Consistent Failure Reporting in Reliable Communication Systems*

Kenneth P. Birman
Bradford B. Glade

TR 93-1349
May 1993

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

*The authors are in the Department of Computer Science, Cornell University, and were supported under DARPA/ONR grant N00014-92-J-1866, and by grants from IBM, HP, Siemens and GTE.

Consistent Failure Reporting in Reliable Communication Systems*

Kenneth P. Birman and Bradford B. Glade

May 19, 1993

Abstract

The difficulty of developing reliable distributed software is an impediment to applying distributed computing technology in many settings. This paper reviews some common platforms for distributed software development and argues that inconsistent failure reporting in communication mechanisms represents a significant obstacle to reliability.

1 Introduction

The premise of this paper is that the communication and programming technologies offered in support of "reliable" distributed software development should treat consistency as an important aspect of reliability, particularly in the way that failures are reported. At present, this is not often the case: the most common distributed computing environments report failures in ways that violate even the simplest notions of distributed consistency. We will argue that inconsistent reporting of failures is at the root of a number of forms of inconsistent distributed behavior in contemporary distributed systems and poses a major obstacle to the creation of robust distributed software.

The paper is structured into three major parts. Section 2 makes the notion of distributed consistency more precise, for the case of failure reporting. Section 3 reviews existing distributed computing technologies, showing how they can be categorized according to the degree of consistency provided. Section 4 discusses the technical issues raised by consistent failure reporting. The paper concludes by suggesting that by requiring this property on a routine basis, reliable application development might be greatly simplified.

*The authors are in the Department of Computer Science, Cornell University, and were supported under DARPA/ONR grant N00014-92-J-1866, and by grants from IBM, HP, Siemens, and GTE.

2 Consistency and failure reporting

We concern ourselves with real-world distributed systems, consisting of processes running on a collection of machines and interacting by message passing. The failure model we will consider is one in which processes fail by crashing, and in which the communication system can fail by delaying messages for long periods of time. We do not consider message loss, duplication, or out-of-order delivery, because it is easy to overcome these types of failures using sequence numbers and acknowledgement mechanisms. We do not consider more serious failure modes (message corruption, processes that lie) because these create much more difficult recovery situations, requiring techniques that go beyond the scope of most "realistic" systems. On the other hand, we do consider *transient* failure cases, in which a machine (and the processes on it) are temporarily unresponsive to messages, but then recover; such situations are extemely common in modern networking environments.

It is surprisingly difficult to give a general definition of "distributed consistency" for an environment of this sort. In a separate paper, we will explore this problem using formal methods, but for the present purposes, we favor a less detailed approach focused on the specific question of failure reporting. Consider the following example where we associate with each process p in the system a set of processes, $alive(p)$, consisting of those process that p believes to be alive and functional. Suppose that a system consists of three processes, a, b and c, and $alive(a) = \{a, b, c\}$, $alive(b) = \{a, b, c\}$ and $alive(c) = \{b, c\}$. Perhaps, a and b are respectively the primary and backup members of a key-distribution service that has been replicated for fault-tolerance, and c is a client of that service which believes process a has failed. Assume further that the logic of the key distribution service is such that it requires that there be only one primary member at any time.

Is this system state consistent? The answer to such a question depends upon the events that ensue: if process a subsequently crashes, and process b amends its $alive$ set to exclude a, one could argue that the system state observed here was transitional, showing c slightly more advanced in its execution than a and b. Scenarios such as this arise easily in asynchronous communication systems.

On the other hand, suppose that this set of states represents a final, quiescent one. Clearly, the system would then be inconsistent: process c may now start to request keys from the backup service, which may fail to respond (believing the primary to be operational), or if it responds, may give out incorrect keys.

From this we see that when failures are reported in a distributed system, the system may go through a period of inconsistency before settling back into a consistent state. Our goal should be that such a consistent state be reached, and that the degree of inconsistency prior to reaching it be limited to situations explicable by asynchrony, but not that the membership of the system be continuously agreed upon by all processes. In fact, continuous agreement is clearly impossible unless the system is static.

In what follows we will say that a failure reporting mechanism is *accurate* if it only reports real failures. We will say that a mechanism is *live* if it reports real failures within finite time. And, we will say that the mechanism is *safe* if, given a time at which to examine the state of the system, all operational processes at that time agree upon the state, operational or failed, of all other processes, and faulty processes are prevented from communicating with operational ones. This last issue is important if a process resumes communication after a period of faulty behavior.

This definition of safety raises leaves open an important issue, namely *when* the safety property should hold. Accordingly, we will say that a system is *consistent* if it guarantees safety, at times determined by an *evaluation rule* (regardless of whether or not it is accurate or live). In a consistent system, a process that takes actions based on the observation of a failure can assume that all other processes with which it interacts will observe the failure too when the time specified by the evaluation rule is reached, unless they themselves fail first. Further, the evaluation rule should indicate the set of processes to which the rule should apply, in an unambiguous way.

The most appropriate type of evaluation rule to employ depends on the intended application environment, hence we will briefly illustrate the idea with two key examples, but will not pursue this issue at any great length in the present paper. (A future, more theoretical, paper focuses on this question).

Stabilization consistency. The term *self-stabilization* is sometimes used to describe the class of concurrent systems capable of restoring a safety property after some event perturbs it. A self-stabilization evaluation rule would require only that if the system is quiet for a sufficiently long period of time, it will enter a state in which safety holds.

Piece-wise consistency. A self-stabilization rule would allow us to say that a system subject to brief periods of chaos that then settle into a safe state is consistent. For example, processes might observe failures in different orders, and some process might even observe another process to fail and then rejoin the system, while a different observer sees nothing at all. It is not at all clear that one would want to allow this sort of behavior.

We will say that a system is *piece-wise* consistent if, for any execution, it is possible to "slice" the execution into segments such that within any segment, the safety property holds for all possible simultaneous states.[1] By simultaneous states, we mean states which the processes in the system could have entered simultaneously, taking into account uncertainty in scheduling, relative execution speeds, and message delays. This concept can be formalized and corresponds to the notion of a *consistent cut*, introduced by Chandy and Lamport [9].

The idea in a piece-wise definition of consistency is that the system should suddenly – instantaneously – make transitions in which all processes switch from one membership list to a different one. In between these transitions, all processes have identical views of which processes are operational and which

[1] One might ask how this can cope with network partitions, in which two groups of machines become isolated from one another, with each considering all members of the other group faulty. In our work, as described later, we require that there be at most one *primary* partition in the system, and define the system to be consistent if the execution of the primary partition is consistent. Melliar-Smith *et. al.* discuss a different approach to the same problem in [1].

are faulty. The effect is that all operational processes see the same status transitions (operational to failed, and back) in the same order. The transition times correspond to the points at which one would slice up the execution. Notice that instead of meaning "instantaneous in real-time", this definition requires only that the transition be concurrent – there could be some small skew involved, but there is no exchange of messages during the transition. This is important because no real distributed system could guarantee that an event, such as the declaration that some process is faulty, would be registered at the same instant in real-time by all processes.

The advantage of a piece-wise consistent system is that if a process observes an event (a failure or recovery), it can send messages to other processes knowing that when they receive the message, they will already have observed the failure or recovery event. Moreover, if two processes both run an algorithm that uses failure/recovery events as an input, the algorithm will make the same state transitions in the same order, allowing them to cooperate implicitly, without needing to run agreement protocols to synchronize their views of the system and the actions they will take. Clearly, this simplifies the programmer's task. Several systems that we describe later in this paper are piece-wise consistent.

One can imagine other evaluation rules corresponding to other consistency properties. However, this paper is intended to be practical in focus, and these two types of consistency are the ones seen in the examples we will discuss in the most detail (Mach uses a form of stabilization consistency; Isis and Transis use piece-wise consistency). For the same reason, since these definitions would require a substantial amount of formalism to state rigorously, we will limit ourselves to an informal definition in the interest of brevity.

To summarize, we will say that a system is consistent if there is an evaluation rule under which safety holds at the times when the rule indicates that one can evaluate the safety property. Although our own work uses piece-wise consistency (in a primary partition) as its failure reporting model, the remainder of this paper will focus primarily on the distinction between systems for which some type of consistency evaluation rule can be defined and shown to hold, and those for which consistency simply is not achieved under any evaluation rule.

3 Consistency in reliable communication primitives

At the time of this writing, a number of types of distributed programming environments exist, each defining some form of reliable communication abstraction. This section starts by considering the consistency properties of *remote procedure call*, or RPC, and of communication *streams*, as implemented by TCP, TP4, or X.25 – well known standards for point-to-point data transmission. We then consider a number of higher-level distributed programming environments, and observe that the consistency properties of RPC and streams are frequently carried into the reliability attributes of more elaborate distributed programming environments.

3.1 Failure detection and consistency issues in RPC environments

Remote procedure call was introduced by Birrell and Nelson in [7] as a simple, familiar abstraction for client-server communication. As an abstraction it is elegant in the simplicity that it provides the programmer. Unfortunately, most implementations of RPC complicate the model by introducing poor failure modes and consistency problems. These implementations typically provide at-most-once semantics, requiring programmers to design their applications around these weak semantics. As an example, consider a simple service that maintains account balances. A typical remote procedure might look like

```
result = debit(account, value);
```

With a normal procedure call, the result might indicate success or failure (such as insufficient funds). In failure instances, the programmer knows that the requested action (debiting the account) was not completed. Common implementations of RPC however, complicate the code by introducing additional failure semantics. Such semantics allow the call to fail even though the server may have debited the account correctly (e.g. the reply may be repeatedly lost). All the programmer can assert upon failure is that the requested action was executed at most once, and even this can require substantial mechanism to implement. For example, synchronized clocks may be needed so that "old" requests can be recognized and discarded, storage may be needed in which to save copies of results until they have been acknowledged, and applications must be designed to accept and handle retransmitted requests, acks, and nacks long after an RPC interaction is terminated. Furthermore, the programmer cannot assume that the service itself has failed, the program may simply have observed a transient communication problem. So, while the RPC abstraction is appropriate for many applications, common implementations of RPC nonetheless force the programmer to address the inconsistencies that failures introduce.

Failure detection in RPC systems is based on timeouts and retries, and is typically reported on a per-call basis. This form of failure detection and reporting provides no guarantees about consistent observations of failure. It does, however, guarantee that if a real failure occurs and the process remains down for long enough, then all processes communicating with the process via RPC, will eventually see RPC failures. These errors are indistinguishable from transient communication failures. We can classify this type of failure detection as one that is inaccurate, providing liveness but not safety. That is, real failures will be reported in a timely fashion, but the same system will also report as failures transient communication problems which are not indicative of the health of any process.

Recalling our definitions, an RPC system is not safe because when it reports failures, it may do so in an arbitrary way that differs from observer to observer. Since such reporting has no particular reason to eventually stabilize on an mutually observed system state, there is no reason to believe that consistency will be achieved at all.

3.2 Failure detection and consistency issues in streams

Connection oriented communication provides what is typically considered a reliable communication mechanism. As in RPC, implementations of connection oriented protocols (such as TCP/IP, DECNet, X.25, ...) have semantics that make them surprisingly *unreliable* when failure consistency is considered. This is often not so much the fault of the commercial vendors, but of the standards themselves: the behavior of these protocols is mandated by the standards. To make this clear consider an application that manages TCP connections between objects in two processes a and b. Now suppose that one of these channels breaks due to transient communication problems. Upon such failures, the programmer would like to take some recovery action, perhaps by connecting to a backup process. Notice, however, that the failure of a connection does not imply the failure of either connected process. For example, a and b could see connection 1 break while connection 2 remains healthy (such scenarios are surprisingly common, and are normally triggered by transient communication problems). Thus, even *within a single program*, code associated with one of these connections could execute a recovery action, while code for the other connection continues to treat it as live – an outcome that could obviously provoke bugs!

Here, we see that although TCP does provide some limited failure consistency, in the sense that if one side sees the connection break, the other eventually will too, it is only consistent with respect to the channel in question, and says nothing about the "health" of either connected process. The programmer can only consider such behavior to be an *unreliable* indication of a process crash.

The technique that connection based protocols use to determine when to break a connection is similar to that of the RPC protocols. A succession of retransmissions of outgoing messages is sent until finally a timer expires and the connection is broken. TCP can keep channels open long after the death of an endpoint if there is no outstanding data and the "KEEPALIVE" mechanism is not being used. Thus, an application that needs timely notification of channel problems must use the KEEPALIVE mechanism (an optional part of the specification) or use the channel continuously. It follows that, like RPC, these common stream based protocols are inaccurate, providing liveness but not safety with respect to failure detection.

With regard to our detailed description of consistency, the comments made about RPC can also be made about streams. In a streams protocol, the observation of a failure does not imply that other operational processes connected to the same endpoint will eventually observe the same event. For this reason, regardless of the consistency evaluation rule one favors, common stream-based failure reporting schemes will not achieve safety and hence are not consistent.

3.3 Higher level systems

One might consider these protocols to be fairly low-level and expect that the higher layers of a system make up for their deficiencies. Interestingly, this is not often the case. We now consider a few such systems:

UNIX, Windows NT, Chorus, DCE, and CORBA. These systems, a mix of old and new, are representative of the communications facilities in a large majority of common systems.

Common implementations of UNIX stop at the TCP level of communications, that is the operating system does not supply stronger communication primitives. Some implementations use the communication facilities to provide common library calls such as *gethostbyname*. Often, these calls result in a remote lookup in a common database. As a result, the call itself becomes subject to the failure semantics of the embedded communication subsystem. This forces the programmer to deal with the weak failure semantics of the underlying communication system even for pieces of the application that may not be directly related to the communication needs of the application itself. This is not uniformly the case: for example, the STREAMS communication abstraction provided by SYSTEM V UNIX allows for higher level communications protocols. In fact, in [20] the authors describe an implementation of a higher level protocol with failure consistency within a STREAMS module. Unfortunately, the authors' concluded that the module system was poorly suited for such protocols. Similarly, Windows NT from Microsoft offers a socket interface that mimics the failure handling of Berkeley UNIX.

Prominent among emerging architectures for distributed computing are DCE (Distributed Computing Environment) and CORBA (Common Object Request Broker Architecure) from the OMG (Object Management Group). Both rely strongly on the semantics of typical RPC. While RPC is a large component of DCE, it is not mentioned at all in CORBA. Rather, CORBA specifies the semantics of method invocation on objects, which match those of common RPC implementations. One could certainly build a CORBA compliant system with stronger failure semantics, but applications wishing to interoperate with other CORBA systems (a primary goal of CORBA) would not be able take advantage of this for improved reliability.

The Chorus microkernel provides "reliable" IPC only for its version of RPC. Asynchronous Chorus IPC is unreliable (that is no non-local errors are reported). Chorus RPC has the same at-most-once failure semantics as other common implementations of RPC. Similarly, timeout is the primary means of failure detection, and consistency is provided on a per call basis. As in the above, Chorus RPC can be classified as inaccurate, providing liveness but not safety with respect to failure detection.

4 Stronger consistency models

The situation seen in the previous section, although common, is not universal. This section considers two modern distributed systems, the Mach operating system and the Isis toolkit. We then point to a number of other distributed systems with consistency models similar to Mach and/or Isis.

4.1 Failure detection and consistency issues in Mach

The Mach microkernel from Carnegie Mellon University offers a strong notion of failure consistency. Mach IPC employs a port death notification mechanism that enables the application to reliably determine if the holder of a port receive-right has failed; in addition it has a notifications for events such as "no more senders" (indicating that outstanding send-rights to a port have been destroyed). Mach also provides strong guarantees for message delivery. To implement this behavior, Mach detects the failure of a machine (and hence all of its tasks) through one of two methods. In the first method the applications learn of port deaths on a machine only after the machine *recovers*. While the machine remains incommunicado, applications trying to communicate with it simply wait. The second method requires that an operator intervene, issuing a command that declares the machine as having crashed, spawning port death and other notifications. Mach thus provides an accurate mechanism for determining the health of a process, but does so at the risk of delaying the death notification indefinitely. This form of failure detection is accurate and safe, but not live.

Mach achieves stabilization consistency, in the sense of Section 2: there is no guarantee that two processes will view the same events in the same order, but if the system remains quiet for sufficiently long, all processes will eventually have consistent views of the overall composition of the network.

4.2 Failure detection and consistency issues in Isis

The Isis system developed at Cornell University also provides a strong notion of failure consistency but uses a different approach. Isis makes use of a failure detector service built using a group membership protocol (GMP) as described in [16]. This service is replicated and fault tolerant and determines when applications can be notified that other applications may be considered as having failed. This system guarantees that if one process observes failure through the Isis communication subsystem, then all processes in the same partition will also observe the failure. Of course, Isis can't perform the impossible and it is impossible to guarantee consensus in an asynchronous system where even one crash failure can occur[12]. In particular, the failure detection service uses timeouts and hints from the operating system and other processes to determine when a machine has crashed. Because transient communication disruptions and other temporary problems (like an unresponsive file server) can mimic failures, the failure detector can be fooled and declare a healthy process dead. The approach however, is such that all processes would consider such a process dead, because they listen to the failure detector and sever connections as appropriate.

To resume communication with healthy applications, a process declared as faulty in this way would need to explicitly rejoin the system. In this manner, "zombie" processes are prevented from somehow interfering with the operational part of the system.

In the case of a network partition, one of two outcomes can arise, depending on how the failure detector service is configured. In the default configuration, those processes in a partition with a minority of processes will be severed from the system; they will be unable to reconnect until the partition is eliminated. The worst case for this scenario occurs when only minority partitions exist: all processes will become "disconnected". Because this will shut down the entire application, a second configuration is also possible, under which processes in a minority partition are notified that a partition may have occurred, but are then permitted to continue running in a state that is now known to be potentially disconnected from the majority of processes. Some applications can continue to operate in a partitioned environment, others cannot.

The approach achieves piece-wise consistency, providing both safety and liveness, but it gains liveness by sacrificing accuracy. That is, all processes see the same sequence of events, and each new event is reported along a consistent cut, but in some situations an otherwise healthy process may be declared faulty.

4.3 Systems with similar properties

At this point we have classified failure detection semantics with respect to accuracy of failure detections, safety and liveness. Notice that the vast majority of systems provide liveness guarantees over accuracy and safety. It is worth looking at some other novel systems that provide safety guarantees in their failure reporting semantics. We briefly examine VAXclusters, Amoeba, Tandem NonStop, Ladin's Lazy Replication, and Transis.

Digital Equipment Corporation's VAXcluster system uses a *connection manager* to provide a mechanism for failure detection and recovery of nodes in a cluster [13]. The connection manager is a membership service that runs on all of the nodes in a cluster and manages virtual circuits between each member. A quorum voting scheme arbitrates the membership during partions of the cluster; weights can be assigned to the nodes to bias the system in favor of including certain nodes in any majority partition. During transient communication failures it is possible that the membership service will deem a node as having failed, when in fact it may have simply been overloaded. In this case, upon reestablishing its connection with the cluster, the node is told to reboot.[2] In this manner consistency with the membership service's prior decision is maintained. Like Isis, VAXclusters failure detection is safe and live but not accurate.

The Amoeba operating system uses a closely related approach to achieve failure consistency for data replication within a segment (a portion of the network not subject to partitions) [19]. In this system a boot service detects *container* crashes (a container is simply a machine with some stable storage) through low-level polling. If the container does not respond to the polling, the boot service will attempt to reboot the machine. If after this, the machine still does not respond to polling, then the boot service will ensure that it remains unavailable by disconnecting it from the network, turning it off via an electronic switch, or by using some other means at its disposal. The Ameoba system uses special hardware that is suited for this purpose. The boot service thus provides failure consistency by forcing observed failures to correspond to real failures.

[2]The VAXcluster system makes use of the low-level VMS Systems Communication Architecture (SCA) to provide these services. Unfortunately, this architecture is not directly compatible with the OSI standard, or the Internet protocol suite.

The ability of the forced reboot approach to tolerate complex network partitioning failures represents a possible area for future theoretical study. If a network might experience arbitrary failures and recoveries including partition, the boot-service itself will need to be implemented as a replicated distributed server. This suggests that to solve the consistency problem at the application level, we may need to solve it within the boot server first. Because such a system is asynchronous, results such as FLP limit the extent to which this can be done.

Tandem Computers, Inc. uses a process pair scheme in their NonStop ™ Operating system. This fault tolerance method transparently handles the recovery of a single hardware or software component failure [4]. The system is based on redundant hardware and a messaging infrastructure that is robust to single failures, and is responsible for declaring a path dead. This decision is sender-based and is made upon the repeated inability to receive acknowledgements. In the Tandem system each message is individually acknowledged and is received in the same order as it was sent. While a Tandem NonStop System can consist of a large number of processors, the fault tolerance of applications in based on a primary-backup approach that involves only two-processors at a time. The occurrence of multiple observered failures (whether real or not), can lead to an inconsistent system. The scheme is thus safe and live up to a single failure, but not safe for multiple failures. Accuracy of the solution depends on the correct function of low-level polling mechanisms and hardware.

The *Lazy Replication* scheme of [14] uses a 3-phase group (view) membership protocol to react to the observed failure of the primary for the global and server ordered updates in their system. This system is not used to provide reliable communication with individual machines and does not present failures to the application, rather the mechanism is used to reliably communicate with fault-tolerant services (abstract data types), the goal being to provide the service of the data type despite failures of individual replicas. Like Isis, this system provides safe and live fault-tolerant services, but may fail to utilize servers that the membership protocol has deemed as dead. Accuracy appears not to be an issue in this approach. When a faulty node recovers, it is brought up to date by a replay of logged update records. A similar approach is used in Peterson's *Psync* system [15].

Transis [2, 3] uses an inaccurate, safe, live membership protocol to provide consistent views of *configuration sets* within *broadcast domains*. This system guarantees that failure events are observed consistently within each configuration set. Like Isis, Transis provides a communication service which is responsible for providing reliability, atomicity, and message delivery ordering. In the event of network partitions, or disconnections, a given configuration set can split into two or more pieces: the consistency properties of the system are defined on a per-partition basis. Transis retains consistent membership views within any single partition, and provides for the merging of partitions upon reestablishment of normal communications.

This review is not intended to be complete, but rather to illustrate our contention that the issues of consistency and liveness in failure reporting are important ones that have been treated through a suprisingly varied set of mechanisms.

5 Implementing consistent failure-reporting

In this section we summarize the major approachs to implementing consistent failure reporting used in current systems. The exact degree of consistency will depend on details of the implementation not considered here, and so we will not comment on this. However, we do note that all of these approaches can be distinguished from failure reporting schemes that make no attempt to be consistent at all.

Wait for restart. This is the approach used in the Mach system; it achieves consistency and accuracy in failure reporting by waiting for the failed machine to recover and tell the world that it died (or by trusting an operator to declare the machine dead). This approach sacrifices liveness, which can represent a considerable drawback to applications that require timely response from a service, or that need to operate themselves automatically under conditions in which no operator can be counted upon. On the other hand it is extremely simple to implement.

Wait for restart is accurate and safe, but not live.

Forced reboot. This is the approach used in VAXclusters and Ameoba: if a machine is declared faulty, a mechanism is used to force it to reboot, thus preventing it from interacting with machines that have taken actions that only make sense under the assumption that the machine in question has failed.

Forced reboot is inaccurate (although self-fulfilling!), but is safe and live. However, it may require special hardware support.

Dynamic consensus - GMP. The third option is to use the sort of membership agreement mechanism seen in systems like Isis, Transis and IBM's HAS. (We believe that the first use of a membership mechanism was in a reliable broadcast protocol by Chang and Maxemchuck, for broadcast networks [10]). A *group membership protocol* is used to maintain agreement on the system membership, and is the sole trusted agent – an oracle – within the system insofar as failure and recovery decisions are concerned. The consistency and liveness properties of the GMP protocol will determine the consistency and liveness characteristics of the applications built over such a layer. Often, in this approach, the application is said to "assume a failstop execution model", referring to a model explored by Schlicting and Schneider [17] in which processes fail by crashing in a detectable manner.

Dynamic consensus protocols can be designed to be accurate, in which case it they are safe but not live, or to sacrifice accuracy in favor of a solution that is safe and live, but in which transient problems can trigger inaccurate failure notifications.

6 When is failure-reporting consistency important?

The first parts of this paper have argued that reliable communication systems can be broadly classified into two categories, depending upon the degree to which failures are consistently reported to users of the system, with a secondary issue involving a tradeoff between liveness and accuracy. The question this leaves open concerns how to recognize applications that need consistency.

This section considers some examples of systems with non-trivial distributed properties, in which liveness of the system as a whole is linked to taking one action if some message is received, and a different action if failure occurs instead. We argue that in most cases, systems having this structure either require consistent failure reporting, or require some sort of application-level agreement protocol by which the application programs can overcome inconsistencies originating in the lower level communication mechanisms.

For example, the Mach IPC model, described earlier, requires that an application program holding a port be *notified* when the last holders of send-rights to that port terminate. Mach also requires that IPC be reliable unless one endpoint of a communication interaction fails, and in particular that send-rights become dead if and only if the holder of the corresponding receive-right has failed, or the receive-right has been destroyed. Further, Mach needs to provide these properties despite dynamicism, such as the migration of receive-rights from node to node. For example, suppose that process a holds a send-right on a port currently owned by process b. Process b transfers the right to process c and, shortly after, the machine on which b is running crashes. Process a now attempts to transmit to the port. Mach IPC is explicit in guaranteeing that a will succeed in sending its message (to c), even though the machine on which a last knew the right to reside is no longer operational. If a is instead informed that it holds a dead right, application bugs might be triggered.

Mach IPC thus offers a distributed reliability guarantee that requires consistent handling of failures and failure notifications. One can generalize the Mach requirement to a form of system-wide invariant, namely that the "reference counts" associated with receive-rights correspond to the number of send-rights on the port, subject of course to some delay in cases where the holder of a send-right has failed but this has not yet been detected.

The idea of maintaining accurate reference counts for distributed objects is also discussed in [18]. In fact, the Mach problem is a specific instance of a more general distributed garbage collection problem seen in many distributed object oriented systems. In this generalization, one wishes to garbage collect any passive (data) objects for which no references remain. Object references can be passed around in the system, so that detecting this condition is not necessarily trivial, an issue further complicated by failures. Inconsistent observation of failures could result in lost references and trigger an inappropriate round of garbage collection. [18] discusses the need for an accurate *oracle*, capable of detecting and reporting failures. The relationship between such an oracle and the GMP approach should be clear: given a communication layer that reports failures in a consistent manner, protocols to maintain reference counts on ports or objects, and to transfer references from one machine to another, can be designed and implemented in a fairly straightforward manner. Lacking consistency, the development of such a solution is much harder.

A similar pattern is seen in many of the tools within the Isis toolkit. For example, Isis has a generalized primary-backup tool called the coordinator-cohort tool. In a normal primary-backup system, one builds pairs of processes; the primary handles all requests and the backup sees some form of trace of incoming requests and replies to them, which it can use to reconstruct the state of the primary after a crash [8]. The Isis tool operates in a similar way, but within groups of processes. For each incoming request, a process is designated to be the "coordinator" (primary) for the handling of that request. Unlike for a normal primary-backup scheme, the role of being primary can be load-shared within the group, so that concurrent requests can receive different coordinators, keeping the whole group busy. For a given request, the non-coordinator processes are ranked and called "cohorts"; they will take over as coordinator in rank order if the coordinator crashes before completing the execution of the request (it signifies completion by replying to the caller that issued the request).

The coordinator-cohort scheme relies upon the members of the process group having consistent information about the group. They need to know that the incoming message was received by all members. Otherwise, a cohort might monitor a coordinator indefinitely, and yet the coordinator may not have received the corresponding request. They need to have the same rankings of group members. Otherwise, a request might end up with multiple coordinators, or no coordinator. And, they require that cohorts be notified of failures in a timely and uniform manner. Otherwise, again, a request might end up with multiple coordinators, or none. As we reported in [6], a tool having these properties can be layered over a communication layer with multicast protocols that report failures in a consistent manner. Inconsistent failure reporting makes the development of such a tool problematic.

Other distributed computing problems that require consistent failure reporting include:

- Group communication and reliable multicast. If failure reporting is inaccurate, an operational process may never be sent a copy of a message that other processes received.
- Synchronization. If a process fails, one may need to "break" locks it held, so that (after cleanup) other processes can make progress. If failures are reported inconsistently, locks may be broken inappropriately.
- Replicated data. Updates to replicated data need to reach all processes holding copies. Inconsistent failure reporting could leave some processes holding stale replicas.
- Parallel database search. If the search of a database is subdivided among a set of processes, inconsistent failure reporting could result in parts of the database not being searched, or some parts being searched multiple times.

These sorts of problems, which are typical of problems for which Isis and Mach are often used, would be difficult or impossible to solve in settings that report failures inconsistently. We believe that inconsistency of failure reporting stands out as a major obstacle to building complex distributed software on platforms such as UNIX, even under next-generation programming environments such as CORBA.

Although this paper will not attempt to address the question of which type of consistency these problems need, we do note that a wider variety of problems has been solved using the "piece-wise" consistency model than with a simple stabilization model. In particular, while all of the problems cited above correspond to "tools" available in the Isis system, we are not aware of any comparable tools built using a stabilization consistency model. As noted in Section 2, detailed treatment of this issue would require more formalism than we felt would be appropriate here.

6.1 Transactional consistency

Until now, we have avoided discussion of the transactional approach to distributed computing and fault-tolerance, which employs a very different approach to reliability and consistency. In these types of systems, consistency is defined through reference to a transactional serializability and atomicity model, which in turn reflects the manner in which typical database applications are structured. A good review of this work can be found in [5].

Although transactional consistency is extremely important, we focused on non-transactional applications intentionally. There is a large body of existing, non-transactional code and systems, and the majority of new distributed applications and services use non-transactional execution models. These systems need to be reliable too, justifying the sort of analysis and argument presented above.

It is interesting to note that consistency of failure reporting does find echoes in transactional settings. For example, in [5] there is a discussion of failure-serialization graphs, basically showing that unless transactions have consistent perceptions of the status, operational or failed, of replicas of objects, transactions on replicated data may not be correctly serializable. Thus, although our consistency model is simpler, the approach we favor is not particularly radical.

6.2 Real-time membership mechanisms

A second topic we have not addressed here concerns protocols for maintaining membership information in which the focus is on liveness but not safety. For example, Cristan proposed a real-time system membership service in [11], which manages information about membership in partitions, merging partitions when the opportunity arises. Relating the type of consistency seen in this class of systems to the type seen in our work represents an interesting open problem.

7 Conclusions

We have seen that consistency in failure reporting can be a useful, even necessary, property in reliable distributed systems. Obtaining such consistency is neither particularly costly nor overwhelmingly difficult, and there are many examples of systems that report failures consistently.

Unfortunately, modern computing systems treat failure reporting as an application-specific problem, forcing users to live in a world of inconsistent failure reporting. This might be characterized as an application of the "end-to-end" philosophy to the failure reporting problem. (Under the end-to-end approach, low levels of a communication system provide minimal guarantees, making applications responsible for introducing mechanisms to recover from failures). We question the appropriateness of this philosophy in the case of failure reporting. Since the solutions either involve some form of consensus on failure, or use some type of trusted system-wide service, it is not clear how an application developer could introduce consistency in a setting where the operating system does not already provide for it.

Standards bodies have also overlooked this issue: no communications standard today requires consistency in failure reporting. Indeed, no standard even provides for the addition of consistency-preserving mechanisms.

Inconsistent failure reporting is one of the major barriers to progress in developing highly reliable, self-managed, distributed software systems and applications. It is ironic that the very trend towards standardization that has made it practical to entertain building such systems may be rendering it nearly impossible to do so! The technology for solving the problem is at hand, and if only its importance were more widely appreciated, we believe that a major barrier to distributed application development could be rapidly eliminated.

Acknowledgements

This paper emerged from discussions with Tushar Chandra, Aleta Ricciardi, Andre Schiper, and Robbert van Renesse. As noted earlier, work is underway on a formal treatment of this general topic; Andre Schiper, Aleta Ricciardi and Ken Birman are active on this. We are also grateful to Mike Reiter for his comments on a preliminary draft.

References

[1] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ri ng. In *International Conference on Distributed Computing Systems*, number 13th, May 1993.

[2] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms in broadcast domains. Technical Report CS92-10, The Hebrew University of Jerusalem, June 1992.

[3] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, Massachusetts, July 1992. institution of Electrical and Electronic Engineers.

[4] Joel F. Bartlett. A NonStop kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages XX–YY, Pacific Grove, California, December 1981. ACM SIGOPS.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, Texas, November 1987. ACM SIGOPS.

[7] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *Transactions on Computer Systems*, 2(1):39–59, February 1984.

[8] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary–backup approach. In Sape Mullender, editor, *Distributed Systems*. ACM Press, 1993.

[9] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *Transactions on Computer Systems*, 3(1):63–75, February 1985.

[10] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *Transactions on Computer Systems*, 2(3):251–273, August 1984.

[11] F. Cristian. Reaching Agreement on Processor Group Memberhsip in Synchronous Distributed Systems. Technical Report RJ 5964, IBM Almaden Research Center, August 1990. Revised from March, 1988.

[12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[13] Nancy P. Kronenberg, Henry M. Levy, William D. Strecker, and Richard J. Merewood. The vaxcluster concept: An overview of a distributed system. *Digital Technical Journal*, (5):7–21, September 1987.

[14] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploting the semantics of distributed services. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 43–58, Qeubec City, Quebec, August 1990. ACM SIGOPS-SIGACT.

[15] Larry L. Peterson, Nick C. Bucholz, and Richard Schlichting. Preserving and using context information in interprocess communication. *Transactions on Computer Systems*, 7(3):217–246, August 1989.

[16] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Procedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*. ACM, August 19-21 1991.

[17] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *Transactions on Computer Systems*, 1(3):222–238, August 1983.

[18] Marc Shapiro, Peter Dickman, and David Plainfosse. Ssp chains: Robust, distributed references supporting acyclic garbage collection. Technical Report 1799, Institut National de la Recherche en Informatique et Automatique, November 1992.

[19] Robbert van Renesse and Andrew S. Tanenbaum. Voting with Ghosts. In *Proc. of the 8th International Conf. on Distr. Computing Systems*, pages 456–461, June 13-17 1988.

[20] Werner Vogels, Luis Rodrigues, and Paulo Veríssimo. Fast group communication for standard workstations. In *Proceedings of the OpenForum '92 Technical Conference*, pages 337–363, November 1992.