Verlag.

[9] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *Proc. 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, pages 187–196, Sicily, Italy, September 1992.

[10] D. Dolev, R. Reischuk, and H.R. Strong. Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[11] J.A. Garay and K.J. Perry. A Continuum of Failure Models for Distributed Computing. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 153–165, Haifa, Israel, November 1992. Springer-Verlag.

[12] J. Goldberg, I. Greenberg, and T.F. Lawrence. Adaptive Fault Tolerance. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 127–132, Princeton, New Jersey, October 1993.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[14] P. Lincoln and J. Rushby. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. In *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pages 402–411, Toulouse, France, June 1993.

[15] F.J. Meyer and D.K. Pradhan. Consensus with Dual Failure Modes. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):214–222, April 1991.

[16] F.B. Schneider. Implementing Fault-Tolerant Services Using the State-Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[17] P. Thambidurai and Y.K. Park. Interactive Consistency with Multiple Failure Modes. In *Proceedings of 7th IEEE Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, Ohio, October 1988.

[18] A. Waterworth, P.D. Ezhilchelvan, and S.K. Shrivastava. Understanding the Cost of Replication in Distributed Systems. Technical report, Computing Laboratory, University of Newcastle upon Tyne, U.K., January 1993.

[19] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.