Understanding Fault-Tolerant Distributed Systems

Flaviu Cristian
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
flaviu@cs.ucsd.edu

May 25, 1993

Abstract

We propose a small number of basic concepts that can be used to explain the architecture of fault-tolerant distributed systems and we discuss a list of architectural issues that we find useful to consider when designing or examining such systems. For each issue we present known solutions and design alternatives, we discuss their relative merits and we give examples of systems which adopt one approach or the other. The aim is to introduce some order in the complex discipline of designing and understanding fault-tolerant distributed systems.

1 Introduction

Computing systems consist of a multitude of hardware and software components that are bound to fail eventually. In many systems, such component failures can lead to unanticipated, potentially disruptive failure behavior and to service unavailability. Some systems are designed to be fault-tolerant: they either exhibit a well-defined failure behavior when components fail or mask component failures to users, that is, continue to provide their specified standard service despite the occurrence of component failures. To many users temporary errant system failure behavior or service unavailability is acceptable. There is, however, a growing number of user communities for whom the cost of unpredictable, potentially hazardous failures or system service unavailability can be very significant. Examples include the on-line transaction processing, process control, and computer-based communications user communities. To minimize losses due to unpredictable failure behavior or service unavailability, these users rely on fault-tolerant systems. With the ever increasing dependence placed on computing services, the number of users who will demand fault-tolerance is likely to increase.

The task of designing and understanding fault-tolerant distributed system architectures is notoriously difficult: one has to stay in control of not only the standard system activities when all components are well, but also of the complex situations which can occur when some components fail. The difficulty of this task can be exacerbated by the lack of clear structuring concepts and the use of a confusing terminology. Presently, it is quite common to see different people use different names for the same concept or use the same term for different concepts. For example, what one person calls a failure, a second person calls a fault, and a third person might call an error. Even the term "fault-tolerant" itself is used ambiguously to designate such distinct system properties as "the system has a well-defined failure behavior" and "the system masks component failures".

This paper attempts to introduce some discipline and order in understanding fault-tolerance issues in distributed system architectures. In Section 2 we propose a small number of basic architectural concepts. In Sections 3 and 4 we use these concepts to formulate a list of key hardware and software issues that arise when designing or examining the architecture of fault-tolerant distributed systems. Since the search for satisfactory answers to most of these issues is a matter of current research and experimentation, we examine various proposals, discuss their relative merits, and illustrate their use in existing commercial fault-tolerant systems. Besides being useful as a design guide, our list of issues also provides a basis for classifying existing and future fault-tolerant system architectures. Finally, Section 5 comments on the adequacy of the proposed concepts.

2 Basic architectural concepts

To achieve fault-tolerance, a distributed system architecture incorporates redundant processing components. Thus, before we discuss the issues which underlie fault-tolerance - or redundancy management - in such systems, we need to introduce their basic architectural building blocks and classify the failures that these basic blocks can experience.

2.1 Services, servers, and the "depends" relation

The three notions that, we feel, provide the best means to explain computing system architectures are the concepts of service, server, and "depends upon" relation among servers.

A computing *service* specifies a collection of operations whose execution can be triggered by inputs from service users or the passage of time. Operation executions may result in outputs to users and in service state changes. For example, an IBM4381 raw processor service consists of all the operations defined in a 4381 processor manual, and a DB2 database service consists of all the relational query and update operations that clients can make on a database.

The operations defined by a service specification can be performed only by a *server* for that service. A server implements a service without exposing to users the internal service state representation and operation implementation details. Such details are hidden from users, who need know only the externally specified service behavior. Servers can be hardware or software implemented. For example, a 4381 raw processor service is typically implemented by a hardware server; however, sometimes one can see this service 'emulated' by software. A DB2 service is typically implemented by software, although it is conceivable to implement this service by a hardware database machine.

Servers implement their service by using other services which are implemented by other servers. A server u *depends* on a server r if the correctness of u's behavior depends on the correctness of r's behavior. The server u is called a *user* (or client) of r, while r is called a *resource* for u. Resources in turn might depend on other resources to provide their service, and so on, down to the atomic resources of a system, which one does not wish to analyze any further. Thus, the user/client and resource/server names are relative to the "depends" relation: what is a resource or server at a certain level of abstraction can be a client or a user at another level of abstraction. This relation is often represented as an acyclic graph in which nodes denote servers and arrows represent the "depends" relation. Since it is customary to represent graphically a user u of a resource r *above* r, u is said to be at a level of abstraction "higher" than r [D71], [P79], [R75].

For example, a file server f, which uses the services provided by a disk space allocation server s and a disk I/O server d to provide file creation, access, update, and deletion service, depends on s and d (see figure 1). To implement the file service, the designer of f assumes that the allocation and I/O services provided by s and d have certain properties. If the specifications of s and d imply these properties and f, s and d are correctly implemented, then f will behave correctly. All of the above servers depend on processor service provided to them by some underlying operating system when they execute (or are interpreted). If they were written in a high level language, they also depend on compilers and link-editors to be translated correctly in machine language. When all software servers under discussion depend on such translation and processor services, it is customary to omit representing this fact in the "depends" graph.

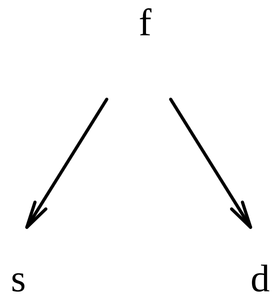

Figure 1: Depends-Upon relation

Note that the static "depends" relation defined above relates to the correctness of a service implementation and differs from the dynamic "call" (or flow control) and "interprets" (or executes) relations which can exist at run time between servers situated at different abstraction levels. For example, the file server f will typically use synchronous, blocking, "down-calls" to ask the allocation server s for free storage and will use asynchronous, non-blocking, down-calls to ask the I/O server d to initiate disk I/O in parallel. When the I/O is completed, the I/O server will typically notify the file server f by using an "up-call" [Cl85] which might interrupt f. If a processor p interprets the programs of f, s, and d these "depend" on p (although p "executes" them).

A distributed system consists of software servers which depend on processor and communication services. Processor service is typically provided concurrently to several software servers by a multi-user operating system such as Unix or MVS. These operating systems in turn depend on the raw processor service provided by physical processors, which in turn depend on lower level hardware resources such as CPUs, memories, I/O controllers, disks, displays, keyboards, and so on. The communication services are implemented by distributed communication servers which implement communication protocols such as TCP/IP and SNA by depending on lower level hardware networking services. It is customary to designate the union of processor and communication service operations provided to application servers as a *distributed operating system* service.

2.2 Failure classification

A server designed to provide a certain service is *correct* if, in response to inputs, it behaves in a manner consistent with the service specification. We assume the specification prescribes both the server's response for any initial server state and input and the real-time interval within which the response should occur. By a server's response we mean any outputs that it has to deliver to users as well as any state transition that it must undergo.

A server *failure* occurs when the server does not behave in the manner specified. An *omission* failure occurs when a server omits to respond to an input. A *timing* failure occurs when the server's response is functionally correct but untimely, that is, the response occurs outside the real-time interval specified. Timing failures thus can be either *early* timing failures or *late* timing failures (*performance* failures). A *response* failure occurs when the server responds incorrectly: either the value of its output is incorrect (*value* failure) or the state transition that takes place is incorrect (*state transition* failure). If, after a first omission to produce output, a server omits to produce output to all subsequent inputs until its restart, the server is said to suffer a *crash* failure. Depending on the server state at restart, one can distinguish between several kinds of crash failure behaviors. An amnesia-crash occurs when the server restarts in a predefined initial state that does not depend on the inputs seen before the crash. A partial-amnesia-crash occurs when, at restart, some part of the state is the same as before the crash while the rest of the state is reset to a predefined initial state. A pause-crash occurs when a server restarts in the state it had before the crash. A halting-crash occurs when a crashed server never restarts. Note that while crashes of state-less servers, pause-crashes and halting crash behaviors are subsets of omission failure behaviors, in general, partial and total amnesia crash behaviors are not a subset of omission failure behaviors. In what follows we follow accepted practice and use the term crash ambiguously to designate one or more of the above kinds of crash failure behaviors; which meaning is intended should be clear from the way the state and the restart operation of the server(s) under consideration are defined.

An operating system crash followed by re-boot in a predefined initial system state and a database server crash followed by recovery of a database state that reflects all transactions committed before the crash are examples of crash failures. A communication service that occasionally loses but does not delay messages is an example of a service that suffers omission failures. An excessive message transmission or message processing delay due to an overload affecting a set of communication servers is an example of a communication performance failure. When some action is taken by a processor too soon, perhaps because of a timer that runs too fast, we speak of an early timing failure. A search procedure that "finds" a key not inserted in a table and an alteration of a message by a communication link subject to random noise are examples of server response failures.

2.3 Server failure semantics

When programming recovery actions for a server failure, it is important to know *what* failure behaviors the server is likely to exhibit. The following example illustrates this point. Consider a client u which sends a service request sr through a communication link l to a server r. Let d be the maximum time needed by l to transport sr and p be the maximum time needed by r to receive, process, and reply to sr. If the designer of u knows that communication with r via l is affected only by omission -not performance- failures, then if no reply to sr is received by u within $2(d+p)$ time units, u will never receive a reply to sr. To handle this, u might resend a new service request sr' to r, but u will not have to maintain any local data that would allow it to distinguish between answers to "current" service requests, such as sr', and answers to "old" service requests, such as sr. If, on the other hand, the designer of u knows that l or r can suffer performance failures, if no reply to sr is received by u within $2(d+p)$ time units, u will have to maintain some local data, for example a sequence number, that will allow it to discard any "late" answer to sr.

Since the recovery actions invoked upon detection of a server failure depend on the likely failure behaviors of the server, in a fault-tolerant system one has to *extend* the standard specification of servers to include, in addition to their familiar failure-free semantics (the set of failure-free behaviors), their likely failure behaviors, or *failure semantics* [C85]. If the specification of a server s prescribes that the failure behaviors likely to be observed by s users should be in class F, we say that "s has F failure semantics" (a discussion of what we mean by "likely" is deferred to Section 2.6). We use the term failure "semantics" instead of failure "mode" because semantics is already a widely accepted term for characterizing behaviors in the absence of failures, and there is no logical reason why such dissimilar words as "semantics" and "mode" should be used to label the same notion: allowable server behaviors.

For example, if a communication service is allowed to lose messages, but the probability that it delays or corrupts messages is negligible, we say that it has omission failure semantics (we discuss what "negligible" means in Section 2.6). When the service is allowed to lose or delay messages, but it is unlikely that it corrupts messages, we say that it has omission/performance failure semantics. Similarly, if a processor is likely to suffer only crash failures or a memory is likely to suffer only omission failures in response to read requests (because of parity errors), we say that the processor and the memory have crash and read omission failure semantics, respectively. In general, if the failure specification of a server s allows s to exhibit behaviors in the union of two failure classes F and G, we say that s has F/G failure semantics. Since a server that has F/G failure semantics can experience more failure behaviors than a server with F failure semantics, we say that F/G is a weaker (or less restrictive) failure semantics than F. Equivalently, F is stronger (or more restrictive) than F/G. When any failure behavior is allowed for a server s, that is, the failure semantics specified for s is the weakest possible, we say that s has *arbitrary* failure semantics. Thus, the class of arbitrary failure behaviors includes all the failure classes defined previously.

It is the responsibility of a server designer to ensure that it properly implements a specified failure semantics. For example, to ensure that a local area network service has omission/performance failure semantics, it is standard practice to use error detecting codes that detect with high probability any message corruption. To ensure that a local area network has omission failure semantics, one typically uses network access mechanisms that guarantee bounded access delays and real-time executives that guarantee upper bounds on message transmission and processing delays [LL89]. To implement a raw hardware processor service with crash failure semantics, one can use duplication and matching, that is, use two physically independent processors that execute in parallel the same sequence of instructions and that compare their results after each instruction execution, so that a crash occurs when a disagreement between processor outputs is detected [TW89].

In general, the stronger a specified failure semantics is, the more expensive and complex it is to build a server that implements it.

The following examples illustrate this general rule of fault-tolerant computing. A processor that achieves crash failure semantics by using duplication and matching, as discussed in [TW89], is more expensive to build than an elementary processor which does not use any form of redundancy to prevent users from seeing arbitrary failure behaviors. A storage system that guarantees that an update is either completely performed or is not performed at all when a failure occurs is more expensive to build than a storage system which can restart in an inconsistent state because it allows updates to be partially completed when failures occur. More design effort is required to build a real-time operating system that provides to software servers processor service with crash failure semantics, than to build a standard multi-user operating system, such as Unix, which provides processor service with only crash/performance failure semantics [LL89].

2.4 Hierarchical failure propagation and masking

A failure behavior can be classified only with respect to a certain server specification, at a certain level of abstraction. If a server depends on lower level servers to provide correctly its service, then a failure of a certain type at a lower level of abstraction can result in a failure of a different type at the higher level of abstraction. For example, consider a value failure at the physical transmission layer of a network which causes two bits of a message to be corrupted. If the data link layer above the physical layer uses at least 2-bit error detecting codes to detect message corruption and discards corrupted messages, then this failure is propagated as an omission failure at the data link layer. As another example, consider a clock affected by a crash failure that displays the same "time". If that clock is used by a higher level communication server that is specified to associate different timestamps with different messages it sends at different real times, then the communication server may be classed as experiencing an arbitrary failure [CASD85].

As illustrated above, failure propagation among servers situated at different abstraction levels of the "depends upon" hierarchy can be a complex phenomenon. In general, if a server u depends on a resource r with arbitrary failure semantics, then u will likely have arbitrary failure semantics, unless u has some means to check the correctness of the results provided by r. The task of checking the correctness of results provided by lower level servers is very cumbersome, so fault-tolerant system designers prefer to use whenever possible servers with failure semantics stronger than arbitrary, such as crash, omission or performance. In hierarchical systems relying on such servers, *exception handling* provides a convenient way to propagate information about failure detections across abstraction levels and to mask low level failures from higher level servers [C89]. The pattern is as follows. Let i and j be two levels of abstraction, so that a server u at j depends on the service implemented by the lower level i. If u down-calls a server r at i, then information about the failure of r propagates to u by means of an exceptional return from r (this can be a time-out event signalling no timely return from r). If the server u at j depends on up-calls from lower level servers at i to implement its service, u needs some knowledge about the timing of such up-call events to be able to detect lower level server failures. For example, if the server u expects an interrupt from a sensor server, every *per* milliseconds, a missing sensor data update can be detected by a time-out. If the server u at j can provide its service despite the failure of r at i we say that u *masks* r's failure. Examples of masking actions that u can perform are down-calls to other, redundant servers r', r'', ... at i, or repeated down-calls to r if r is likely to suffer transient omission failures. If u's masking attempts do not succeed, a consistent state must be recovered for u before information about u's failure is *propagated* to the next level of abstraction, where further masking attempts can take place. In this way, information about the failure of a lower level server r can either be hidden from the human users by a successful masking attempt or can be propagated to the human users as a failure of a higher level service they requested. The programming of masking and consistent state recovery actions in a client c of u is usually simpler when c's designer knows that u does not change its state when it cannot provide its standard service. Servers which, for any initial state and input, either provide their standard service or signal an exception without changing their state (termed "atomic with respect to exceptions" in [C89]) simplify fault-tolerant programming because they provide their users with a simple-to-understand omission failure semantics.

We illustrate the hierarchical failure masking pattern described above by an IBM MVS operating system example running on a processor with several CPUs (other examples of hierarchical masking can be found in [W75]). When an attempt at reading a CPU register results in a parity check exception detection, there is an automatic CPU retry from the last saved CPU state. If this masking attempt succeeds, data about the original failure is logged and the human operator is notified, but the original (transient) omission CPU failure occurrence is masked from the MVS operating system and the software servers above it. Otherwise, the observed parity exception followed by the unsuccessful CPU retry is reported by an interrupt as a crash failure of that CPU server to the MVS system, which in turn may attempt to mask the failure by re-executing the program which caused the CPU register parity exception (from a previously saved checkpoint) on an alternate CPU. If this masking attempt succeeds, the failure of the first CPU is masked from the higher levels of abstraction -the software servers which run application programs. If there are no alternate CPUs or all masking attempts initiated by the MVS system fail, a crash failure of the MVS system occurs.

2.5 Failure masking through server groups

To ensure that a service remains available to clients despite server failures, one can implement the service by a *group* of redundant, physically independent, servers, so that if some of these fail, the remaining ones provide the service. We say that a group *masks* the failure of a member m whenever the group (as a whole) responds as specified to users despite the failure of m. While *hierarchical masking* (discussed in the previous Section) requires users to implement any resource failure masking attempts as exception handling code, with *group masking*, individual member failures are entirely hidden from users by the group management mechanisms. The *group output* is a function of the outputs of individual group members. For example, the group output can be the output generated by the fastest member of the group, the output generated by some distinguished member of the group, or the result of a majority vote on group member outputs. We use the phrase "group g has F failure semantics" as a shorthand for "the failures that are likely to be observable by users of g are in class F".

A server group able to mask from its clients any k concurrent member failures will be termed k-fault tolerant; when k is 1, the group will be called *single*-fault tolerant, and when k is greater than 1, the group will be called *multiple*-fault tolerant. For example, if the k members of a server group have crash/performance failure semantics and the group output is defined to be the output of the fastest member, the group can mask up to k-1 concurrent member failures and provide crash/performance failure semantics to its clients. Similarly, a primary/standby group of k servers with crash/performance failure semantics, with members ranked as primary, first backup, second backup, ... , (k-1)th backup, can mask up to k-1 concurrent member failures and provide crash/performance failure semantics. A group of $2k+1$ members with arbitrary failure semantics whose output is the result of a majority vote among outputs computed in parallel by all members can mask a minority, that is, up to k member failures. When a majority of members fail in an arbitrary way, the entire group can fail in an arbitrary way.

Hierarchical and group masking are two end points of a continuum of failure masking techniques. In practice one often sees approaches that combine elements of both. For example, a user u of a primary/backup server group that sends its service requests directly to the primary might detect a primary server failure as a transient service failure and might explicitly attempt to mask the failure by re-sending the last service request [B81]. Even if the service request were automatically resent for u by some underlying group communication mechanism which matches service requests with replies and automatically detects missing answers, it is likely that u contains exception handling code to deal with the situation when the entire primary/backup group fails.

The specific mechanisms needed for managing redundant server groups in a way that masks member failures and at the same time makes the group behavior functionally indistinguishable from that of single servers depend critically on the failure semantics specified for group members and the communication services used. The stronger the failure semantics of group members and communication, the simpler and more efficient the group management mechanisms can be. Conversely, the weaker the failure semantics of members and communication, the more complex and expensive the group management mechanisms become.

To illustrate this other general rule of fault-tolerant computing, consider a single-fault tolerant storage service S. If the elementary storage servers used to build S have read omission failure semantics (error detecting codes ensure that the probability of read value failures caused by bit corruptions is negligible), one can implement S as follows: use two identical, physically independent elementary servers s, s', interpret each S-write as two writes on s and s', and interpret each S-read as a read of s, and if the s-read results in an omission failure, a read of s'. If the elementary storage servers are likely to suffer both omission and read value failures, that is, it is possible that in response to a read either no value is returned or the value returned is different from the one written, then three elementary, physically independent servers are needed for implementing S: each S-write results in three writes to all servers and each S-read results in three elementary reads from all servers and a majority vote on the elementary results returned. If a majority exists, the result of the S-read is the majority value read. If no majority exists, the S-read results in an omission failure. The S service implemented by triplexing and voting is not only more complex and expensive than the service S implemented by duplexing, but is also slower.

Other illustrations of the rule that group management cost increases as the failure semantics of group members and communication services becomes weaker are given in [CASD85] and [ES86], where families of solutions to a group communication problem are studied under increasingly weaker group member and communication failure semantics assumptions. Statistical measurements of run-time overhead in practical systems confirm the general rule that the cost of group management mechanisms is higher when the failure semantics of group members is weaker: while the run-time cost of managing server-pair groups with crash/performance failure semantics can sometimes be as low as 15% [Ba89], the cost of managing groups with arbitrary failure semantics can be as high as 80% of the total throughput of a system [PB85].

Since it is more expensive to build servers with stronger failure semantics, but it is cheaper to handle the failure behavior of such servers at higher levels of abstraction, a key issue in designing multi-layered fault-tolerant systems is how to *balance* the amounts of failure detection, recovery, and masking redundancy used at the various abstraction levels of a system, so as to obtain the best possible *overall* cost/performance/dependability results. For example, in the case of the single-fault tolerant storage service described above, the combined cost of incorporating effective error correcting codes in the elementary storage servers and implementing a single-fault tolerant service by duplexing such servers is in general lower than the cost of triplexing storage servers with weaker failure semantics and using voting. Thus, a small investment at a lower level of abstraction for ensuring that lower level servers have a stronger failure semantics can often contribute to substantial cost savings and speed improvements at higher levels of abstraction and can result in a lower overall cost. On the other hand, deciding to use too much redundancy, especially masking redundancy, at the lower levels of abstraction of a system might be wasteful from an overall cost/effectiveness point of view, since such low level redundancy can duplicate the masking redundancy that higher levels of abstraction might have to use anyway to satisfy their own dependability requirements. Similar cost/effectiveness "end-to-end" arguments in layered implementations of fault-tolerant communication services have been discussed in [SRC84].

2.6 On choosing the right failure semantics

When is the probability that a server r suffers failures outside a given failure class F small enough to be considered "negligible"? In other terms, when is it justified to assume that the only "likely" failure behaviors of r are in class F? The answer to these questions depends on the stochastic requirements placed on the system u of which r is a part.

The specification of a server r must consist of not only *functional* requirements S_r and F_r that prescribe the server's standard and failure semantics, but also of a *stochastic* specification. The stochastic requirements should prescribe a minimum probability s_r that the standard behavior S_r is observed at run-time, as well as a maximum probability c_r that a (potentially catastrophic) failure different from the specified failure behavior F_r is observed. When a higher level server u that depends on r is built, critical design decisions will depend on S_r and F_r. Any verification that the design satisfies u's own standard and failure functional specifications S_u and F_u also relies on S_r and F_r [C85]. To check that u satisfies its stochastic specifications, a designer has to rely on s_r, c_r and stochastic modelling/simulation/testing techniques

[T82] to ensure that the probability of observing S_u behaviors at run-time is at least s_u and the probability of observing unspecified (potentially catastrophic) behaviors outside S_u or F_u is smaller than c_u. If c_r is small enough to allow demonstrating that the design of u meets the stochastic requirements s_u, c_u, then F_r is a failure semantics that is *appropriate* for using r in the system u. If c_r is significant enough to make such a demonstration impossible, the designer of u has to settle for a failure semantics F_r' weaker than F_r and re-design u by using new redundancy management techniques that are appropriate for F_r'.

To illustrate this point, consider a single-fault tolerant storage service S specified as follows: the standard functional specification requires that an S-read following an S-write returns the value written; the failure specification states that S-writes never fail and the only admissible S-read failures are omission failures (reading a value different from the one previously written is considered to have potentially catastrophic consequences); the stochastic specification requires that the probability of observing the specified standard behavior be at least 1-f_S, where $f_S = 10^{-10}$, and the probability of observing an S-read value failure be at most $c_S = 10^{-18}$.

Assume that, to build S, one decides to use a duplex design based on two physically independent storage servers s, s' which use 1 bit error detecting codes. The specification of these elementary storage servers is as follows: an s-read returns the value previously written with probability at least 1-f_s, where $f_s = 10^{-9}$; an s-write always succeeds; the probability that an s-read returns a value different from the one written is at most $c_s = 10^{-19}$ (less than one in 10^{10} s-read failures is a value failure); when an s-read value failure occurs, that is, the value read V is different form the value written V_0, V can be any value among 10^{10} storage word values that are different from V_0.

A simple stochastic calculation shows that the duplex design described in Section 2.5 satisfies the S requirements: the probability of an S-read omission failure is of the order of 10^{-18} and the probability of an S-read value failure is of the order of 10^{-19}. Thus, the omission failure semantics assumed by the duplex S design is appropriate: to build S one can "neglect" the probability $c_s = 10^{-19}$ that this failure assumption is violated at run-time.

If the requirement is to design a more reliable storage service S' with a specification as before, except that the probability of an S'-read value failure should be at most $c_{S'} = 10^{-20}$, then the duplex design based on the omission failure hypothesis for memories with 1 bit error detecting codes is no longer adequate. Indeed, for that design, the probability $c_s = 10^{-19}$ that an s-read returns an undetectably corrupted value that is passed to an S' user is unacceptably high. A simple stochastic calculation shows that the triplex design with voting described in Section 2.5 based on three memories with 1 bit error detecting codes satisfies the S' requirements (for simplicity, we assume perfect voting). Indeed, the triplex design ensures that the probability of an S'-read omission failure is of the order of 10^{-18} and the probability of an S'-read value failure is of the order of 10^{-48}. Thus, if the goal is to build the S' storage system, it is no longer appropriate for a designer to assume that memories with 1 bit error detection code have omission failure semantics. For S' the probability c_s of observing an s-read value failure becomes "non-negligible" and the designer must adopt the weaker failure hypothesis that the memories with 1 bit error correcting codes used have omission/value failure semantics. Other examples of how to choose server failure semantics for highly dependable systems are discussed in [Po89].

The remainder of this paper assumes that preliminary stochastic analyses or practical statistics have helped settle the issue of the adequacy of choosing a certain failure semantics over another one in the context of the systems to be described.

3 Hardware architectural issues

To provide processor service to application software servers, an operating system needs hardware resources such as CPUs, memory, I/O controllers, and communication controllers. Some hardware architectures package several of these basic resources into single replaceable units. Other architectures make each one of these resources a replaceable unit by itself. By a *replaceable hardware unit* we mean a physical unit of failure, replacement and growth, that is, a unit which fails independently of other units, can be removed from a cabinet without affecting other units, and can be added to a system to augment its performance, capacity, or availability. The ultimate goal behind hardware replaceable units is to enable them to be physically removed (either because of a failure or because of preventive maintenance or horizontal growth) and inserted back into a system without disrupting the activity of software servers running at higher levels of abstraction. This is often too expensive or impossible to achieve. The next goal is to ensure that the service provided by the hardware servers in each replaceable unit has a "nice" failure semantics, such as crash or omission, so that the higher software levels of abstraction can provide low-overhead hardware failure masking by relying on such stronger failure semantics. Depending on whether or not a qualified field engineer is needed to remove or install a replaceable unit, it is customary to classify them into *field replaceable* (which need the intervention of a field engineer) and *customer replaceable* (which need not).

3.1 What are the replaceable units? How are they grouped and connected?

Depending on the granularity of the replaceable hardware units of a processor architecture, it is possible to distinguish between coarse granularity architectures and fine granularity architectures. In a coarse granularity architecture, some replaceable units package together several elementary hardware servers such as CPUs, memory, I/O controllers and communication controllers. In a fine granularity architecture, each elementary hardware server is a replaceable unit by itself. Some examples of commercially successful coarse granularity architectures are Tandem [B81], the DEC VAX Cluster [KLS86] and the IBM MVS/XRF [IBM87]. Examples of fine granularity architectures are Stratus [TW89] and Sequoia [B88]. Other examples of fault-tolerant architectures can be found in [AL81], [La90], and [S90].

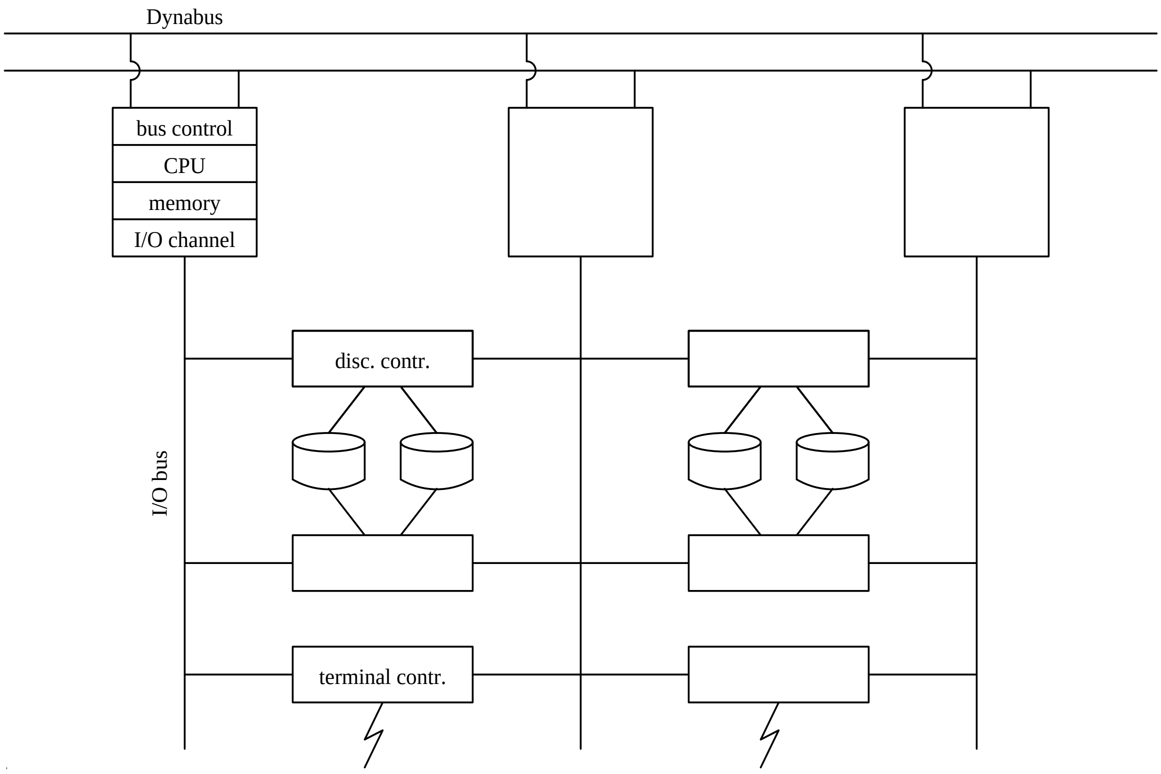

Figure 2: Tandem Processor Architecture

The Tandem processor architecture packages CPU, memory, bus and I/O controller servers into single replaceable units as illustrated in Figure 2. These units can communicate among themselves via a dual bus called Dynabus. Disk, tape, communication, and terminal controller servers are field replaceable units by themselves. While in high-end Tandem systems, CPU/memory units are field replaceable, in the newer, low-end CLX systems they became customer replaceable. The key ideas behind the Tandem architecture were as follows: 1) ensure the existence of *two* disjoint access paths from any terminal controller to any physical servers needed for interpreting user commands, such as CPU, memory, disk, or tape, and 2) group servers into failure masking *server pair groups*. The operating system implements the pair management algorithms and decides which resources play an *active* role in interpreting user commands and which resources play a backup role. The operating system uses a combination of hierarchical and group masking techniques to mask hardware resource failures from users: if one of the active hardware resources on the active path fails, then the operating system uses the other path to continue to provide service to the affected users [B81]. For example, if a software server interpreting user commands uses a disk via a certain disk controller, and the controller fails, the operating system masks the failure by re-directing further disk accesses through the other disk controller. The operating system can also ensure that disk, bus, or bus controller failures are masked from higher level application processes. If a CPU/memory replaceable unit fails, any software server executing on that unit also fails. The hardware architecture ensures that another CPU/memory unit with access to the resources needed by the failed servers exists. If a failed server is a primary in a group implementing an operating system service, such as disk I/O, its failure is automatically masked from higher level user servers by the promotion of its backup [B81] to the role of primary. Since user level application servers are generally not implemented by process pairs, to avoid the complications associated with programming periodic check-pointing [G86], the failure of user application servers is visible to human users which have to wait until these servers are re-started.

The duplication of hardware resources needed by software servers makes the Tandem architecture single-fault tolerant: any single hardware replaceable unit failure can be tolerated, provided no second replaceable unit failure occurs. As reported in [G86], the use of server pair groups to implement operating system services also enables the masking of a significant fraction of the operating system server failures caused by residual software design faults. Single-fault tolerance does not mean that it is impossible to mask two or even more concurrent hardware replaceable unit failures. For example, the simultaneous failure of two disk controller servers attached to distinct disks can be masked from higher level software servers. What single fault-tolerance means is that the architecture guarantees masking of *any single* hardware replaceable unit failure but it does not guarantee that *any two* concurrent replaceable unit failures can be masked. That is, there exist double failures which cannot be masked. For example, if two CPU/memory or disk controller replaceable units attached to the same disk fail, then any service that depends upon the disk also becomes unavailable. All the other commercial system architectures that we will discuss are single fault-tolerant, so we will refrain from mentioning this point further.

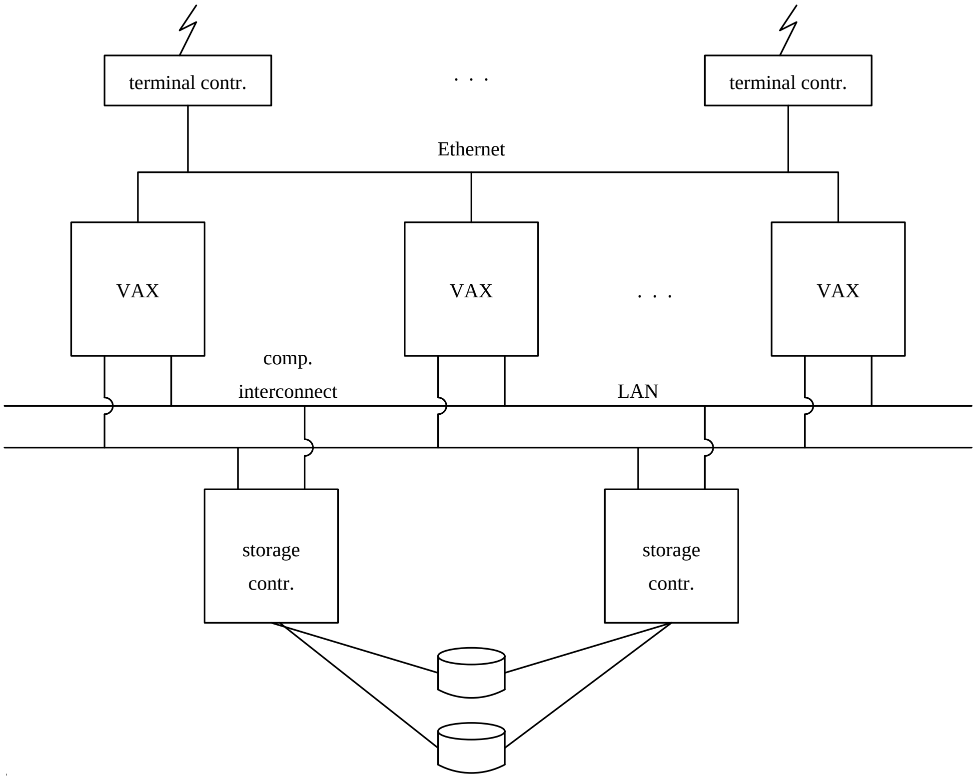

Figure 3: VAX Cluster Processor Architecture

In the VAX Cluster processor architecture (Figure 3), the field replaceable units are entire VAXes (containing CPU, main storage, as well as LAN controllers), entire storage controllers (containing microprocessors, specific device drivers, as well as LAN controllers) or entire terminal controllers. Two kinds of local area networks can be used in a VAX Cluster: a high speed custom-designed Computer Interconnect (in Figure 3 this is duplexed and connects several VAXes and storage controllers) and a low speed Ethernet. For a VAX cluster to be single-fault tolerant, dual LANs are needed for connecting processors to storage controllers. In a manner similar to the Tandem architecture, the idea is to ensure the existence of at least two disjoint access paths from any terminal controller to any physical resources such as CPU, memory, disk, or tape needed by a software server which interprets user commands. Hardware replaceable unit failures are masked by a combination of hierarchical and group masking techniques. Disks can be dually ported to different storage controllers, so that if a controller fails, the operating system can re-direct I/O to an alternate controller. A computer interconnect permanent failure is similarly masked at operating system level by re-directing all remaining traffic on the alternate computer interconnect. If a VAX fails, then all software servers which were running on it also fail. Any other VAX with enough capacity in the cluster can become active in running these servers, but they have to be explicitly re-started, as was the case with user servers in the Tandem system. Thus, for both the Tandem and VAX cluster architectures, single failures of CPU/memory replaceable units can result in the failure of higher level application servers, and these failures are visible to human users.

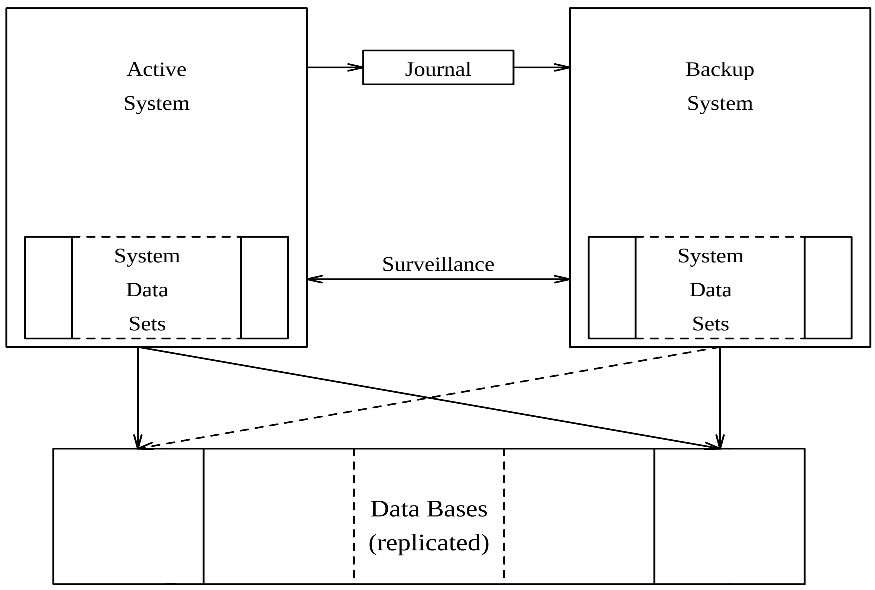

Figure 4: IBM XRF Architecture

In the IBM XRF architecture (Figure 4) the replaceable hardware units are entire high-end IBM processors. (In general, whenever previously existing processor architectures have to be integrated into a fault-tolerant system based on a local area network, the granularity of the resulting architecture is naturally coarse.) The XRF system provides continuous IMS database service by using group masking: a group of two IMS servers running on two distinct high-end processors connected by a point-to-point local area network to provide IMS service. One of the servers, the primary, interprets user requests and has an up-to-date view of the application state, while the other server, the backup, lags behind in its knowledge of the current application state. The backup maintains this delayed knowledge by reading a log generated by the primary. This arrangement allows hardware and operating system failures to be masked to users of the IMS service.

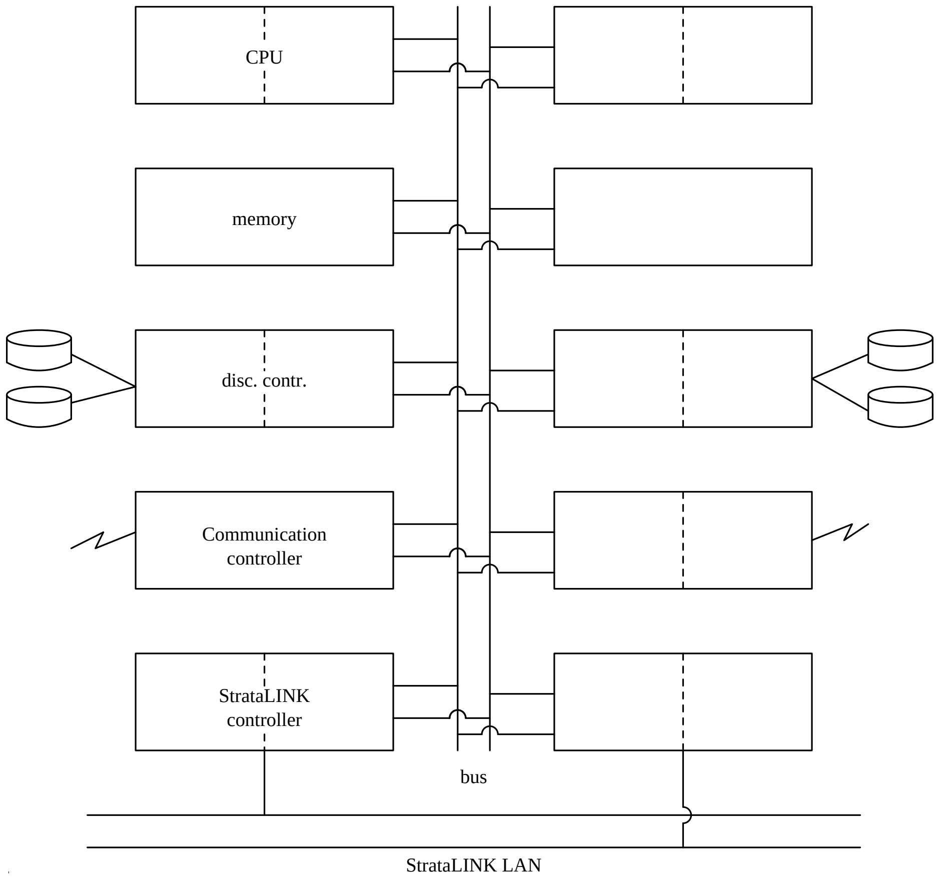

Figure 5: Stratus Processor Architecture

The Stratus processor architecture (Figure 5) was the first to introduce the systematic use of CPU, I/O and communication controller hardware servers implemented by paired microprocessors that execute identical instruction streams and continuously compare their results. This implementation technique ensures that such servers have crash failure semantics with very high probability. Another characteristic of this architecture is its fine granularity: most elementary hardware servers are customer replaceable units with their own power supply. This can substantially reduce maintenance costs and ease horizontal growth. A third idea is to use group masking at the hardware level by pairing elementary hardware servers with crash or omission failure semantics. This enables some of the hardware server failures to be masked from the higher software levels. For example, in order to provide single-fault tolerant processor service to software servers, two CPU servers with crash failure semantics (each of them implemented by a pair of microprocessors executing in lock-step that continuously compare their results) are organized as a masking group. Each group member receives the same sequence of inputs. In the absence of member failures the output of either member is accepted. If one member suffers a crash failure (detected as a disagreement among the two microprocessors which implement it), the output is taken from the other member. Similarly, memory banks with read omission failure semantics (implemented by using two bit detection/one bit correction codes) can be duplexed to obtain a single-fault tolerant storage service with read omission failure semantics. Not all elementary hardware server failures are masked directly at the hardware level. For example, disk failures are hierarchically masked at the operating system level, since at the lower hardware levels there is not enough state information to enable pure hardware group masking to take place. A dual bus, called the Stratabus, enables the replaceable units of a processor to communicate among themselves despite any single bus failure, and a dual LAN named Stratalink enables processors to communicate among themselves despite any single Stratalink failure. Terminals can optionally be connected to a processor by a pair of communication buses redundantly connected to two communication controllers attached to the processor Stratabuses. This enables the system to mask single communication controller or bus failures without requiring the user to move from one terminal to another one. A double hardware failure or an operating system failure can cause an operating system crash. In this case all the user servers that were using services provided by that operating system also fail.

Figure 6 shows the Sequoia multi-processor architecture, another example of a fine granularity architecture. The CPU and I/O controller services are implemented by paired microprocessors that execute the same instruction stream in lock-step and continuously compare their results, as in the Stratus architecture. This ensures that the CPU and I/O controller hardware services used by operating system servers have crash failure semantics. Memory servers rely on error detecting/correcting codes to implement omission failure semantics with very high probability. Individual CPU, memory, and I/O controller servers are packaged as customer replaceable units. Fault-tolerant communication among replaceable units is provided by a dual system bus. Each bus is composed from a processor segment that connects CPUs, a memory segment that connects memory elements and I/O elements, and a global segment which connects the previously mentioned buses via master (MI) and slave (SI) interfaces. Both the master and slave interfaces provide electrical isolation for the adjacent bus segments for error confinement reasons. The operating system uses hierarchical masking techniques to hide any single hardware server failure from the software processes running above it. For example, when a CPU server crashes, the operating system automatically attempts to re-start the software server that was executed by the failed CPU on another CPU server by using a previously saved check-point of the server state. To ensure that server state updates are atomic with respect to CPU crashes, a shadowing

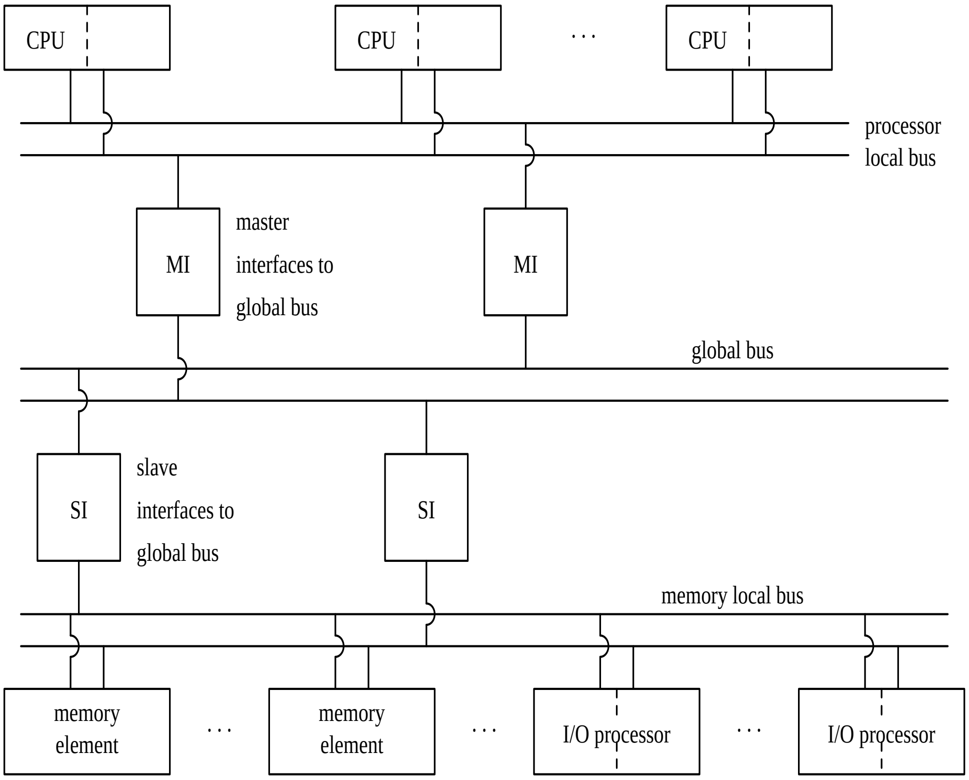

Figure 6: Sequoia Multiprocessor Architecture

technique is adopted. This requires that the operating system maintains two copies of the state of a software server on two different memory units. To mask memory read omission failures, all writable data pages are duplicated on different physical memory elements, and all read or executable-only pages are also stored on disks. If a memory element fails, the data page which could not be read is recovered either from another memory element or a disk, depending on whether it was a writable or only a readable or executable page. The operating system can also mask single disk failures by duplicating disk pages on dual-ported disks attached to different I/O controllers. In this way, the operating system can mask any single hardware replaceable unit failure as long as at least two CPU, two I/O controller, and two memory servers work correctly and all failures that occur are fully recovered before a second failure occurs. When the number of correctly working hardware servers drops below these thresholds, the operating system can continue to provide processor service with crash/performance failure semantics as long as at least one CPU, one I/O controller and one memory server continue to work correctly. If multiple concurrent failures occur or CPU, memory or I/O controller service is no longer available because all CPU, memory, or I/O controllers have failed, the operating system, and hence all higher level software servers will fail. Processor service failures can of course also occur because of residual design faults in the operating system.

3.2 What failure assumptions are made about hardware replaceable units?

Developers of operating system software typically assume that CPU, I/O and communication controller servers have crash failure semantics, that memory elements have read omission failure semantics, that disks have seek performance failure semantics and read/write omission failure semantics, and that communication buses and communication lines have omission or omission/performance failure semantics. Available field statistics indicate that, for commercial on-line transaction processing systems characterized by downtimes of the order of minutes or hours per year, these failure assumptions are adequate. All previously mentioned systems make these assumptions. Such strong hardware failure semantics enable system designers to use known hierarchical masking techniques to mask hardware server failures, such as storage duplexing to mask loss of data replicated on two memory or disk servers with read/write omission failure semantics [LS81] or virtual-circuits to mask omission or performance communication failures by using time-outs, sequence numbers, acknowledgements and retries [T81]. Moreover, when masking is not possible, strong hardware server failure semantics such as omission and crash enable system programmers to ensure that the operating system and communication services they implement have a strong failure semantics. For example, CPU and disk controllers with crash failure semantics and disks with read/write omission failure semantics enable the implementation of a higher level stable storage service [C85], [LS81] with write operations that are atomic with respect to crashes: any stable storage write interrupted by a crash is either carried out to completion or is not performed at all. Such stable storage service can then be used by other higher level servers to implement atomic transactions by relying on known database recovery techniques such as write-ahead logging and two phase commit [G78]. Similarly, lower level data gram communication services with omission/performance failure semantics enable the implementation of higher level virtual-circuits with crash failure semantics [T81].

Beyond these restrictive hypotheses about the failure semantics of the basic hardware replaceable units, the complexity and cost of detecting, diagnosing, and recovering from elementary hardware server failures increases to levels that many regard as unacceptable for most commercial applications in on-line transaction processing and telecommunications. Examples of processor architectures designed for highly critical application areas were, because of ultra-high stochastic dependability requirements, designers had to assume that their elementary hardware servers have arbitrary failure semantics can be found in [Ha78], [HLD88], and [Wa78].

3.3 How is the specified hardware failure semantics implemented?

In order to detect failures in buses, communication lines, memory and disk servers, all the previously discussed architectures use error detecting codes. This hardware failure detection technique is well understood. There is a rich literature specializing in error detecting codes and this subject seems to have reached a fairly mature state [PW72], [W78]. To detect failures in hardware servers, such as CPU, I/O and communication controllers, systems such as IBM, VAX and high-end Tandem use error detecting codes while newer systems such as Stratus, Sequoia, Tandem CLX and DEC VAXft 3000 use lock-step duplication with comparison.

While error-detecting codes seem to remain the choice method for detecting failures in storage and communication hardware servers such as memories, disks, buses and communication lines, they seem to give way to duplication and matching in complex circuits, such as CPUs and device and communication controllers based on off-the shelf microprocessors. One reason for this trend is that duplication and matching seems to provide a better approximation of crash failure semantics for these complex servers than error detecting codes. Indeed, while for CPUs and I/O controllers based on error detecting codes there is a possibility that the data written to a bus or storage during the last "few" cycles before a failure detection is erroneous, duplication and matching by using self-checking [M82] comparator circuits virtually eliminates the possibility of such damage. The cost for this excellent failure detection capability is that two physical hardware servers plus the comparison logic are needed instead of only one elementary server augmented with error detecting circuitry.

Besides providing a better guarantee of crash failure semantics for complex servers such as CPU and I/O controllers, duplication and matching has a number of other attractive characteristics. The absence of error detecting circuitry in elementary physical servers reduces their complexity, leading to increased reliability and reduced design and testing costs. The elimination of the error detecting circuitry also makes these servers faster. Another reason for using lock-step duplication is the availability of cheap fast microprocessors which do not have much error detection circuitry. Pairing these off-the-shelf components and adding a comparator can be cheaper than developing proprietary processor designs with elaborate error-detecting capabilities.

Moreover, pairing off-the-shelf processors enables computer system manufacturers to "ride the wave" of rapid improvements in chip speed and update their product lines promptly as new chips become available on the market. One last advantage of lock-step duplication worth mentioning is improved software quality growth. When any elementary hardware server failure is promptly detected as a disagreement before any data damage occurs, it is easier to disambiguate failures caused by software design faults from failures caused by physical faults. Indeed, if an operating system failure occurs and no disagreement between hardware processors is observed, then with high probability, the failure is due to a software design fault. This is in sharp contrast with what occurs in more traditional processor architectures based on error correcting codes. Since in such architectures hardware failures in CPU and I/O servers can result in undetectable damage to data due to latencies in failure detection, a large class of system failures attributable to the software are in fact caused by the hardware. The knowledge that hardware failures can with significant probability "masquerade" as software design faults can create serious difficulties in the proper diagnosis of system failures. In this case operating system developers may blame hardware malfunctioning for system failures they cannot diagnose, and hardware developers may blame the operating system for failures where the hardware apparently has worked properly.

3.4 How are replaceable hardware unit failures masked?

It is possible to implement the redundancy management mechanisms which mask hardware server failures directly *in hardware*, for example by using group masking techniques such as triplexing physical hardware servers with arbitrary failure semantics and voting [Ha78] or duplexing hardware servers with crash failure semantics (which in their turn can be implemented by server pairs based on duplication and matching [TW89]). This allows single processor failures to be masked from higher software levels of abstraction and increases the mean time between failures for the raw processor service. Note, however, that hardware triplexing or quadruplexing does not eliminate the need for handling at application software levels processor service failures in the same way as if this service were implemented by a non-redundant processor. Although such failures will occur less frequently, they will occur and must be handled. For example, if the transactions implemented at a database service level must be atomic with respect to processor service crashes, the code for implementing the transaction failure semantics will have to be written, possibly by relying on logging and recovery facilities provided by the underlying operating system, whether the processor service is implemented by using a simple CPU or a quadruplex CPU arrangement. Note also that hardware triplexing or quadruplexing is of no help in providing fault-tolerance at the operating system and application level.

Several systems attempt to mask hardware server failures at the *operating system* level, so that application software servers can continue to run without interruption. For example, the Sequoia and MVS operating systems use a combination of hierarchical and group masking methods to hide single CPU failures from higher level software servers by restarting a server that ran on the failed CPU in a manner transparent to that server. Similar masking of CPU, bus or disk controller failures is done by the operating systems of Tandem, VAX cluster and the IBM XRF architectures. Although the choice of masking hardware server failures at the operating system level can provide tolerance not only to hardware, but also to operating system server failures [G86], it does not solve the problem of how to ensure fault-tolerance of application services.

The use of redundant application software server groups allows both hardware, operating system and software server failures to be masked at the highest level of abstraction: the *application level*. The idea is to implement any service that must be available to users despite hardware or software failures by a group of redundant software servers which run on distinct hardware processor hosts and maintain redundant information about the global service state. When a group member fails, either because of a lower level hardware or software service failure or because of a residual design fault in its program, the surviving group members have enough service state information to continue to provide the service. Although the approach of masking server failures at the highest, application level is the most "powerful", in that it can mask both hardware and software server failures, among the successful commercial systems discussed previously, only Tandem and IBM XRF make use of redundant software server groups.

4 Software architectural issues

Software servers are analogous to hardware replaceable units. These are the basic units of failure, replacement, and growth of software. As with hardware replaceable units, the ultimate goal is to enable software servers to be removed from a system (due to failures, upgrades, and horizontal growth) without disrupting the activity of the users. When this is impossible to achieve, either because of cost and complexity reasons or because the number of active servers providing a service drops below some threshold, the next goal is to ensure that software servers have a "nice" failure semantics, such as crash, omission or performance. This will allow their (possibly human) users to recover from failures by relying on simple masking protocols such as "log in again" or "wait for some time and try again". Because of the similarity in goals, the issues that have to be dealt with at the software architecture level are similar to the ones discussed in Section 3. The methods for solving these issues, however, can be quite different in their implementation details.

4.1 What failure semantics is specified for software servers?

Depending on whether the state of a service is persistent or not, one might require the application servers that implement the service to provide atomic transaction failure semantics or simply amnesia or partial-amnesia crash failure semantics. An atomic transaction, which is a sequence of operations on persistent data usually stored in databases, must change a consistent database state into another consistent database state or must not change the database state at all when a lower level processor service crash occurs [G78], [LS81]. For software servers, such as low level I/O and communication controllers which typically do not have persistent state, one is usually content with amnesia crash failure semantics: after a failure these servers must properly re-initialize themselves and accept new service requests.

To implement atomic transaction or simply crash failure semantics, the commercial systems described previously assume that the programs that implement the operations exported by software servers are totally, or at least partially, correct [C89]. A program is *totally* correct if it behaves as specified in response to any input as long as the services it uses do not fail. A *partially* correct program may suffer a crash or a performance failure for certain inputs even when the services it uses do not fail. In this way, if a server delivers a result or performs a state transition in response to a service request, that result/state transition is correct, although the result may not be timely or even delivered at all. Partial correctness as a basic hypothesis about the failure semantics of software servers is appealing for several reasons. First, it is more easily achievable than total correctness. Second, it seems to be achievable in practice: after undergoing extensive reviews and tests, the software available commercially occasionally crashes or is slow, but does not output bad results. Third, a partially correct server, which by definition has crash/performance failure semantics when the lower level software and hardware services it depends upon *do not* fail, maintains its crash/performance failure semantics when these lower level services *do* fail, provided the lower level service failures are crash, omission or performance failures. Thus, if at all levels of abstraction of a system, such as hardware, operating system, and applications, all servers have crash/omission/performance failure semantics, the system will have crash/omission/performance failure semantics as a whole. This yields a simple to understand, homogeneous quality assurance goal for *all* abstraction levels:

make the hardware have crash or omission failure semantics, and make the software be totally, or at least partially, correct.

Some special-purpose systems do not assume that their component servers have "nice" failure semantics, such as crash/omission/performance [Aa85], [Ha78], [HLD88] and [Wa78]. By assuming that group management services such as clock synchronization, atomic broadcast, and voting work correctly, such systems can mask a *minority* of server failures in application groups built from members with arbitrary failure semantics. Depending on the assumptions made about the faults which cause the failures, the systems can be classified in two classes: those that attempt to tolerate only physical faults [Ha78], [HLD88] and [Wa78], and those that attempt to also tolerate design faults [Aa85]. The first class of system replicates the application servers on different, physically independent processors. Since physical faults tend to occur independently in independent processors, such faults are likely to cause minority application group failures. Experience with the systems [Ha78], [HLD88] and [Wa78] confirms that these can effectively mask the consequences of physical faults. Recent work on diverse software design [A89] pursues the goal of producing a diverse software design methodology that would ensure that groups of application servers running diverse programs do not suffer majority failures despite the existence of residual design faults in these programs. Perhaps because there are no accepted techniques to estimate with confidence the increase in reliability that results from the use of diverse programming with voting, the work on design diversity has generated considerable controversy to date [A89], [KA89].

4.2 How is the specified software server failure semantics implemented?

The problems related to the implementation of atomic transactions on persistent data despite processors with crash/performance failure semantics, disks with omission failure semantics, and communications with omission/performance communication failures have been investigated intensively for more than twenty years by researchers in the area of database systems. Monographs and books, such as [BHG87] and [G78] treat the subject in great detail. To separate concerns, these monographs assume that the programs used in implementing transaction atomicity are at least partially correct. The problems that need to be solved in implementing totally or partially correct programs have been the object of intensive study for the last four decades in the software engineering community. Over the last thirty years, several ideas have emerged that have proven effective in helping software designers prevent the introduction of design faults in programs. Among these, we mention the pursuit of simplicity and clarity during design, hierarchical design methods based on information hiding and abstract data types, rigorous design specification and verification techniques, systematic identification, detection, and handling of all exception occurrences, and use of modern inspection and testing methods. None of the papers describing the commercial systems discussed earlier deal with this issue; however, it is reasonable to assume that all have extensively used modern software engineering methods when attempting to ensure that their software is totally, or at least partially, correct.

Among the systems discussed, most provide concurrency control and recovery support for implementing application servers with atomic transaction failure semantics. The goal is to ensure that concurrently executing transactions are serializable, that is, their execution is equivalent to a serial execution, and that sequences of changes made to stable storage are either performed to completion or aborted. These servers rely on techniques such as locking, logging, disk mirroring, and atomic commit [BHG87], [G78], and [LS81].

4.3 How are software server failures masked?

All of the commercial systems discussed earlier use hierarchical masking techniques to mask hardware and certain software server failures. While such techniques can be effective in masking crash, omission and performance server failures as long as the underlying processor service needed by these servers does not fail, one needs a group masking technique to tolerate software server failures caused by failures of the underlying processor service.

Software group masking techniques based on members with crash/performance failure semantics are used by two of the commercially successful systems discussed previously: Tandem and IBM XRF. In Tandem systems, processes pairs are used for implementing fault-tolerant basic operating system services such as disk I/O service, spool service, or logging/commit service, while in the XRF system pairs of primary/backup database and communication servers are used to implement fault-tolerant database and communication services.

A prerequisite for the implementation of a service by a software server group capable of masking processor service failures is the existence of multiple *host processors* with access to the physical resources used by that service. For example, if a disk containing a database can be accessed from four different processors, then all four processors can host database servers for that database service. A four member database server group can then be organized to mask up to three concurrent processor failures. More generally, *replication* of the resources needed by a service is a prerequisite for making that service available despite individual resource failures. For example, the use of at least two disks with omission failure semantics to store a database enables one to implement a database service that can mask any single disk failure.

The use of software server groups raises a number of novel issues that are not well understood. First, how should group members running on different processors maintain consistency of their local states in the presence of member failures, member joins, and communication failures? Second, how should server groups communicate? Third, how is it automatically ensured that the required number of members is maintained for server groups despite operating system, server, and communication failures?

The difficulty of solving these problems might well be one of the main reasons why software server groups are not more widespread in commercial applications. A second reason might be that, while for a failed elementary hardware server such as a CPU or a memory bank the repair can take hours or even days, for a software server that crashes the "repair", that is, the restart, can often only take minutes. Thus, since elementary hardware server failures can heavily contribute towards the user visible service downtime, most manufacturers prefer to attack this problem first. A third reason why software server groups are unpopular is the additional complexity associated with the group management mechanisms and their inherent run-time overhead. A fourth reason might be that it is not *a priori* clear whether low overhead groups of members with crash/performance failure semantics can provide effective tolerance to residual software design faults. Recent statistics [G86] show that software server group management mechanism designed for servers with crash/performance failure semantics [B81] can be realistically effective in masking server failures caused by hardware faults as well as residual design faults left in production quality software after extensive reviews and testing. An example reported in [G86], which is based on a sample set of 2000 Tandem systems representing over 10 million system hours, indicates that for the primary/backup spooling process group used in Tandem's distributed operating system, only 0.7% of the failures affecting the group were double server failures, that is, group failures. The remaining failures (99.3%) detected in the group were single member failures that left the other member, and hence, the group, running correctly. A plausible explanation for this phenomenon is that most physical faults affecting physically independent processors occur independently, and hence cause any affected software servers to crash independently. Similarly, most residual software design faults manifest themselves intermittently in rare exceptional conditions that result in time-dependent synchronization errors or in a slow accumulation of resources acquired for temporary use and never released. Such errors and residues seem to accumulate at different rates in different group members and eventually lead individual group members to crash at different times.

4.3.1 How are the local states of group members synchronized?

For any service, the *server group synchronization* policy prescribes the degree of local state synchronization that must exist among the servers implementing the service.

Close synchronization (also called masking or active redundancy) prescribes that local member states are closely synchronized to each other by letting all members execute all service requests in parallel and go through the same sequence of state transitions. The group output depends on the failure semantics assumed for its members. If group members can fail in arbitrary ways, majority voting is the most common method used: a group answer is output only if a majority of the members agree. This group organization masks minority member failures at the price of slowing down the group output time to the time needed for a majority of members to compute agreeing answers and for the voting process to take place. If a majority of members fail concurrently then the group output can be incorrect. If group members have crash/omission/performance failure semantics, any output computed by a member can be sent to users. If all members send their outputs in parallel, the group output can be understood as being the output computed by the set of fastest members. The advantage of this output sending technique is that a group will perform correctly as long as at least one group member stays correct. The drawback is a high communication overhead. To reduce this overhead, group members can be ranked with respect to communication, or *c-ranked*, and output sending can be restricted to the highest functioning c-ranked member. The cost is an increased output delay when the highest c-ranked member fails, due to the need to detect its failure and agree on a new c-ranking among surviving members. Closely synchronized groups of software servers are used by all systems that attempt to tolerate arbitrary server failures, such as [Ha78], [HLD88] and [Wa78]. Examples of closely synchronized groups of members with crash/performance failure semantics are described in [CDD90] and [Co85]. A number of rules for transforming non fault-tolerant services implemented by non-redundant application programs into fault-tolerant services implemented by closely synchronized server groups have been proposed in [L84] and are discussed further in [S86].

In contrast to close synchronization, *loose synchronization* (also called dynamic or standby redundancy) ranks the group members with respect to how closely they are synchronized to the current service state, which can be understood as being the application of all operations requested by clients since the service initialization to the initial service state. To distinguish this ranking from the c-ranking introduced before we call it *s-ranking*. Loose synchronization requires that only the highest s-ranking group member, usually called the primary server, process service requests and record the current service state as its local state. Thus, a prerequisite for using this form of group synchronization is that group members have crash or crash/performance failure semantics. The highest ranking member is also usually the one who sends the group output to users (since it is the first to know them) so the c- and s-rankings often coincide. One or more backup servers with lower s-ranks can log service requests and can receive periodically state checkpoints from the primary. In such an arrangement, the local state of a backup server lags behind the state of the primary: the backup state does not reflect the execution of some recent service requests by the primary server. If the primary fails, the highest s-ranking backup server can recover the state existing before the primary failure by re-executing the service requests logged since the last state check-point obtained from the primary. A particular case of loose synchronization is the situation when the group size is one, that is, there is only a primary server with no active backups. In such a case, the server does the check-pointing and logging. After a failure, the new primary that is started reads the last state check-point and recovers a state that existed before the failure by executing the logged service requests. In this way, the failure can be masked to users, who only experience a delay in getting their response. Variants of this check-pointing rollback masking technique have also been developed for the case when the primary server is implemented by a set of distributed processes which check-point and log service requests independently [JZ87], [KT87], [SY85].

For groups composed of servers with performance failure semantics, the main advantage of loose over close synchronization is that only primary servers make full use of their share of the required replicated service resources, while backups make only a reduced use. This allows more servers to coexist for a given amount of computing power. The main drawback is that the delays seen by clients when group members fail are longer. If we call the maximum sequence of service requests processed by a primary between successive state check-points the primary/backup processing lag, then the worst case delay in answering a client request after a primary failure will not only be composed of the time needed to detect and reach agreement about the primary failure, but also of the time needed by the new primary to absorb the primary/backup processing lag. For on-line transaction processing environments, such delays are considered tolerable. For real-time applications, if the response time required is smaller than the time needed to detect a member failure and to absorb the primary/backup processing lag, close synchronization has to be used. Examples of loosely synchronized server groups are discussed in [B81], [Ba89], [BJ87], [CDD90], [IBM87], and [OL88].

Close and loose synchronization as described above are just two end points of a continuum of synchronization policies. One can imagine intermediate synchronization policies which would share the advantages and drawbacks of these end-points to various degrees.

4.3.2 How many members should each server group have?

For any server group, the *replication* policy prescribes the number of members the group is expected to have. The more servers in a group, the greater its availability and capacity for servicing service requests in parallel. On the other hand, the more members a group has, the higher the cost for communication and synchronization among group members. Given a certain required service availability, one can use stochastic modelling and simulation methods for determining an optimum number of members by taking into account the individual server failure rates, the server failure semantics, the service request arrival rates, and the message communication costs.

Note that the degree of software server redundancy necessary for providing a certain level of service availability influences the host processor redundancy degree for that service. For example, if stochastic modelling/simulation studies show that to achieve a certain radar surveillance and tracking service availability level, it is necessary to use three physically independent and closely synchronized servers, then three host processors are necessary for running these servers in parallel as shown in Figure 7.

4.3.3 What group communication protocols should be used?

Communication among server groups is different from the point-to-point communication services offered by traditional protocols such as SNA and TCP/IP. Moreover, if the group state is replicated at several members, these members need to update the replicas by using special group protocols that ensure replica consistency in the presence of process and communication failures. There already exist a significant number of protocols that have been proposed for group communication [ASC85], [B81], [Ba89], [BD85], [BJ87], [Ca85], [CASD85], [CM84], [Cri89], [CZ85], [GS89], [KT90], [KGR89], [L89], [LG90], [LLS90], [OL88], [MMA90], [S88], [SSCA87], and [VRB89]. This area of research is presently one of the most active. The large variety of approaches proposed reflects the fact that the properties required from group communication protocols depend not only on protocol goals and the failure semantics of

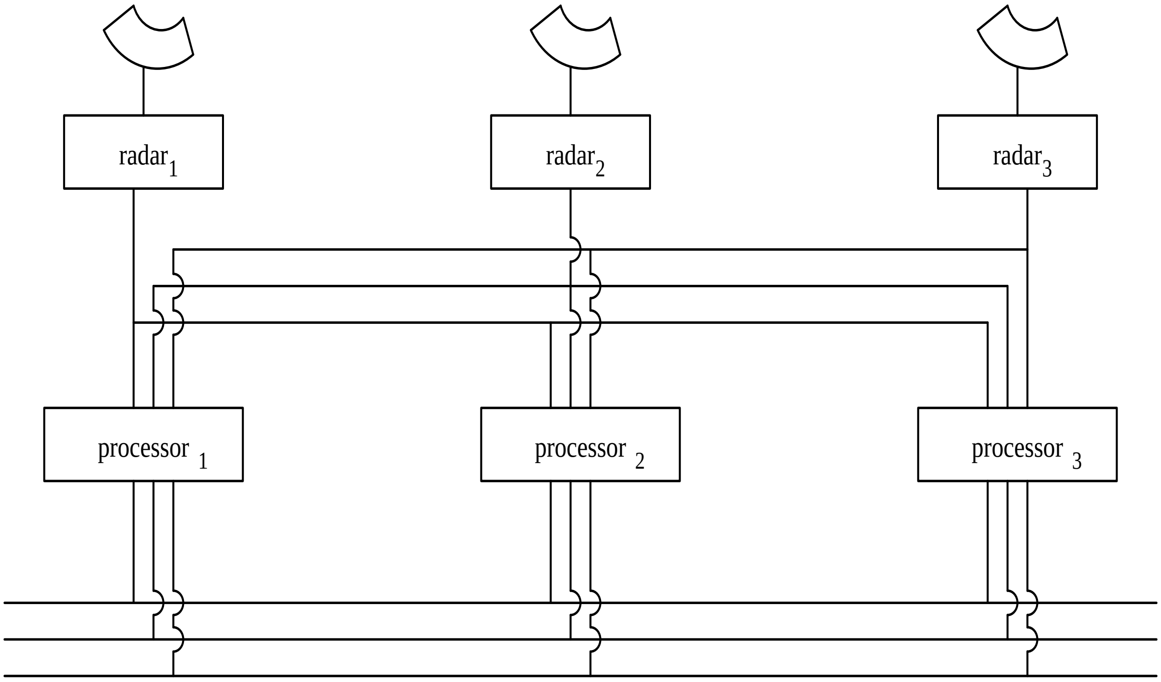

Figure 7: Triple Modular Redundancy

group members, but also on the failure semantics of the lower level communication services used.

Related to communication is the issue of *naming*: what rules are used by clients and servers to define service names, and how are such names translated into lower level message routing information? In fault-tolerant systems, where servers can move from one host processor to another one, it is convenient not to require clients to know the location of the servers that provide a service. The goal of such location-transparent naming services is to mask from clients the effects of individual server failures in a group. Several examples of fault-tolerant location-transparent name services that have been implemented in real-systems can be found in [Ca85], [CDD90], [KLS86]. A recent survey of the issues involved in naming can be found in [CP89].

4.3.4 How to enforce group availability policies automatically?

The synchronization and replication policies defined for a service implemented by a server group constitute the *availability* policy for that service (or group).

One possible approach to enforcing a certain group availability policy, illustrated in

[B81] is to implement in each group member mechanisms for reaching agreement on the c- and s-rankings in the group, as well as mechanisms for detecting member failures and procedures for handling new member joins and promotions to higher s-ranks (when the group is loosely synchronized). The drawback of this approach is that it results in substantial code duplication and lack of modularity, since the group management mechanisms for all services which must be made highly available are basically the same.

An alternative approach, adopted in a recent distributed fault-tolerant system built for air traffic control [CDD90], requires group members to implement only the application specific get-state and check-point (in case the group is s-ranked) procedures needed for local member state synchronization at join and check-point times. The tasks of detecting server failures, coordinating promotions and enforcing replication policies are entrusted to a *service availability management* service. Since the availability in the presence of failures of other distributed system services becomes dependent on the availability of this service, availability management must itself be implemented by a group of servers which we call *service availability managers*. We will assume that these servers have crash/performance failure semantics.

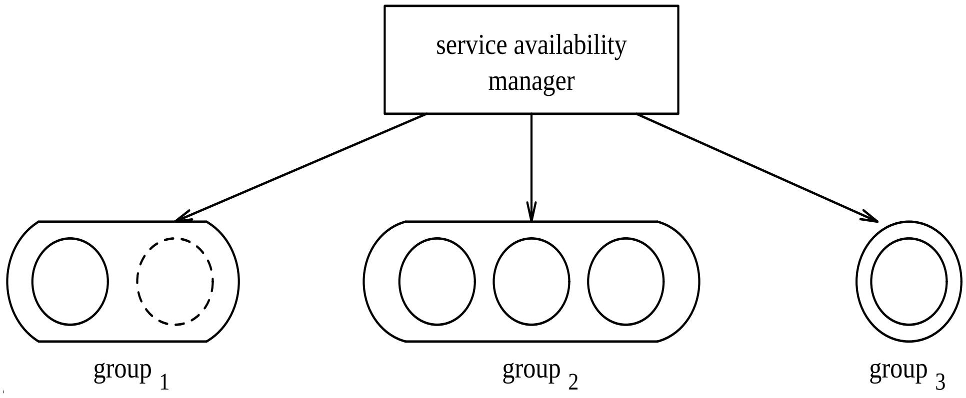

Figure 8: Service Availability Management

The objective is to ensure that for each service s, which the system operator enables the system to provide, the server group availability policy defined for s is effectively enforced without any human intervention. Figure 8, in which an arrow means "enforces the availability policy of" instead of "depends on", attempts to illustrate this goal. To simplify our presentation of an availability management service we assume that for each service s all servers must run on distinct processors and they can be started in any order, i.e., we ignore issues related to the existence of s-rankings in some groups. We also assume that the hosts designated for the various services have enough computing power to support all these services, so we will not mention issues related to load shed and load balancing.

Let *r(s)* be the degree of server replication for s, *h(s)* denote the set of host processors for s, and *members* denote the *processor membership* [C88], that is, the set of correctly functioning processors that agree to communicate among themselves in order to collectively provide to clients all services that are enabled in the system. The goal of the service availability management service is to ensure that for each enabled service s there are $\min(\mid h(s) \cap \mid, r(s))$ servers. That is, there are *r(s)* servers as long as the number of correctly functioning hosts is greater or equal to *r(s)* and, when the number of correctly functioning hosts drops below *r(s)*, there are as many servers as correctly functioning hosts.

The state of each server availability manager consists of the set *members* of processors that function correctly in the system, the set S of services enabled in the system, the set P of servers functioning correctly, and two mappings (or tables) that relate these sets: a mapping *servers*: *members* $\rightarrow$ set-of-P that, for each functioning processor $m \in$ *members*, gives the set *servers(m)* of servers running on m, and a mapping *service*: $P \rightarrow S$ that, for each server $p \in P$, gives the service implemented by p. The state of a server group availability manager can be changed by the following events: a service s is enabled by the system operator, a service s is disabled by the system operator, a server p implementing a service s fails on a processor m, a server p implementing a service s is started on a processor m, a processor m fails, and a processor m starts. The state transitions that an availability manager must undergo in response to these events can be specified as follows. When a service s is enabled, then servers for s must be started on *r(s)* of the processors in $h(s) \cap members$ if $r(s) \leq \mid h(s) \cap members \mid$, else servers must be started on all processors in $h(s) \cap members$; also s must be added to the set S, all servers started must be added to the set P, and for each server started, the mappings *servers* and *service* must be updated accordingly. When a service s is disabled, it must be removed from S, all existing servers for s must be shut down, and the state variables *servers, service*, and P must be updated accordingly. When a server p for a service s fails, another server for s must be started (if there is a free host in $members \cap h(s)$) and the state variables *servers, service* and P must be updated accordingly. The successful start of a server p should be reflected in an appropriate update of the *server, service* and P state variables. The failure of a processor p can be interpreted as the failure of all servers running on p (see the interpretation of server failures sketched above) plus the removal of p from the *members* state variable. The start of a processor p should result in the start of servers for all services s that can be hosted by p and for which the present server population is below the *r(s)* replication threshold specified, as well as in the appropriate updates to the state variables *members, servers, service*, and P.

Assuming the problems of server and processor failure detection are solved, it would not be difficult to build a centralized service availability manager that achieves the state transitions above and runs on some processor in the system. However, such a solution would not be satisfactory because when that processor is down, no server group availability policy would be enforced. As mentioned before, if the objective is to have a specified service availability policy enforced whenever at least one host processor works in the system, then server availability managers must be replicated on all processors that work in system. This results in a need to replicate the global state variables *members, service, P, S* and *servers* on all working processors in *members*. To maintain the consistency of these replicated global state variables at all processors in the presence of random communication delays and failures, the occurrence of any global system state update must be broadcast to all correctly functioning processors in such a way that these processors all see the *same sequence of state updates*.

If different availability managers see different sequences of updates, then their local views of the global system state will diverge. This might lead to violations of a specified server group availability policy. To illustrate this point, consider a service s such that *r(s)*=2 and $h(s) \cap members = \{m, m'\}$. If m sees the sequence of events: "s enabled", "s disabled", and *m'* sees the sequence "s disabled", "s enabled", then at least the state variable S on m will end up being different from the variable S on *m'*. Moreover, *m'* will start a local server for s while m will not, and this will cause a violation of the availability policy *r(s)*=2 specified for s.

4.3.5 How to achieve agreement on global states?

Random communication delays and performance failures can cause messages broadcast for updating state replicas to be lost or received out of order. To ensure that all correct servers agree on the same sequence of global state updates, one has to solve two major problems. First, achieve agreement on a sequence of processor joins and failures, so that at each point in time, each correctly functioning processor knows the group of other correct processors with which communication is possible. Second, one has to achieve agreement on the order of messages broadcast by processors in a group. A protocol that ensures agreement on a unique temporal sequence of successive processor group and group memberships $members_1$, $members_2$, ..., $members_i$, ... is termed a *processor membership protocol* [C88]. A protocol that ensures that all broadcasts originated by processors in a group $members_i$ are received in the same order by all correct processors in $members_i$ is termed an *atomic broadcast protocol* [CASD85].

The requirements for both membership and atomic broadcast consist of a number of safety and timeliness properties. *Safety* properties are invariants which must be true at all points in real time. For example, a membership protocol must always ensure that any two processors joined to the same group agree on the group membership, and an atomic broadcast protocol must ensure that correct processors in a same group always agree on the order of broadcast messages. *Timeliness* properties require that there be upper bounds on the time it takes to propagate information among group members. For example, a membership protocol might require that surviving members detect any member failure within a bounded time, and an atomic broadcast protocol might require that all correct members of a group learn about any atomic broadcast within a bounded time.

According to whether they assume or not the existence of a bound d on message delays, existing membership and atomic broadcast protocols can be divided in *synchronous* protocols, such as [C88], [CASD85], and [Cri89], and *asynchronous* protocols, such as [BJ87], [Ca85] and [CM84]. In general, synchronous protocols assume that processor clocks are synchronized within some constant maximum deviation *epsilon*, while asynchronous protocols do not require clocks to be synchronized.

Synchronous protocols have strong timeliness properties: they guarantee that information propagates within bounded times among group members even when failures and member joins occur concurrently with the propagation. For known synchronous protocols, the propagation speed depends on d and ϵ. The assumption that actual message delays are shorter than d is crucial for a synchronous approach to work: if the delays experienced by protocol messages can exceed d, that is, the servers which communicate through these messages can be *transiently partitioned*, a synchronous protocol might violate its safety requirements. For example, after transient partition occurrences, servers might disagree on the order of messages broadcast and on group membership. Protocols for detecting transient partitions and reconciling diverging server views have been investigated in [SSCA87]. Such a posteriori partition detection and recovery protocols need to compensate for any damage done while disagreement existed among group members.

In contrast with synchronous protocols, one can build asynchronous protocols that never violate their safety requirements, even when communication delays are unbounded and communication partitions occur. Such strong safety guarantees come however with a price. First, the timeliness properties of asynchronous protocols are rather weak: they do not guarantee any a priori known upper bound on the amount of time that can elapse between the instant when one processor learns of an event and another processor learns of the same event. For example, if member joins and failures continue to occur, an asynchronous membership protocol might need an unbounded amount of time to detect changes in the set of correctly working processors. Similarly, a message that was atomically broadcast to a group might need an unbounded amount of time to reach some group members. Second, most asynchronous protocols make the activities of membership computation and broadcast interfere, so that broadcasts that occur during a membership change have to be aborted [BJ87], [Ca85], [CM84]. Third, the asynchronous protocols specifically designed to tolerate partitions require that the correctly working processors of a system form a quorum before any work can be done [CM84], [KLS86]. This requirement, needed to prevent divergence among processors in distinct partitions, affects adversely system availability, since a quorum of processors needs to be functioning before any work can be done. If no quorum of correct processors is present, one possibility is to ask the human operator for permission to go ahead [KLS86]. Another possibility is to attempt to modify quorums dynamically [BGMS89].

Designers of distributed fault-tolerant systems are thus faced with the following choices: attempt to ensure the existence of an upper bound d on message delays or accept unbounded message delays. The first alternative requires advance knowledge of the maximum system load, real time operating systems, and massive communication hardware redundancy to ensure a negligible probability that network partitions occur. If an upper bound d is achievable and this bound allows a satisfactory recovery speed from events such as processor and communication failures, then one can adopt a synchronous approach which guarantees strong timeliness properties and enables the system to continue to work autonomously for as long as there exists at least one correct processor. If the bound d which is achievable by taking into account the specified peak load and real-time operating system scheduling algorithms is not small enough to allow the system to react sufficiently fast to component failures (for example, d is measured in minutes when the required recovery time from failures is measured in seconds) or if no bound can be realistically found, then two options exist. Both are based on the adoption of a *timeout delay* d' (When an unsatisfactory bound d exists, the timeout delay d' is by definition smaller than d.) If d' is such that a synchronous approach based on it provides sufficient recovery speed and the cost of compensating for actions taken during transient partitions is smaller than the cost of not meeting recovery deadlines, then one can adopt a synchronous approach based on d'. This will ensure strong timeliness properties and will eliminate the need to worry about quorums. Moreover, if d' is sufficiently large compared to the median message delay, it is likely that few messages delays will exceed d' time units [Cr89], and transient partitions will therefore be rare. On the other hand, if the cost of compensating for inconsistent actions taken as a consequence of transient partition occurrences is higher than the cost of missing recovery deadlines, one can adopt an asynchronous approach with timeout delay d'. The cost will then be weak timeliness properties and the need to worry about quorums. In both timeout-based approaches, the choice of the timeout delay d' is a delicate matter. As d' becomes larger, the failure detection and recovery times become larger, but the frequency of transient partition detections decreases. Conversely, as d' becomes smaller, the failure detection and recovery times become smaller, but the frequency of transient partition detections increases. Thus, the choice of d' must balance the cost of recovering from transient partitions and the cost of not detecting permanent failures fast enough.

5 Conclusion

Understanding a technical area as complex as fault-tolerant distributed computing requires identifying its fundamental concepts and naming them unambiguously. This paper has proposed a small number of concepts that we believe are fundamental when designing and understanding fault-tolerant distributed systems. Some of these concepts, such as the notions of service, server, and the "depends" relation, are fundamental to any kind of system. Others, such as the notions of failure semantics, hierarchical failure masking, and group failure masking, are specific to fault-tolerant systems. The latter concepts capture the goals of fault-tolerant computing as well as the trade-offs: mask component failures when possible, and when masking is not possible or economical, ensure that the system has some clearly specified failure semantics. To demonstrate the "adequacy" of these concepts in capturing essential architectural aspects, we used them to 1) formulate a list of key hardware and software issues that arise in fault-tolerant distributed systems design, 2) describe various design choices and alternatives known for each issue, 3) comment on the relative merits and drawbacks of these alternatives, and 4) illustrate how the issues have been addressed in existing practical system architectures. The fact that all four objectives could be achieved solely by using the basic notions introduced in Section 2 indicates their "power" to express crucial architectural issues in fault-tolerant distributed systems.

Clear concepts and terminology help but do not entirely solve the problem of how to design a fault-tolerant distributed system that uses the right amount of redundancy at various abstraction layers to achieve some optimum function/dependability/cost result. Very little of a "complexity theory" of fault-tolerant computing exists today that would guide a designer in choosing among a multitude of possible redundancy management solutions at hardware, operating system and application levels. Such choices are not only made difficult by the lack of analytical or experimental cost information about various redundancy management techniques, but also by the lack of published data about what kind of failure behaviors various system components are likely to exhibit and what failure distributions are associated with such components. The stochastic aspects inherent in fault-tolerance, not discussed in much detail in this paper, add another dimension of complexity to design. Moreover, even if one had available all the needed theories and experimental cost and failure data, the number of choices to consider would be so immense that a systematic search for optimality is unlikely to happen. For these and other reasons, it is likely that building distributed fault-tolerant systems will remain an art in the foreseeable future. One thing seems to be certain: with the ever increasing dependence placed on computing systems, the availability of computing and communication services in the presence of component failures, hardware and software changes, and horizontal growth will become more and more important. To achieve such high levels of availability, more systems in the future will need to be fault-tolerant.

Acknowledgements

The motivation for recording this snapshot of my understanding of basic concepts and issues in fault-tolerant distributed systems was provided by invitations to give overviews on fault-tolerant computing at two recent conferences: the 1987 Annual Conf. of the Canadian Information Processing Society, Edmonton and the 3d Brazilian Conf. on Fault-Tolerant Computing, Rio de Janeiro, 1989. I would like to thank the program committees of these conferences, in particular Professors Tamer Ozsu and Julius Leite, for inviting me. I would also like to thank Phil Bernstein and the CACM reviewers for their helpful comments and suggestions.

REFERENCES

[AL81] T. Anderson, P. Lee: Fault-tolerance- Principles and Practice, Prentice Hall, 1981.

[A89] A. Avizienis: Software Fault Tolerance, IFIP Computer Congress, San Francisco, August 1989.

[Aa85] A. Avizienis, P. Gunningberg, J. Kelly, L. Strigini, P. Traverse, K. Tso, U. Voges: The UCLA Dedix System: a Distributed Testbed for Multi-version Software, 15th Int. Conf. on Fault-tolerant Computing, Ann Arbor, Mi 1985.

[ASC85] A. E. Abbadi, D. Skeen, F. Cristian: An Efficient Fault-Tolerant Protocol for

Replicated Data Management, 4th ACM Conf. on Principles of Database Systems, 1985.

[B81] J. Bartlett: A NonStop Kernel, 8th Symp. on Operating System Principles, Dec. 1981.

[B88] P. Bernstein: Sequoia: a Fault-tolerant Tightly Coupled Multiprocessor for Transaction Processing, IEEE Computer, February 1988.

[Ba89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, W. Oberle: Fault-Tolerance under Unix, ACM Trans. on Computer Systems, Vol. 7, No. 1, Feb 1989.

[BD85] O. Babaoglu, R. Drumond: Streets of Byzantium: Network Architectures for Fast Reliable Broadcast, IEEE Tr. on Software Engineering, Vol. SE-11, No. 6, 1985.

[BGMS89] D. Barbara, H. Garcia-Molina, A. Spauster: Increasing Availability under Mutual Exclusion Constraints with Dynamic Vote Reassignment, ACM Trans. on Computer Systems, Vol. 7, No. 4, Nov 1989.

[BHG87] P. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

[BJ87] K. Birman, T. Joseph: Reliable Communication in the Presence of Failures, ACM Trans. on Computer Systems, Vol. 5, No. 1, February 1987.

[C85] F. Cristian: A Rigorous Approach to Fault-tolerant Programming, IEEE Tr. on Software Eng., Vol. SE 11, No. 1, 1985.

[C88] F. Cristian: Agreeing on Who is Present and Who is Absent in a Synchronous Distributed System, 18th Int Conf on Fault-Tolerant Computing, Tokyo, June 1988.

[C89] F. Cristian: Exception Handling, in "Dependability of Resilient Computers", T. Anderson, Ed., Blackwell Scientific Publications, Oxford, 1989.

[Ca85] R. Carr: The Tandem Global Update Protocol, Tandem Systems Review, Vol. 1, No. 2, June 1985.

[CASD85] F. Cristian, H. Aghili, R. Strong, D. Dolev: Atomic Broadcast: From Simple Diffusion to Byzantine Agreement, 15th Int. Conf. on Fault-tolerant Computing, Ann Arbor, Mi 1985.

[CDD90] F. Cristian, R. Dancey, J. Dehn: Fault-tolerance in the Advanced Automation System, 20th Int. Conf. on Fault-tolerant Computing, Newcastle upon Tyne, England, June 1990.

[Cl85] D. Clark: The Structuring of Systems using Up-calls, 10th ACM Symp. on Operating Systems Principles, 1985.

[CM84] J.M. Chang, N. Maxemchuck: Reliable Broadcast Protocols, ACM Tr. on Computer Systems, Vol. 2, No. 3, August 1984.

[Co85] E. Cooper: Replicated Distributed Programs, PhD thesis, UC Berkeley, 1985.

[CP85] D. Comer, L. Peterson: Understanding Naming in Distributed Systems, Distributed Computing, Vol. 3, pp. 51-60, 1989.

[Cr89] F. Cristian: Probabilistic Clock Synchronization, Distributed Computing, Vol. 3, pp. 146-158, 1989.

[Cri89] F. Cristian: Synchronous Atomic Broadcast for Redundant Broadcast Channels, IBM Research Report RJ 7203, Dec 1989.

[CZ85] D. Cheriton, W. Zwaenepoel: Distributed Process Groups in the V Kernel, ACM Tr. on Comp. Systems, Vol. 3, No. 2, May 1985.

[D71] E. Dijkstra: Hierarchical Ordering of Sequential Processes, Acta Informatica, Vol 1, pp. 115-138, 1971.

[ES86] P. Ezhilchelvan, S. Shrivastava: A Characterization of Faults in Systems, 5th Symp. on Reliability in Dist. Softw. and Database systems, Los Angeles, January 1986.

[G78] J. Gray: Notes on Database Operating Systems, Operating Systems - An Advanced Course, Lecture Notes in Computer Science, Springer Verlag, Vol 60, 1978.

[G86] J. Gray: Why do computers stop and what can be done about it? 5th Symp. on Reliability in Dist. Softw. and Database systems, Los Angeles, January 1986.

[GS89] H. Garcia-Molina, A. Spauster: Message Ordering in a Multicast Environment, 9th Int. Conf. on Distributed Systems, Newport Beach, California, June 1989.

[Ha78] A. Hopkins, B. Smith, J. Lala: FTMP-A highly reliable fault-tolerant multiprocessor for aircraft, Proceedings IEEE, Vol. 66, Oct 1978.

[HLD88] R. Harper, J. Lala, J. Deyst: Fault Tolerant Parallel Processor Architecture Overview, 18th Int Conf on Fault-Tolerant Computing, Tokyo, June 1988.

[IBM87] IBM International Technical Support Centers: IMS/VS Extended Recovery Facility (XRF): Technical Reference, 1987.

[JZ87] D. Johnson, W. Zwaenepoel: Sender Based Message Logging, 17th Int Conf on Fault-Tolerant Computing, Tokyo, June 1987.

[KA89] J. Knight, P. Amann: Issues Influencing the Use of N-version Programming, Proceedings IFIP Congress, San Francisco, August 1989.

[KT90] F. Kaashoek, A. Tanenbaum: Fault-tolerance Using Group Communication, 4th ACM SIGOPS European Workshop, Bologna, Sept. 1990.

[KGR89] H. Kopetz, G. Grunsteidl, J. Reisinger: Fault-tolerant Membership in a Synchronous Real-time System, IFIP Working Conference on "Dependable Computing for Critical Applications", Santa Barbara, August 1989.

[KLS86] N. Kronenberg, H. Levy, W. Strecker: VAXclusters: A Closely-Coupled Distributed System, ACM Transactions on Computer Systems, Vol. 4, No. 2, 1986.

[KT87] R. Koo, S. Toueg: Check-pointing and Rollback Recovery for Distributed Systems, IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, 1986.

[L84] L. Lamport: Using Time Instead of Time-outs in Fault-Tolerant Systems, ACM Trans on Programming Languages and Systems, vol. 6, no. 2, 1984.

[L89] L. Lamport: The Part Time Parliament, DEC SRC Report 49, Sept 1989.

[La89] J. C. Laprie: Dependability: A unifying concept for reliable computing and fault-tolerance, T. Anderson, Ed., Blackwell Scientific Publications, Oxford, 1989.

[La90] J. C. Laprie, J. Arlat, C. Beounes, K. Kanoun: Definition and Analysis of Hardware and Software-Fault-Tolerant Architectures, IEEE Computer, July 1990.

[LG90] S. Luan, V. Gligor: A Fault-tolerant Protocol for Atomic Broadcast, 10th Int Conf on Distributed Computing Systems, Paris, May 1990.

[LL89] G. Le Lann: Critical Issues in Distributed Real-Time Computing, Proceedings of "ESTEC Workshop on Communication Networks and Distributed Operating Systems within the Space Environment", European Space Agency Report WPP-10, Noordwijk, Oct.24-26, 1989.

[LLS90] R. Ladin, B. Liskov, L. Shrira: Lazy Replication: a Method for Managing Replicated Data, 9th Annual ACM Symposium on Principles of Distributed Computing, August 1990.

[LS81] B. Lampson, H. Sturgis: Atomic Transactions, in "Distributed Systems: An Advanced Course", Lecture Notes in Computer Science Vol. 105, Springer Verlag, 1981.

[M82] E. McCluskey: Fault-Tolerant Systems, Technical Report CSL-199 Stanford University, 1982.

[MMA90] M. Melliar-Smith, L. Moser, V. Agrawala: Broadcast Protocols for Distributed Systems, IEEE Tr on Parallel and Distributed Systems, Vol. 1, No. 1, Jan 1990.

[OL88] B. Oki, B. Liskov: Viewstamped Replication: a New Primary Copy Method to Support Highly Available Distributed Systems, 7th ACM Symp. on Principles of Distributed Computing, August 1988.

[P79] D. Parnas: Designing Software for Ease of Extension and Contraction, IEEE Tr. on Software Engineering, Vol. SE-5, No. 2, March 1979.

[Pa89] D. Powell: La Tolerance aux Fautes Dans les Systemes Repartis: Les Hypotheses d'Erreur et leur Importance, LAAS Research report 89-258, September, 1989.

[PB85] D. Palumbo, R. Butler: Measurement of SIFT operating system overhead, NASA Technical Memo 86322, 1985.

[PW72] W. Peterson, E. Weldon: Error Correcting Codes, MIT Press, Massachusetts, 1972.

[R75] B. Randell: System Structure for Software Fault-Tolerance, IEEE Trans. on Software Eng., Vol. SE-1, No. 2, 1975.

[S90] D. Siewiorek: Fault-tolerance in Commercial Computers, IEEE Computer, July 1990.

[S86] F. Schneider: The State Machine Approach: a tutorial, TR 86-800 Cornell Univ., 1986.

[S88] F. Schmuck: The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems, PhD thesis, TR88-928 Cornell Univ., 1988.

[SRC84] J. Saltzer, D. Reed, D. Clark: End-to-end Arguments in System Design, ACM Trans. on Computer Systems, Vol. 2, No. 4, Nov, 1984.

[SSCA87] R. Strong, D. Skeen, F. Cristian, H. Aghili: Handshake Protocols, 7th Int. Conf. on Distributed Computing Systems, Berlin, September 1987.

[SY85] R. Strom, S. Yemini: Optimistic Recovery in Distributed Systems, ACM Transactions on Computer Systems, Vol. 3, No. 3, 1985.

[T81] A. Tanenbaum: Computer Networks, Prentice Hall, Englewood Cliffs, NJ, 1981.

[T82] K. Trivedi: Probability and Statistics with Reliability, Queuing and Computer Science Applications, Prentice Hall, 1982.

[TW89] D. Taylor and G. Wilson: The Stratus System Architecture, in "Dependability of Resilient Computers", T. Anderson, Ed., Blackwell Scientific Publications, Oxford, 1989.

[VRB89] P. Verissimo, L. Rodrigues, M. Baptista: AMp: A Highly Parallel Atomic Multicast Protocol, in Proceedings, ACM SIGCOM'89, Austin, Texas, Sept 89.

[W78] J. Wakerly: Error Detecting Codes, Self-checking Circuits, and Applications, Elsevier North-Holland, Inc., New York, 1978.

[Wa78] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, M. Melliar-Smith, R. Shostak, C. Weinstock: SIFT: Design and Analysis of a Fault tolerant Computer for Aircraft Control, Proc IEEE Vol. 66, Oct 1978.

[W75] W. Wulf: Reliable Hardware-Software Architecture, 1975 Int. Conf. on Reliable Software, SIGPLAN 10, No. 6, 1975.

General Terms: Design, Reliability

Additional Key Words and Phrases: Exception Handling, Failure, Failure Classification, Failure Masking, Failure Semantics, Fault-Tolerant System, Group Communication, Redundancy, Server Group, System Architecture.