Abstractions for Constructing Dependable Distributed Systems

Shivakant Mishra[1] and Richard D. Schlichting

TR 92-19

Abstract

Distributed systems, in which multiple machines are connected by a communications network, are often used to build highly dependable computing systems. However, constructing the software required to realize such dependability is a difficult task since it requires the programmer to build fault-tolerant software that can continue to function despite failures. To simplify this process, canonical structuring techniques or programming paradigms have been developed, including the object/action model, the primary/backup approach, the state machine approach, and conversations. In this paper, some of the system abstractions designed to support these paradigms are described. These abstractions, which are termed fault-tolerant services, can be categorized into two types. One type provides functionality similar to standard hardware or operating system services, but with improved semantics when failures occur; these include stable storage, atomic actions, resilient processes, and certain kinds of remote procedure call. The other type provides consistent information to all processors in a distributed system; these include common global time, group-oriented multicast, and membership services. In addition to describing the fundamental properties of these abstractions and their implementation techniques, a hierarchy highlighting common dependencies between services is presented. Finally, a number of systems that use these abstractions are overviewed, including the Advanced Automation System (AAS), Argus, Consul, Delta-4, ISIS, and MARS.

August 3, 1992

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]Current address: Dept. of Computer Science and Engineering, University of California San Diego, La Jolla, CA 92093, USA

1 Introduction

The need for computer system *dependability*, defined as the basic trustworthiness of a computer system that allows people to rely on the service it delivers [Lap92], has been steadily increasing. Not only are computers becoming more pervasive, they are also being used in critical applications where failures resulting in deviation from specified service can have disastrous consequences. For example, air traffic control, banking, and nuclear reactor control are all applications that fit into this category. Moreover, there are many, well-documented instances where problems in hardware and/or software have in fact caused system failures resulting in loss of life or substantial economic disruptions [Neu91].

Distributed systems, where a collection of processors are connected by a network with no physically shared memory, are often used as a basis for providing highly dependable computing services. One reason for this is that many critical applications are process-control situations in which the components being controlled—and hence the controlling computers—are physically dispersed. For example, automated factories and nuclear reactors often fit this description. In cases such as these, the particular architecture is fixed by the demands of the application and is not a variable in the design process.

Another reason for the use of distributed systems is that such an architecture provides a natural framework for using *fault-tolerance* techniques to enhance system dependability. These techniques, which are used to construct a system that can continue to function despite the failure of internal components, are all based on using redundancy of some form to mask and/or detect failures. The multiple processors, memories, and secondary storage units typically found in a distributed system inherently provide redundancy that can be used for fault-tolerance purposes. Thus, distributed systems are a good basis for dependable computing even when not dictated directly by the characteristics of the specific application.

A distributed architecture by itself is, of course, only a starting point: it rests with the software to actually translate this potential into a dependable computing system. To accomplish this, the software must be constructed as *fault-tolerant software*, that is, software that can continue to provide service despite (some number and type of) failures. Unfortunately, there are a large number of complicating factors that must be taken into account when writing this type of software. These include complexities caused by the distributed nature of the software (e.g., concurrency, network delays), as well as the arbitrary and asynchronous nature of hardware and software failures themselves. Moreover, many of the applications requiring dependable computing have real-time constraints as well, which further complicates the situation by adding yet another degree of freedom that must be taken into account. All of the these factors conspire to raise the complexity level of such software and to make its design, implementation, and validation a daunting task.

The problems associated with developing complex software have, of course, been recognized for years, and many different techniques have been proposed for rectifying the situation. Of these, one that has proved especially effective for constructing fault-tolerant software is the judicious use of *programming paradigms*, which reduce the complexity of the task by providing canonical software organization techniques and supporting abstractions for a given type of problem. Important paradigms that have been developed for fault-tolerant software include the *object/action paradigm* [Gra86], the *primary/backup approach* [AD76], the *state machine approach* [Sch90], and *conversations* [Ran75]. Fundamental abstractions that have been defined in conjunction with these paradigms include *stable storage* [Lam81], *atomic actions* [Lis85], *common global time* [Lam78], and *reliable multicast* [CM84]. These abstractions serve as a convenient base for realizing the various paradigms by defining operations with higher-level functionality or with semantics that are well-defined even when failures occur. For example, many of these abstractions can be thought of as more dependable variants of common hardware or operating system services. These paradigms and abstractions have been used in many systems oriented towards fault-tolerant distributed applications, such as the Advanced Automation System (AAS) [CDD90], Argus [LS83, Lis88], Consul [MPS91], Delta-4 [PSB+88], ISIS [BJ87, BSS91], and MARS [KDK+89].

In this paper, we overview these abstractions, which we refer to by the general term *fault-tolerant services*. Our primary goal in doing so is to identify useful services and describe their relevant properties. In the process, we also outline various implementation techniques for each service, paying special attention to their assumptions and limitations. Our secondary goal is also to describe the relationships between the different services, and to highlight the common ways in which services depend on each other. We do this by constructing a service hierarchy that illustrates how a given service can be implemented using the other services. This focus on surveying a wide range of fault-tolerance abstractions for distributed systems and explaining their interrelationships help distinguish this paper from other similar efforts (e.g., [BMD91, Cri91, Koh81, RLT78, Sch90].)

This paper is organized as follows. We begin by outlining our software and hardware system model in Section 2, with a special focus on identifying properties that affect the algorithms used in the implementation of the different fault-tolerant services. We also describe the four software structuring paradigms mentioned above in more detail. Section 3 through 8 then describe the fault-tolerant services; specifically, we consider services that implement a common global time in a distributed system, multicast communication, remote procedure call, membership, atomic actions, and resilient processes. The common dependencies between services are outlined in Section 9, while Section 10 overviews some of the fault-tolerant systems that have been designed and built using these abstractions. Finally, Section 11 summarizes the paper.

2 System Model

2.1 Overview

The hardware basis for a distributed system consists of a collection of processors connected by a communication network. Each processor has its own local memory, but there is typically no shared memory between processors. This property implies that the only means for processes executing on different machines to communicate is by message exchange. The actual configuration of the network can vary within these constraints, ranging from, for example, local-area broadcast networks like an Ethernet, to store-and-forward networks like those commonly used for wide-area communication.

The software found on such systems varies widely in its overall structure and organization, but can generally be divided into *application software* and *system software*. The fault-tolerant services that are the focus of this paper are part of the system software, so the organization of the software on a particular processor is as depicted in Figure 1. From the bottom, the software consists of the standard operating system services providing abstractions such as processes and virtual memory, a *fault-tolerance support layer* realizing the fault-tolerant services, and finally the application software. As is standard in such a level-structured organization, each layer uses the abstractions defined by the level below it to implement its own services. Section 9 further refines this organization for the fault-tolerance support layer by giving common dependencies between the various services.

The purpose of the fault-tolerance support layer is to implement abstractions—i.e., fault-tolerant services—that simplify the programming of distributed applications requiring resilience to failures. For many of these abstractions, the implementation requires that software components on two or more machines communicate and cooperate, so the most accurate way to view this support layer is as a single

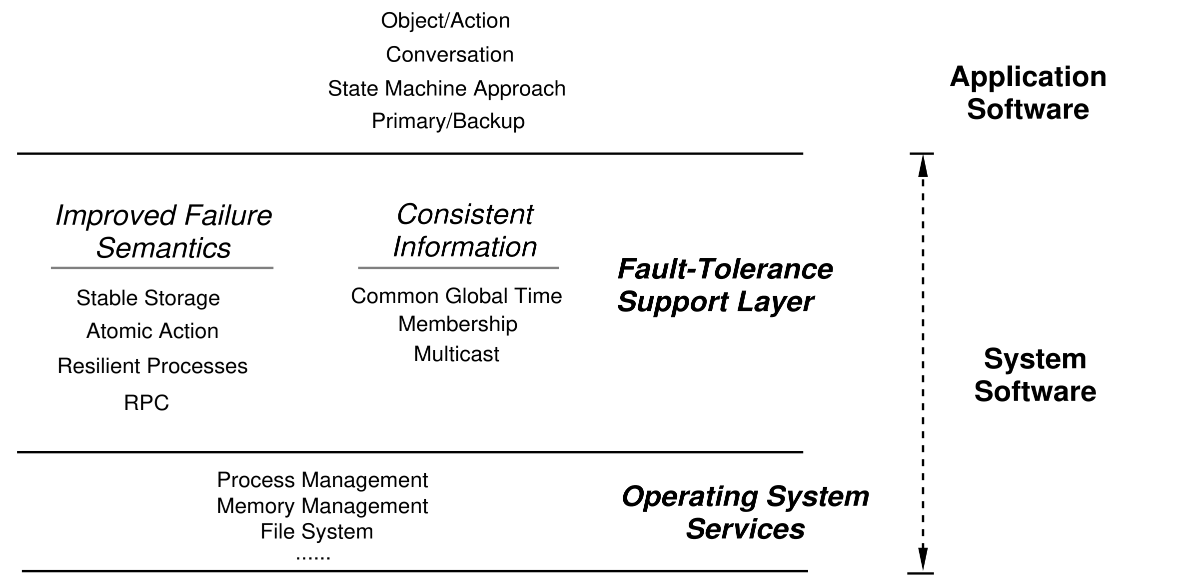

Figure 1: Overall system model

logical entity that spans multiple machines. *Protocols* are the set of rules that software components on different machines use to realize a given abstraction. We also use this term informally to refer to the actual software components implementing these rules, so each fault-tolerant service can be thought of as being implemented by one or more protocols on each machine. These protocols are typically identical on all machines.

The abstractions implemented by the fault-tolerance support layer can be classified into two general categories based on the kind of functionality they provide. The first contains those abstractions that are similar to features found in standard systems, but with improved failure semantics. Examples of such abstractions in Figure 1 include stable storage, atomic actions, resilient processes, and certain types of remote procedure call (RPC). Stable storage is data storage that suffers no failures itself and is not affected by the failure of other components [Lam81]; thus, stable storage is similar to standard memory or disk storage, but with better semantics in the face of failures. Atomic actions are sequences of instructions potentially spanning multiple machines that are guaranteed to either execute completely or not at all despite failures [Lis85]; again, this makes atomic actions similar to standard sequential execution sequences, but with better behavior when failures occur. Resilient processes are processes that can continue executing correctly even if interrupted by failure and then restarted; the similarity here is, of course, to regular processes, but with the ability to tolerate failures. Finally, RPC refers to a collection of interprocess communication protocols commonly used in distributed systems that attempt to provide semantics similar to procedure calls; while most of these protocols simply terminate a call abnormally when failures occurs, a few actually provide strong fault-tolerance guarantees.

The second category encompasses abstractions whose purpose is to provide consistent information to processes executing on different machines in a distributed system. Examples of such abstractions in Figure 1 include common global time, membership, and multicast. Common global time provides a consistent time base for all machines despite the lack of a single physical clock; this service is especially useful for consistently ordering events in a distributed system. Membership is a service that provides a consistent view of which processors are functioning and which have failed at any given moment in time.

Finally, multicast is a communication service that allows a message to be transmitted asynchronously to a group of processes rather than just a single process; properties often associated with multicast primitives designed for fault-tolerant systems include atomicity and various ordering properties, which ensure that messages are delivered to all processes in some sort of consistent order.

The collection of fault-tolerant services needed for a given application and the exact way in which they are implemented depend on a multitude of factors. Of these, three can be identified as especially important: the *programming paradigm* used for the application, the *failure model* assumed, and the *synchrony* of the system. The first refers to the way in which the application software is organized; as mentioned in the Introduction, several canonical structuring techniques have been identified, so we concentrate our attention on these. The failure model is the type of failure a component in the system is assumed to suffer; common failure models used for fault-tolerant distributed systems include *fail-stop* [SS83], *crash* [PSB$^+$88], and *Byzantine* [LSM82]. The synchrony of the system is related to assumptions made about the time bound on certain activities; components are usually assumed to be either *synchronous* or *asynchronous*. Given the importance of these factors, we now discuss each in more detail.

2.2 Programming Paradigms

As noted above, four common programming paradigms for fault-tolerant distributed software are the object/action model, the conversation model, the primary/backup approach, and the state machine approach. As its name implies, the primary components of the object/action model are objects and actions. Objects are passive entities that encapsulate a state and export certain operations to modify that state; typically, this state involves long-lived data that is assumed to be stored on stable storage to survive failures. Actions are active entities similar to threads that invoke operations on objects to carry out some task. The objects comprising an application can potentially be located on multiple machines in a network, which implies that actions may logically cross machine boundaries during their execution. An action has two properties that guarantee the atomicity of its execution with respect to both failures and the concurrent execution of other actions. First, it is *recoverable*, that is, it is either executed completely or not at all, despite failures; second, it is *serializable*, that is, the effect of executing multiple actions concurrently is equivalent to some serial schedule. Serializability has also been called indivisibility [Lis85], while recoverability has also been called totality [Wei89] and the unitary property [Lam81]. In the context of databases, atomic actions are usually called transactions [BHG87]. Abstractions that are useful for supporting the object/action model include stable storage, atomic actions, RPC, and resilient processes.

In the conversation model, processes and messages play a primary role. An application is structured as a collection of concurrent processes that communicate by exchanging messages. Processes periodically *checkpoint* their state onto stable storage so that they can recover and continue executing following failures. Conversations are a structuring technique for coordinating checkpoints among processes to guarantee that the checkpoints represent a consistent global state or, equivalently, form a *recovery line* [RLT78]. This avoids the *domino effect*, in which a single failure can force the rollback of multiple processes to successively earlier checkpoints to find a consistent state. In addition to checkpointing facilities, some sort of reliable interprocess communication is also required to implement this model. The object/action model and conversation model have been shown to be duals of one another [SMR88]. Abstractions that are useful for supporting conversations include stable storage and resilient processes.

An application following the primary/backup approach is organized as a collection of services, each of which is implemented by multiple processes to provide fault-tolerance. The name comes from the notion that only one of the processes for a given service is active at any time; this process is called the *primary*, and all requests for service are routed to that process. The remainder of the processes, which are referred to as *backups*, do not respond to requests unless a failure occurs that prevents the primary from providing service. At this point, one of the backups becomes the primary, typically starting in a state that was checkpointed by the primary onto stable storage prior to failure; alternatively, this state could be actively propagated to the backups during execution rather than checkpointed. This approach is also sometimes called *passive replication*. Abstractions useful for supporting the primary/backup approach include stable storage, multicast, and membership.

Like primary/backup, in the state machine approach, an application is structured as a collection of services that are implemented by multiple processes for fault-tolerance. Here, each service is characterized as a state machine, which maintains *state variables* that are modified in response to *commands* that are received from other state machines or the environment. Execution of a command is deterministic and atomic with respect to other commands. The output of a state machine, that is, the sequence of commands to other state machines or the environment, is completely determined by the sequence of commands input for execution by the state machine. The fundamental difference from primary/backup is that the fault-tolerant version of a state machine is implemented by replicating that state machine and running each replica in parallel on a different processor in a distributed system. This approach is sometimes called *active replication*. Issues that must be addressed in the state machine approach include maintaining replica consistency at all times and integrating repaired replicas following failure. Abstractions that are useful for supporting replicated state machines include common global time, multicast, membership, RPC, and, if reintegration of replicas back onto the computation is desired, stable storage and resilient processes.

2.3 Failure Models

When a specification of a component's acceptable behavior is available, it provides a standard against which the behavior of that component can be judged. The specification may prescribe both the component's response for any initial state and input sequence, and the real-time interval within which the response should occur. A component is *correct* if, in response to inputs, it behaves in a manner consistent with the specification; if it behaves otherwise, it has *failed* [Cri91].

A *failure model* is a way for precisely specifying assumptions about how a component behaves when it fails. A number of such failure models have been defined; although we state these in terms of generic components, they are most often applied to processors. In the *fail-stop* failure model, it is assumed that the component fails by ceasing execution without undergoing any incorrect state transition and that this failure is detectable by other components [SS83]. In a *crash* model, a component is assumed to fail in the same way, but without the guarantee of detectability. This model has also been termed *fail-silent* [PSB$^+$88]. The *omission failure* model assumes that a component fails by not responding to some input [CASD85]. Under the assumption that a component remains inactive following a crash failure, then failures in this class are a special case of omission failures in which a component never responds to inputs following its first omission. The *timing* failure model assumes that a component fails by giving an untimely response; that is, the response is functionally correct but occurs outside the required real-time interval [CASD85]. The timing failure can be *early* timing failure or a *late* timing failure; late timing failures are also sometimes called *performance* failures. A failure is classified as *arbitrary* or *Byzantine* if the component's failure behavior is completely unspecified [LSM82]. A component assumed to fail in this make may unknown, inconsistent, or even malicious actions. The inclusion relationship among these models is illustrated in Figure 2.

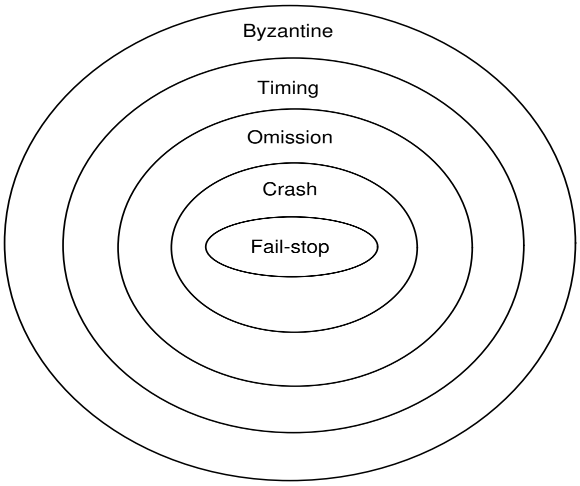

Figure 2: Failure model hierarchy

2.4 Synchrony

Synchrony refers to assumptions that are made about the execution bounds on components. A hardware or software component is *synchronous* if it always performs its intended function within a finite and known time bound, and *asynchronous* otherwise. This bound on the execution time of the synchronous component must hold whenever the component is correctly operating, and in particular, under all operational conditions within its specification. Synchrony can be defined for communication channels, communication networks, processors, and protocols. For example, in a *synchronous communication channel*, the transmission delay of a unit of data across the link is known and bounded. Similarly, a *synchronous processor* is a processor in which the time to execute a unit of work is known and bounded.

These definitions of synchronous components extend to groups of components as well. For example, a *synchronous network* is one in which the time required to transmit a unit of data from any machine to another is known and bounded. This will be the case either if all of the channels are synchronous, or if a time limit is placed on transmission such that any data arriving after that limit has expired is considered an indication of failure on the part of the network or sending machine. Similarly, a distributed system is considered synchronous if it contains a synchronous network and synchronous processors, and asynchronous otherwise. Examples of synchronous systems used in the literature include [CDD90, KDK+89]; a completely asynchronous system is assumed in [FLP85]. Some systems in the literature [BJ87, CM84] have been described as asynchronous, but actually employ certain mechanisms that have the effect of making them synchronous. Examples of such restrictions include assumptions about failure detection or, as noted above, bounds on message transmission time. In fact, strong impossibility results related to cooperation among processes have been proved for truly asynchronous systems [FLP85], so any realistic system is actually based on assumptions of synchrony.

The notion of synchrony can be applied to software components as well. A protocol is defined to be synchronous if the time to perform the sequence of events required by the protocol is known and bounded. Two different approaches have been taken to designing such protocols: *clock-driven* and *clockless* [Ver90]. In the clock-driven approach, the protocol relies on a common global time base constructed using the clock synchronization techniques described in Section 3. Most of these protocols are periodic in nature, taking certain actions based on the time read from the common clock; synchronous processors and communication are typically assumed by these protocols. In contrast, clockless protocols do not rely on the existence of such synchronized clocks, nor do they explicitly assume the existence of synchronous processors or communication. The result is a radically different system style and organization, where the protocols themselves approximate processor and communication synchrony by using techniques such as timeouts and acknowledgements to put a bound on delays.

3 Common Global Time

In a standard distributed architecture, each processor has its own clock, but there is no global physical clock that can be accessed by all processors. This lack of a global time base has important negative implications, perhaps the most important of which relates to determining the *causal relation* among various events on different processors. This causality relation holds between two events a and b on the same or different processors if the execution of a could possibly have affected execution of b. For example, two consecutive statements in the same process are causally related, as are the send of a message in one process and the receive of the same message in the destination process. The relation is extended using transitive closure.

Normally, one would think to use the local clock time to determine the causal relation between two events a and b. That is, if the time at which a occurred is less than the time at which b occurred, then a and b would be defined to be causally related. However, given that local clocks can drift relative to one another at a variable and unpredictable rate, this may not hold if a and b are events on different processors. For example, if would be possible to conclude that the send of a message occurred *after* its receipt if the clocks were skewed in the right way, a violation of causality. This type of clock drift also makes it difficult to determine the real time at which an event occurred.

A fault-tolerant time service in a distributed system addresses these problems by providing the abstraction of a common global time despite failures. Since this service provides functionality similar to a single shared clock, it facilitates the construction of event orderings that are consistent with the causality relation. This property is useful for, among other things, ensuring that messages multicast among a group of processes are received in the same order by all processes and in an order that reflects causality. This, in turn, provides the kind of ordering that is needed to preserve replica consistency in the state machine approach to constructing fault-tolerant programs.

There are two basic approaches to implementing a decentralized fault-tolerant time service in a distributed system. In the first approach, local processor clocks are synchronized at regular intervals in such a way that the clocks remain within some maximum distance of each other. The time of an event at a process executing in processor P is then defined to be the value of P's clock at the time the event occurs. The timing of events at different processors may be compared by allowing for the maximum difference by which the local clocks may differ before they are synchronized.

The second approach derives the temporal order in which different events occur in the system without direct association to a hardware clock value. To do this, a logical clock is constructed that causally orders different events of the system. For any two events, say *a* and *b*, exactly one of the following three relationships holds: event *a* occurred before event *b*, event *a* occurred after event *b*, or events *a* and *b* occurred at the same logical time. The logical clock is constructed to assign values to these events in such a way that these relationships are preserved.

Each of these two approaches has advantages and disadvantages. For example, logical clocks do not provide a mapping from the timing of an event to real time, whereas synchronized clocks may provide this mapping by synchronizing with an external time source. On the other hand, logical clocks preserve causality among different events depending on what events have been seen by the processes when an event occurs. In particular, two events happen at the same time at different processes if neither process is aware of the other event. A synchronized clock coerces an order that depends on their local times, thereby leading to loss of information about causality.

In the following, we discuss the details of how a common global time base can be constructed in a distributed system using each of these approaches.

3.1 Synchronizing Clocks

As mentioned, clock synchronization involves periodically adjusting the values of local clocks to prevent them from drifting too far apart. There are two basic variants on clock synchronization. In the first, termed *internal clock synchronization*, the processor clocks are always kept within a certain maximum drift of one another. In the second, termed *external clock synchronization*, the processor clocks are always kept within certain maximum deviation from an external time reference. By definition, externally synchronized clocks are also internally synchronized, while internally synchronized clocks may deviate arbitrarily from the external time reference. In the following, we introduce the salient features and algorithms of clock synchronization; $C_p(t)$ is used to denote the local clock time at processor p at real time t. Our discussion concentrates primarily on internal clock synchronization since the additional mechanism required to synchronize to an external time source is usually straightforward.

Properties of synchronized clocks

The following three correctness criteria can be used to assess clock synchronization schemes.

- *Monotonicity*: The clock is a monotonically increasing counter, that is

  $$C_i(t + \tau) \geq C_i(t) \qquad\qquad \tau \geq 0$$

  In general, since a clock increases by discrete values, it is possible that in a small real-time interval, τ, $C_i(t + \tau) = C_i(t)$. However, the granularity of most clocks is very small and for all practical reasons it is correct to assume that $C_i(t)$ is a strictly increasing function of t.

- *Precision*: The synchronized clocks are always within some maximum deviation of each other. That is,

  $$\mid C_i(t) - C_j(t) \mid < \beta$$

  where β is the specified synchronization precision.

- *Interval Preservation*: Also known as the *linear envelope*, this property states that any interval measured by the synchronized clocks is within some linear function of the real time interval:

  $$(1 - \rho)\tau \leq C_i(t + \tau) - C_i(t) \leq (1 + \rho)\tau$$

  Here, ρ is called the maximum clock drift rate.

While β specifies the maximum allowed distance between any two clocks, ρ specifies the maximum allowed drift of a clock from real time. Thus, ρ and β together specify the interval in which the local clocks must resynchronize. Multiple local clocks that are synchronized so as to satisfy the above three properties can be thought of as a common global clock $\hat{C}$ that has the following property for all pairs of i and j

$$(1 - R) \;\leq\; \frac{\hat{C}_i(t+\tau) - \hat{C}_j(t)}{\tau} \;\leq\; (1 + R)$$

Here, $\hat{C}_p(t)$ denotes the value of the common global clock at process p, and R is the maximum allowed drift between any two synchronized clocks per unit time. This second value is called the *drift rate* of the synchronized clock.

A synchronized clock may be used to measure intervals and to order various events in the system. The most common way to measure time intervals is by using a function *get_time_elapsed(t : time)* that returns the time elapsed since the clock showed time t. This function typically compensates for the changes in the clock value due to synchronization. In another approach [HSSD84], the notion of a clock is not bound to specific hardware and a processor may possess any number of clocks. In particular, every instance of clock synchronization logically gives rise to a new version of the clock. Here, the version of the clock used to time an event is the most recent version at the time the event occurred. Various events in the system can then be ordered using the local clock time of the processor at which they occur.

Complexities of clock synchronization

One of the basic functions needed to synchronize clocks is the ability to read the value of a remote clock. This is done either through the exchange of messages using some underlying communication network or through special hardware that generates clock signals and propagates them to other processors. In either case, there is a random propagation delay introduced before a process receives the message or signal. Thus, the time it takes to read a local clock or to set a local clock is not deterministic. This variation, along with the variable processing time for various messages received, introduces a random processing delay in the process of clock synchronization. The random propagation delays and the random processing delays limit the extent to which the clocks may be synchronized. The need to consider failures also complicates the algorithms, especially when failures are assumed to be Byzantine.

A few results are known that put a limit on the closeness with which clocks may be synchronized. In [LWL88], the authors show that n clocks cannot be synchronized with certainty closer than $(1 - 1/n)(max - min)$, even in the absence of any failures; here, max and min represent the maximum and minimum delay in message communication. Another result states that N clocks cannot be synchronized in the presence of more than $N/3$ Byzantine failures if *authentication* is not used, that is, if it is not possible to determine reliably who sent a given message. However, clocks may be synchronized in presence of any number of Byzantine failures given authentication [DHS86]. Optimal algorithms for clock synchronization under different failure scenarios are also known [ST87].

Algorithms

The problem of clock synchronization has been studied extensively, and a large number of algorithms proposed [Cri89, HSSD84, KO87, LMS85, LWL88, ST87]. A survey of some of these algorithms appears in [RSB90]. These algorithms differ from each other in their assumptions about the clocks and the network topology, as well as their failure hypothesis. The mechanics of clock synchronization involves periodically exchanging information about local clock values and then computing a correction factor and applying it to the local clock.

Both hardware and software approaches have been taken to address the problem of clock synchronization. In the former, special hardware is used to propagate signals between network nodes and to calculate correction values, as opposed to the use of explicit messages and software in the latter. Software approaches tend to be more flexible but suffer from larger clock skews, since the lower bound on clock skews in such cases is the difference between the minimum and the maximum message transit times. The hardware approaches provide a smaller clock skew, but are expensive and inflexible. Some of the algorithms use a combination of hardware and software, where the smaller skew of the hardware algorithms is sacrificed for a lower software cost [RSB90].

Most of the software approaches use a convergence function that guarantees the properties of monotonicity, precision and interval preservation. One class of algorithms consists of first exchanging local clock values and then applying a fault-tolerant averaging function to these values to compute a new clock value [LMS85, LWL88]. Among the fault-tolerant averaging functions used are egocentric average, fast convergence algorithm, fault-tolerant midpoint, and fault-tolerant average [Sch87]. These algorithms require a fully connected network, a known upper bound on message transit delay, and initial synchronization of the clocks.

In another class of algorithms, the clock values of various processors are first obtained through a protocol that guarantees an agreement among all correct processors on a vector of values, one from each clock [LMS85]. Each processor then applies the same averaging function to compute a new clock value. The agreement process is used to tolerate failures and ensures that all the processors apply the averaging function on the same set of values. In general, these algorithms do not require a fully connected network or initial synchronization of clocks, but do require a bound on message transit delays and a limit on the maximum number of processes that may fail.

A third class of algorithms use a synchronizer process to synchronize clocks [HSSD84, ST87]. In this approach, the synchronizer process collects values from all local clocks and then propagates these values to all processes, which use them to compute a new clock value. To avoid problems caused by the single point of failure represented by the synchronizer process, every process in the system periodically attempts to become the synchronizer at preset time intervals. At least one is guaranteed to succeed. An agreement protocol is used to guarantee that all correct processes attempt to become the synchronizer at roughly the same time. These algorithms require a bound on the message time delays and initial synchronization of the clocks. The network need not be fully connected.

A probabilistic approach has been used in [Cri89], where an algorithm is given that allows a process to read the clock on another machine to within some specified precision with a probability as close to one as desired. When a process succeeds in reading the clock, it knows the actual reading precision achieved. This method of reading a remote clock can also be used to improve most of the algorithms described above. A master-slave arrangement, in which one clock acts as master and others as slaves, is used to synchronize the clocks here, where the slave clocks adjust their value according to the value of the master clock. In general, the algorithms to elect a new master clock are fairly complex.

All the algorithms described above make the assumption that the network is synchronous, i.e., that message transit times are bounded. In [Mar84], the author assumes an asynchronous network and uses explicit timeouts to put a bound on processor and communication delay.

3.2 Logical clocks

The original approach for constructing a logical clock was proposed by Lamport in [Lam78]. In this paper, a *happened before* relation is defined that can be used to construct a logical clock in terms of the ordering of events in a distributed system. Specifically, given that a and b are events, a "happened before" b (denoted $a \rightarrow b$) if either of the following are satisfied.

- a and b are events in the same process and a comes before b, or
- a corresponds to the sending of a message and b corresponds to the receipt of the same message.

Furthermore, this relation is transitive, so that if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. If events a and b are such that $a \not\rightarrow b$ and $b \not\rightarrow a$, then they are said to be *concurrent*.

This relation is used to construct a logical clock C by assigning a value $C(a)$ to every event a in the distributed system. This value can be thought of as the logical time at which the event occurred. This assignment is done in such a way that the happened before relation is preserved, so that for any two events a and b, if $a \rightarrow b$ then $C(a) < C(b)$. A logical clock constructed in this manner can then be used to order the various events in the system.

The various algorithms that have been proposed to implement logical clocks differ in the notations they use and the amount of information they convey through the clock values. In the original solution by Lamport, the system-wide logical clock C is implemented by a collection of individual logical clocks C_i for each process P_i; here, C_i is a function that assigns an integer $C_i(a)$ to every event a that happens in process P_i. The logical clock assigns an integer $C(a)$ to such an event a by using C_i, *i.e.*, $C(a) = C_i(a)$. Each process P_i implements C_i by maintaining a counter K_i which is incremented between successive events. Also, on receipt of a message, m, P_i sets K_i to the larger of the current value of K_i and a value greater than the logical clock time of the event corresponding to the sending of m.

In this solution $C(a) < C(b)$ if $a \rightarrow b$, but the converse is not true. That is, $C(a) < C(b)$ does not necessarily imply $a \rightarrow b$. As a result, given any two events, it is not always possible to determine if they are concurrent using these logical clock values. Extensions have been described in [Fid88, Mat89] to rectify this problem. Here, the clock value is a vector of size n (sometimes called *version vector*), where n is the total number of processors in the system. Each entry i in this vector keeps a count of the messages received from process P_i. The update of the vector follows a similar procedure to that described above. Two vectors V_1 and V_2 can then be compared as follows :

$$V_1 < V_2 \quad if \;\; \forall i,\; 1 \leq i \leq n,\; V_1[i] \leq V_2[i] \;\; and \;\; \exists j,\; 1 \leq j \leq n,\; V_1[j] < V_2[j]$$

Using this, two events a and b are concurrent if the corresponding logical times, say, vectors V_a and V_b respectively, satisfy the following :

$$V_a \not< V_b \quad and \quad V_b \not< V_a$$

A logical clock is also constructed as part of the Psync multicast primitive [PBS89]. Here, the complete temporal order of message-passing events in the system is represented in the form of a graph called the *context graph*. A node in the graph represents an event corresponding to message transmission and an edge represents the happened before relationship. For any two events a and b, there is a path from a to b in the graph if $a \rightarrow b$. The absence of a path between a and b implies that a and b are concurrent events.

4 Multicast

Providing consistent information to multiple processes is important for constructing fault-tolerant distributed programs, particularly those structured using the state machine approach. A key component to providing such consistency is multicast, an interprocess communication (IPC) mechanism that provides the ability to send identical copies of a message to each process in a group. Such a service is useful in other kinds of distributed applications as well. For example, distributed database update and commit protocols, managing replicated data, distributed synchronization, and distributed transaction logging require multicast of one type or another.

Properties
Many different multicast services have been designed, each with features tuned to the specific requirements of the target application. Nonetheless, most multicast services provide some combination of five largely orthogonal properties, as follows.
Dissemination: The message is disseminated to all processes in a group. In a point-to-point network, this is achieved by sending a copy of the message to every process in the group separately. In local-area networks such as Ethernets and token rings that provide a multicast primitive, the dissemination can be done with a single lower-level operation.

Atomicity: The message is delivered either to all the correctly functioning processes in the group or to none. This property ensures that the information received by every functioning process is identical.

Reliability: The message is delivered to every process in the group. If a process has failed, a mechanism is provided to deliver this message following recovery.

Order: Messages sent by different processes are delivered in some consistent order at all the group members. Possible consistent orders include:

- *Partial order*: The messages are delivered in an order that preserves the causality or happened before relation. Processes may receive concurrent messages in different orders, but a message is only delivered after all the messages that precede this message in the relation have been delivered. This is sometimes called *causal ordering* [BJ87].

- *Semantic dependent order*: Messages are delivered in an order that varies at processes depending on the semantics of the information carried in a message. For example, the order of two messages could be different at different processes and still preserve the correctness of the application if the messages contain commutative operations. Typically, this ordering is a combination of other kinds of ordering.

- *Total order*: Messages are delivered in the same order to all the processes. In other words, if a message m_1 is delivered before m_2 at one process, m_1 is delivered before m_2 at every process.

- *Total order preserving causality*: Messages are delivered in the same order at all the processes and this order preserves causality.

These orderings become more and more restrictive as we go down this list and, in general, most costly in terms of how much synchronization they require between processes. As a result, the ordering used by an application should be the least restrictive that is sufficient to preserve the correctness of the application.

Termination: Every message is delivered to all correct processes in the group within a known time interval, even if concurrent failures and recoveries occur. This property can be satisfied only if the communication protocol is synchronous.

Examples of Multicast Services
The various multicast services that have been developed differ in which of the above properties they provide. A large number, typically called *atomic broadcast* services, provide atomicity and total order; examples include [BJ87, CM84, CASD85, KTHB89, MSM89, NCN88, PBS89, VRB89]. The total order provided by [PBS89, VRB89] also preserves causality, while the service provided by [CASD85]

also includes the termination property. An atomic broadcast service is useful in many distributed agreement applications, such as propagating updates to manage replicated data and committing distributed transactions.

The multicast service proposed in [GMS91] preserves both atomicity and reliability, but not necessarily order. This service, sometimes called a *reliable multicast service*, is useful in applications that need fast delivery of messages where the order of delivery is not critical. Examples include managing highly available replicated databases and some real-time applications.

The multicast services proposed in [BJ87, PBS89] provide atomicity and partial order. These services are useful in cases where concurrent events may be executed in different order at different processes. Moreover, using these services, it is possible to construct more restrictive multicast services. An example of this is found in [MPS89], where a multicast is described that provides atomicity and a semantic-dependent ordering based on the commutativity of the operations.

Algorithms

The algorithms used to implement multicast services are typically complex due to the uncertain nature of the communication network and the possibility of processor failures. In particular, messages may be lost or corrupted on the communication channel or may be received in different order at different processors, while processors may fail in different ways. As a result, the two main problems that are encountered in designing such algorithms—how to order messages and how to make the broadcast atomic—must deal with these situations. The way in which this is done is also influenced by the assumptions made about the topology of the network, the failure models used, and the synchrony of the network and processors.

In [CASD85], synchronized clocks are used to order different messages. Each message includes the clock time at which it was sent and the messages are ordered according to this time. The message is then delivered to each process at local time $t + \triangle$, where t is the time when the message was sent and $\triangle$ is a constant that depends on such network properties as message delivery time. Atomicity is achieved by diffusing every incoming message onto every outgoing link and treating non-receipt of a message with time $t + \triangle$ time units as a failure. With this approach, a family of broadcast protocols that tolerate increasingly general fault classes—omission, timing and Byzantine—is constructed. All these protocols assume a point-to-point communication network.

Algorithms proposed in [Lam78, MSM89, PBS89, BJ87, MPS89] use logical clocks to implement order. Atomicity is achieved either by using positive acknowledgements, where every receiver sends an acknowledgement for every message received [BJ87], or by using a negative acknowledgement scheme, where a retransmission is requested by the receiver only when a missing message is detected[PBS89, MSM89]. A point-to-point communication network is used in all but [MSM89], which assumes the existence of a broadcast network. All of these algorithms assume a crash failure model.

Another approach employs a single process to order messages [CM84, GMS91, NCN88, KTHB89]. In this approach, every broadcast message is first sent to one process, called the *funnel process*, that puts a sequence number on the message and then resends it to all the processes in the group. The messages are then delivered in an order corresponding to the sequence numbers. This approach only supports a total ordering among the messages exchanged in the system. There are also two other disadvantages to this approach. First, the funnel process is a single point of failure and the protocols must provide a way to recover from this failure, something that can be very complicated. Second, the funnel process is potentially a performance bottleneck since it must process every message. The atomicity in this approach is achieved by positive acknowledgements [NCN88], negative acknowledgements [KTHB89], or a combination of positive and negative acknowledgements [CM84]. The approach proposed in [CM84] uses a broadcast communication network, while the remainder assume point-to-point. Once again, crash failures are assumed.

Multicast protocols that tolerate Byzantine failures, commonly called *Byzantine agreement algorithms*, are inherently more complex. In [LSM82], it is shown at least $3t + 1$ processes are needed to tolerate t failures for any deterministic algorithm, and an algorithm is presented that achieves this bound. This algorithm is based on multiple rounds of message exchange among all processes and is essentially recursive in nature, with processes executing the algorithm for one fewer failure during each successive round.

Simplier Byzantine agreement algorithms can be constructed if authentication schemes [RSA78] or randomization [Rab76] are used. In the authentication schemes, a correct processor can sign a message such that any modification of the message can be detected by other processors and other processors cannot forge its signature. Using this, several algorithms have been proposed that can tolerate any number of Byzantine failures [DS83, LSM82, BD85]. Algorithms using randomization techniques make use of randomly chosen numbers to simulate a local random coin toss. Algorithms that use this approach [Rab83, CC85, Per85] differ from each other mainly in their assumptions about synchrony, in the number of failures they can tolerate, and in their complexity.

5 Remote Procedure Call

Remote procedure call (RPC) is an IPC mechanism based on the well-known and easily understood procedure call mechanism. In particular, an RPC is like an ordinary procedure call except that the invocation statement and procedure body are executed by two different processes, called the *client* and *server* respectively, potentially on different machines in a distributed system. When a remote procedure is invoked, the argument values are marshalled into a message by the client and transmitted to the server; any result values are returned in an analogous manner following execution. The synchronization is like that of a procedure call, so the client does not continue execution until the server completes execution of the invoked procedure and returns the results.

RPC has many attractive features. One is it possesses clean and well-understood synchronization semantics, which simplifies the process of writing distributed applications. Another is that it enhances network transparency by hiding the fact that the client and server may be on different machines. Yet a third is that its implementation can be optimized to the point that the resulting performance is superior to other IPC mechanisms [SB90] and even within an order of magnitude of regular procedure call in the case that the client and server are on the same machine [BALL90].

These features make RPC a good abstraction for building many kinds of distributed applications, including those with fault-tolerant requirements. However, in such applications, careful attention needs to be paid to the precise semantics in the presence of failures. For example, what is the effect on the program if the processor executing a remote procedure crashes during the call operation? Although failure to execute the desired action is sometimes a concern with regular procedures (these are often termed *exceptions* [Cri84, Goo75]), the nature and effect of failures on RPC are fundamentally different since two separate processes potentially communicating across a network are involved. This leads in turn to different problems and different approaches for dealing with failures in the context of RPC.

Properties of RPC

Given that RPC is intended to be a natural extension of standard procedure call to multiple processes, it is desirable that RPC semantics be as close to those of procedure call as possible. However, this is sometimes difficult given the inherent uncertainties associated with a distributed architecture, including the possibility of lost messages, out-of-order message delivery, processor failures, etc. To cope with these, a number of different execution semantics have been defined for RPC. These differ based on what inferences may be made in the invoking process about the number of times the remote procedure has been executed, both in the case when the invocation terminates normally and when it terminates abnormally; the latter occurs, for example, when the server crashes prior to completing execution of the remote procedure. The most common are [Nel81, PS88]:

- *At Least Once*: The remote procedure has been executed one or more times if the invocation terminates normally. If it terminates abnormally, no conclusion is possible, i.e., it may have been executed one time or multiple times, may have been partially through an execution, or may not have been executed at all.
- *Exactly Once*: The remote procedure has been executed exactly one time if the invocation terminates normally. If it terminates abnormally, no conclusion is possible other than it has not been executed more than once.
- *At Most Once*: The same as Exactly Once if the invocation terminates normally. However, in addition, the effect if it terminates abnormally is guaranteed to be atomic, i.e., has either been executed completely or not at all.

Communication and processor failures during a remote invocation can also give rise to *orphans*, which are unneeded computations that continue at the server even after a call has been terminated abnormally. For example, a client that crashes during an RPC may reissue the call after being restarted even though the server is still executing the original call. Another possible scenario is that the client reissues an RPC after failing to receive a response from the server within a specified period of time even though the server is still up and running. Orphans can cause consistency problems by interfering with subsequent calls to the server or with other computations going on in the system [PS88]. In addition, orphans waste system resources.

The possible creation of orphans is a fundamental problem for any RPC mechanism, so techniques must be provided to detect such computations and eliminate them. In addition, if the semantics of the mechanism require that there be no side effects in the event of an abnormally terminated call—as would be the case with At Most Once semantics, for example—the effects of the orphan must be undone. A number of RPC orphan detection and abortion algorithms have been described [LS83, PS88, RC89], and will be discussed further below.

Another property that should be preserved by an RPC mechanism is *call ordering*. This criterion states that a sequence of invocations generated by a given client should result in the computations being performed by the destination servers in the same order. This requirement can be relaxed somewhat, however, if the invoked procedures operate on disjoint data. Due to the synchronous nature of RPC, this criteria is trivially satisfied in the absence of failures, so the problem reduces to ensuring that orphans not invalidate call ordering. Note that this property is very similar to the partial ordering property of multicast, and that the ability to relax the requirement with disjoint data is like a semantic-dependent ordering.

Replicated RPC

RPC has been generalized in both system [Coo85, SDP91] and language [CGR88] contexts to work in cases where the client and/or server have been replicated to enhance fault-tolerance. This facility is called *replicated RPC*, with the replicated client termed the *client replica set* and the replicated server termed the *server replica set*. In this approach, an RPC results in independent invocation messages being generated by processes in the client replica set, with each client replica sending invocation messages to all processes in the server replica set. Upon receiving these messages, each server replica executes the appropriate remote procedure. No coordination is done between server replicas on a call, so the remote procedures will be executed concurrently. Upon completing execution, each server replica will send a reply message to every client replica. Note that the client and server can be viewed as replicated state machines, making this program structure a variant of the state machine approach.

In addition to the general RPC issues, there are other issues that arise due to the replicated nature of the invocation. One is fixing the point at which a server replica begins to execute the remote procedure, with the basic choices being either as soon as a message from one client replica arrives, or only after messages from all client replicas have arrived. Each choice represents a tradeoff. In the first case, execution can proceed with no delay, but the results of the procedure execution must be retained by the server replica until it has been communicated to every client replica. In the second case, the execution delay is longer, although error detection and transparent error correction can be provided by checking whether all the messages received are identical.

Another aspect of replicated procedure calls that is somewhat more complex is ensuring call order. When a server is replicated, not only must the concurrent calls from different client replica sets be ordered by each server replica, but the order chosen must be consistent across all replicas. This requires additional coordination that makes the implementation more complex than standard RPC.

Algorithms

Numerous RPC mechanisms have been described [BN84, Coo85, Cou81, LG85, LS83, PS88, RC89]. These algorithms differ from one other in their assumptions about the underlying network and the type of processor failures to be tolerated, in the particular semantics they implement, whether they support replicated procedure call, and in their treatment of orphans. The algorithms proposed in [BN84, Cou81, Coo85, PS88] implement Exactly Once semantics, while those in [LG85, LS83] are At Most Once. Techniques to detect and abort orphans are provided in [LS83, PS88], with orphan adoption being used in [RC89]. The algorithms in [BN84, PS88] can deal with a fixed number of communication failures (i.e., lost messages), while those in [Coo85, LG85, LS83, RC89] can also tolerate crash failures of processors executing servers. Only [LG85] and [LS83] add support for recovery to remove side effects caused by orphans prior to being aborted.

All of these algorithms can be viewed as having two distinct components. The first deals with issues such as naming a procedure, locating remote machines, and managing message transfer, while the second deals with detecting failures, and detecting and aborting orphans. The algorithms are broadly similar in how they implement this functionality, although there are many differences in the details. A typical scheme is to have a manager process on every processor that can be contacted using some well-known network address. This process acts as a conduit for all RPC requests either originating from or destined for its machine. In the first instance, it accepts the message from a local client and then either deals with the invocation itself if the server is local, or forwards it to the appropriate manager if not. In the second, it accepts messages from other managers for servers on its machine and generates the invocation. In [PS88], the manager process spawns a server process on receiving a request, with all subsequent calls from that particular client being handled directly by the server. In [BN84], every call goes through the manager.

The algorithms differ from each other in the way they handle orphans. In [PS88], orphans are managed by including a deadline and crash count in every invocation. The deadline indicates the maximum time allowed for execution of the remote procedure by the server; if a server exceeds this deadline, it aborts itself. The crash count is used to distinguish between calls prior to a processor crash from new calls. In [RC89], servers are replicated, so an orphan is adopted by one of the secondary replicas of the server when the primary one fails. Algorithms proposed in [LG85, LS83] provide backward error recovery techniques to abort an orphan and remove side effects. These techniques are similar to the ones described in Section 7.

6 Membership

To ensure consistent actions, a group of cooperating processes typically needs to have an agreement on the set of functioning members at any moment in time. Changes in group membership may occur due to the failure of processes, the recovery of previously failed processes, new processes joining the group, or a process voluntarily leaving the group. A membership service is used to maintain such a consistent, system-wide view of which processes are functioning at any given moment. This service has proved to be one of the most fundamental services in fault-tolerant distributed systems, simplifying many problems. It is especially associated with the state machine approach, although variants are used in other paradigms as well.

There are actually two types of membership services, each serving a different purpose [VM90]. The first can be viewed as a user-level service that typically translates the failure or recovery confirmation into an event that is then ordered with respect to other events in the system. This ordering is then made available to the application to use in making decisions. Examples of this kind of service include [BJ87, CM84, Cri88, KGR91]. In this case, the application program is explicitly notified of the changes in the group membership.

The other type of membership service is sometimes called a *monitor service* [VM90]. In contrast to the user-level orientation of the first type, the monitor service is used by the system itself to maintain a consistent view of which processes are functioning and hence participating in system decisions. For example, such information is used in reliable multicast protocols to determine when a message has been received and acknowledged by every functioning process so that it can be committed to the application. The processor failure or recovery event must again be consistently ordered with respect to other events such as interprocess communication to guarantee that messages are committed consistently, but the failure notification is not necessarily passed on to the application. Examples of this kind of protocol include [MPS92, VM90].

Properties
Intuitively, an algorithm solves the membership problem if it ensures that the processes using this service remain consistent in the presence of failures and recoveries. Although this implies that the solution to the membership problem is application-dependent, there are solutions that are general enough to ensure the correctness of any distributed application. Typically, such a solution enforces agreement among all the processors on a unique sequence of process joins and departures, and the precise way in which these membership changes interleave with regular events such as message receipt. A large number of membership services satisfy this condition [BJ87, CM84, Cri88, KGR91, RB91]. However, such a condition may actually be overly restrictive for many applications. The membership protocol described in [MPS92] is less restrictive in this regard. Here, an *sf-group* at process P is defined to be the set of all the processes that have failed simultaneously as perceived by the process P. The proposed solution ensures that all the processes in an sf-group are removed simultaneously and the order of removal of these sf-groups is the same at all the processes, but the points at which these changes occur need not be same at all processes.

There are some critical applications, such as process control, in which the membership service must also satisfy the *timeliness* property. This property states that, once initiated, the membership service is guaranteed to terminate in a known real time interval. This property is typically satisfied by membership services implemented using clock-driven protocols, that is, protocols built on top of a global time base [Cri88, KGR91]. Membership protocols constructed without such a facility do not satisfy the timeliness property.

Failure and Recovery Detection
As mentioned above, changes in membership occur when a process fails or recovers. Thus, the membership protocol is initiated when a process is suspected to have failed or when a functioning process learns about the recovery of a previously failed processor. The technique used to detect the failure varies from system to system depending on the system model used. Typically, a failure of a process *P* is suspected when no messages from *P* arrive in a given interval of time. This *failure detection* mechanism is typically implemented by a *heartbeat* protocol where every functioning member of the group periodically sends "I am alive" messages. Examples of protocols using such a mechanism include [BJ87, Cri88, KGR91]. This detection protocol can also be application dependent, where the application messages being exchanged are monitored and a failure is suspected when a message expected by the application fails to arrive within certain interval of time [MPS92]. For systems that assume asynchronous processors and communication, it is this failure detection mechanism that puts an upper bound on response time, and hence, essentially adds the synchrony required to reach agreement. Notification of recovery is typically done explicitly by the recovering process as part of its reboot process; when received by other group members, the membership protocol is initiated. These mechanisms may also be used to detect failures or recoveries while the membership protocol itself is in progress, thus allowing simultaneous failures and recoveries to be handled.

Network Partitions
A network partition occurs when a subset of processes in the group cannot communicate with another subset due to a failure. In such a case, processes in each subset may conclude that all the processes in the other subset have failed. Some of the clockless membership protocols can tolerate network partitions by allowing a subset with a clear majority of processes to continue functioning [CM84, RB91]. However, clock-driven protocols cannot tolerate a network partition since this may lead to divergent views among different processors. There are known techniques to reconcile divergent views [SSCA87], but inconsistent actions may be taken while the reconciliation protocol is in progress.

Algorithms
A number of algorithms have been proposed to solve the membership problem. In [Cri88, EL90, KGR91], the authors have proposed clock-driven solutions to the membership problem for systems with a broadcast communication network. The algorithm proposed in [Cri88] relies on an atomic broadcast service and a message diffusion service; periodically, each process affirms its existence by sending a *present* message. In [KGR91], global time is used to control access to the communication channel by a synchronous TDMA (Time-Division Multiple-Access) strategy; a process includes certain membership information with every message that it broadcasts, which is then used by all the processes to compute group membership.

Clockless membership algorithms such as those proposed in [BJ87, CM84, MPS92, RB91] tend to be more complex since they typically assume only asynchronous processors and then add synchrony through mechanisms such as failure detection. A completely connected network with FIFO channels is required in the algorithm proposed in [RB91]. A distinct manager process is used to coordinate updates to the other processes' local views. A two-phase protocol is used by the manager to coordinate updates and a three-phase protocol is used to select a new coordinator when the manager is thought to have failed. The protocol proposed in [CM84] also makes use of a distinct manager process. In this approach, all the normal traffic is suspended while the protocol is in progress. The protocol is three-phase for the manager process and two-phase for other processes. The protocol proposed in [MPS92] is fully distributed in the sense that it does not require a single manager process; instead, the functioning processes use multicast communication among themselves to agree on removal of failed processes. As mentioned above, a novel feature in this algorithm is that additional failures during execution of the membership protocol are handled incrementally.

7 Atomic Actions

Atomic actions are an abstraction that is central to the structuring of many fault-tolerant distributed systems [Lam81, Lis88, SDP91, Svo84]. An atomic action is defined informally as a program-specified computation that, although composed of many primitive computational steps executed at different times and by different processors, is seen as an indivisible state transformation by other computations despite concurrency and failures. The relevance of atomic actions to the design of fault-tolerant systems is that they provide a simple framework for controlling the effects of failures, since a failure can only occur (conceptually) between atomic actions. They are most commonly used in contexts where long-lived data stored on stable storage is subject to concurrent access by multiple processes.

An atomic action satisfies two important properties: *serializability* and *recoverability*. The serializability property states that the effect of executing a collection of atomic actions is equivalent to some serial schedule in which the actions are executed one after another. The recoverability property states that the external effect of an atomic action is all-or-nothing; that is, either all the state modifications performed by the atomic action take place or none of them. Note that this potentially involves multiple processors. These properties and various techniques to implement them have been discussed in detail in [BHG87, Koh81]. We discuss some of the salient features of these in the following.

7.1 Serializability

A simple way to implement serializability is by forcing actions to actually execute sequentially. However, this method does not allow the constituent steps of the various actions to be interleaved or executed concurrently, a decided disadvantage especially in a distributed system. The usual method for avoiding this problem is to synchronize access to shared resources in such a way that the overall effect is as if the actions had been run sequentially even through concurrent execution is actually taking place.

Algorithms
One of the most popular techniques for implementing serializability is *two-phase locking* [BSW79, EGLT76, Pap79]. In this scheme, a lock is associated with each shared resource, with the requirement that a lock be acquired prior to any access of the associated resource. The action is further constrained in the order in which it can acquire and release locks to go through two distinct phases. In the first, sometimes called the *growing phase*, needed locks are accumulated; in the second, called the *shrinking phase*, locks are released. The key to this scheme is that an action is prohibited from acquiring additional locks once it has entered its second phase by doing a release. While ensuring serializability, two-phase locking can lead to deadlocks in which one or more actions waiting to acquire locks may block forever.

Deadlock detection and elimination schemes such as [CES71, GS80, Hol72, Mar76, MR79] must be used in such situations. This deadlock may also be avoided by using a conservative approach in which every action acquires all the locks that it needs at one time.

A second technique for implementing a serializable schedule uses timestamps to order various actions [SM77, Tho79]. A timestamp is a system-wide unique number chosen from a monotonically increasing sequence that is assigned to an action. In this technique, a shared resource is accessed by various actions in their respective timestamp ordering. If the timestamp of an action trying to access a shared resource is smaller than some action that has accessed that resource then this access is denied and the corresponding action must be aborted. This scheme essentially regulates the access to shared resources according to the logical global time at which the actions start (see Section 3). Timestamps can also be used to avoid deadlocks in the two-phase locking scheme [RSL78].

Another approach maintains a dynamic graph of how different actions are accessing various shared resources at any point in time [Bad79, Cas81, HY86]. This graph contains a node for each action that is currently executing, as well as a node for actions that have committed. The edges between nodes specify dependencies between actions, and serializability is guaranteed by ensuring that the graph always remains acyclic. The unbounded growth of this graph is controlled by deleting the nodes and the corresponding edges for the actions that will not be involved in a cycle at any time in the future. An easy way to do this is to delete the nodes that have no incoming edges and the corresponding action has been terminated.

Many more techniques have been proposed to ensure serializable schedules, including those based on token circulation [LeL78], analysis of conflicts among various actions [BSR80], use of reservation lists [Mil79], and certification tests [KR81]. Many of these techniques can be combined in various ways to produce additional approaches [BG81, BGL83].

7.2 Recoverability

The recoverability property implies that the system state at any given time—and in particular, the state of data on stable storage—reflects only the effects of completely executed actions. The mechanism for realizing this is a *commit* operation, which is executed by the action to make its state changes across all of the machines on which it executed available to other actions; a commit is an irrevocable action that must appear to be indivisible with respect to failures. An action that does not commit due to, say, bad data or an untimely processor failure, is aborted; in this case, all machine states must be restored to their original values. Thus, the two problems that must be addressed are, first, providing a mechanism to install and restore the state on an individual machine, and, second, ensuring that the decision on whether an action is committed or aborted is made consistently across all machines.

Installing and Restoring State

The techniques used to deal with the problem of state installation and restoration depends on which of two basic *update strategies* are used [BHG87]. One strategy is called *in-place updating*; this involves keeping a single copy of each data element that is modified directly by actions during execution. Installing the new state is trivial and typically involves releasing locks; note that this installation is easy to restart should, for example, it be interrupted by a processor crash. Restoring the state should the action be aborted is somewhat more difficult since each modified data element must be restored to its old value. Perhaps the most common way to do this is to maintain an incremental log of all changes on stable storage, which can later be used to recreate the initial state of a data element.

The second update strategy is called *shadow updating*. This strategy involves maintaining two copies of each data element, and also two copies of directory or index that is used to access the data elements. One copy of the index contains references to the *current copy* of each data element, that is, the value from the most recently committed actions. The second copy contains references to a *shadow copy* of each data element, which is the copy updated by actions that have not yet committed. The *current copy pointer* indicates which index contains references to the current copies of the date elements. Installing a new state when an action commits is done by changing the current copy pointer to indicate the other index, effectively reversing the roles of the current and shadow copies. Note that this can be done indivisibly since it involves changing only a single value. Restoring the state of an aborted action is trivial: the current copy pointer is simply not changed. This leaves each data element in its original state.

Commit Protocols

While the above techniques solve the problem of committing or aborting actions on a single machine, an atomic action in a distributed system may execute across many machines prior to committing or aborting. Thus, the second problem to be addressed is guaranteeing that machines make a consistent decision on whether to commit or abort a given action, a process that is complicated substantially by the need to tolerate failures during the decision-making process. A *commit protocol* is an algorithm that ensures such consistency despite failures [Gra78].

A large number of commit protocols have been proposed [DIW89, Gra78, LS76, ML83, NS89, Ske82a, SC90], with different approaches based on assumptions about the type of failures, the network model and so on. The best known of these protocols is the *two-phase commit protocol* [Gra78], which is designed to reach a consistent decision despite processor crashes. In this approach, a collection of processes, one on each machine on which the atomic action executed, cooperate to decide whether to commit or abort the action. One of these processes—usually the one on the machine where the action originated—acts as the *coordinator*, while the others are *participants*. In this scheme, all participants make a tentative local decision as to whether to commit or abort. The protocol then guarantees the following:

- All processes that reach a final decision reach the same one.
- If all local decisions are to commit and there are no processor crashes and communication failures during protocol execution, the final decision is to commit.
- If all machines that crash eventually restart and remain up sufficiently long, all processes eventually reach a final decision.

In the first phase of the protocol, the coordinator starts by sending a prepare message to each participant. Upon receipt of such a message, the participant replies "commit" if its local decision is commit and "abort" if its local decision is abort. If the coordinator receives at least one negative response, it multicasts an abort message. Upon receiving such a message, a participant does a local abort of the action, and acknowledges receipt. On the other hand, if all processes have replied affirmatively, the coordinator commits the action by writing a *commit record* to stable storage and multicasting a commit message. On receipt, each participant commits the action on its machine and sends an acknowledgment.

The consequences of a failure of the machine executing the coordinator depend on how far the protocol has progressed at the time of the failure. If the failure occurs after the coordinator has decided to commit, as evidenced by the existence of the commit record, then upon restart, the coordinator checks to ensure that every participant has been notified to commit the action. If a commit record is not found, an abort message can be transmitted. The effect of the failure of a participant's machine also depends on how far the protocol has progressed. Should the failure occur during the prepare phase, the action is aborted by the coordinator. On the other hand, should failure occur after a final decision has been reached, the coordinator waits until the participant recovers and then retransmits its decision. Note that the protocol is subject to blocking if the coordinator crashes or if the network suffers a partition.

The two-phase commit protocol has been extended and improved in many different papers, including [Bor81, DIW89, HS80, ML83, MSF83, MLO86, SC90]. A three-phase commit protocol has also been developed to avoid the blocking property mentioned above [Ske82a, Ske82b]. Many of these variants are described in [BHG87].

8 Resilient Processes and Stable Storage

A resilient process is a process that can continue to execute correctly even if interrupted by a failure and then restarted. This abstraction is oriented towards crash or fail-stop type failures, so the intuition is that these are processes whose execution was suddenly halted at some arbitrary point and then later restarted, either on the same processor after repair and reboot, or on another functioning processor. The hardware model also typically assumes that storage is divided into two types: volatile storage, whose contents are lost when the failure occurs and stable storage, whose contents remain intact. Given that stable storage is an important abstraction for programming fault-tolerant systems in its own right, we briefly elaborate on it below as well.

8.1 Recovery Techniques

Implementation of resilient processes is based on *recovery techniques*, which involve restoring the process to some well-defined state following a failure so that it can continue execution. The most common variant of this strategy is *backward recovery* in which enough values are saved on stable storage to enable some past state of the process to be reconstructed should a failure occur. Recovery techniques are useful in all of the programming paradigms outlined in Section 2. In particular, recovery is used to maintain the atomicity of object operation execution in the object/action model, to ensure that the states of all processes remain consistent following failure in the conversation model, to provide a starting state for a new primary in the primary/backup approach, and to reconstruct the state of a failed state machine in the state machine approach.

The most common form of backward recovery is based on the use of *checkpoints*, in which the entire state of a process is periodically written to stable storage. Then, should a failure occur, the most recent checkpoint is used as the beginning state after restart. This checkpoint must be written atomically with respect to failures, implying the use of recoverability techniques similar to those described in Section 7.2. Most often, a variant of shadow updating is used in which two copies are maintained along with an indication of which is current. If stable storage is implemented using multiple machines (see below), a commit protocol is needed as well.

The decision of when to checkpoint involves a tradeoff between the time it takes to write the checkpoint to stable storage and the amount of computation that must be redone in the event of failure. Details of the particular application also can play a role; for example, it may not be possible to restart the computation from every state, which would make these ineligible for use as a checkpoint. Or, one might choose a state in which the size of the state to be checkpointed is small in order to minimize the overhead associated with writing to stable storage. Yet another factor to be considered in the timing of checkpoints is whether values generated by the program as output can be safely repeated; this can occur since any computation after a checkpoint may potentially be executed more than once in the event of failures. If this is not feasible, then the output and the checkpoint must be done as one atomic action. It is essentially this same problem that can cause difficulties with interacting resilient processes, where the output in this case consists of the messages being transmitted between processes.

Checkpointing and, in fact, most recovery techniques rely on stable storage, an abstraction of perfect storage that survives processor failures [LS76]. Access is performed using atomic read and write operations, with different techniques used to implement failure resilience depending on the needs of the application. For some, keeping a single copy of the values on a non-volatile device like a disk is sufficient. For others, where the cost associated with losing values in stable storage (i.e., the abstraction failing) is less acceptable, redundant copies are kept. These can be on the same device or, if more safety is needed, on multiple devices with independent failure behavior. The abstraction can also be implemented by having redundant processes executing on multiple machines keep copies of the data, such as done in [CASD85]. [1]

Finally, as already noted above, the techniques for achieving atomic access to stable storage are identical to those described earlier for atomic actions. This naturally raises the question of whether these techniques are implemented in this case as part of the stable storage abstraction or as a separate service. Although this depends on many factors (e.g., whether data replication is used), for the purposes of this paper, we adopt the view that stable storage directly provides atomic access for relatively small-grained values (e.g., a single variable), with the atomic action abstraction used to implement atomicity for writing multiple values, such as would typically be required for a checkpoint.

8.2 Interacting Resilient Processes

Consider a distributed program in which resilient processes interact with each other using message passing to accomplish a task. Following [JZ90], we characterize such a process by a sequence of events, where an event is either a local computation, or the send or receipt of a message. The state of a process after receiving a message, say m, becomes dependent on the state the sender had just before it sent m. [2] Thus, as a result of message exchanges in the system, the states of various processes become dependent on one another in interlocking ways. Define the *system state* at a particular time to be a history of events that constitute the set of all process states at that time. Then, a system state is said to be *consistent* if for every event corresponding to the receipt of a message in the state, the event corresponding to the sending of that message is also included [JZ90].

In a collection of interacting resilient processes, the fundamental issue is ensuring a consistent system state after recovery following a failure. The particular problem is that it may be necessary to modify the states of processes other than the one that was on the failed processor because of the state dependence caused by message passing. For example, consider a scenario in which a process fails after sending a message that is received by another process. If the state to which the process is restored during recovery is prior to the send, then the corresponding event will no longer be in the process state. As a result, the resulting system state will be inconsistent unless the event corresponding to the message receipt is also removed from the state of the receiving process. Special recovery techniques have been designed to deal with this type of problem by ensuring that the state of the system remains consistent following recovery of one or more processes.

[1] This technique is, in fact, just an example of the state machine approach, so many of the services described in this paper are directly relevant to the problem of implementing stable storage as well as other applications.

[2] This notion of state dependence is also captured in formal axiomatic rules for message passing as, for example, in [AFdR80, LG81, SS84].

Checkpoint and Rollback Recovery

The first technique is based on each process checkpointing similar to that described above. Recovery then involves rolling back all the processes to the most recent combination of saved states that gives a consistent system state. There are two approaches to creating these checkpoints. In the first, each process periodically checkpoints independent of the other processes. During recovery, then, the processes must dynamically determine a set of checkpoints, one from each process, such that the system state constructed out of these checkpoints is consistent. In this approach, no coordination between the processes is required while checkpointing but processes must coordinate during recovery. One of the drawbacks of this approach is that the rollback of a process may result in a cascade of rollbacks that, in the worst case, can push all processes back to their starting states. This is again the *domino effect* mentioned in Section 2 [Ran75, Rus80]. Moreover, since cascading rollbacks may require any of the previously stored checkpoints, the processes must retain all of their checkpoints indefinitely.

This independent checkpointing approach is used in a variety of contexts [BL88, Had82, Kim78, KYA86, MPS91, Ng88, RS88, SY85]. The scheme proposed in [Had82] is limited to a centralized database, while the ones proposed in [Kim78, KYA86] rely on an intelligent underlying processor system to automatically establish checkpoints of the coordinating processes. In [BL88], a recovering process computes the set of globally consistent checkpoints by invoking a two-phase rollback algorithm. In the first phase, it collects the information about relevant message exchanges in the system and uses it in the second phase to determine both the set of processes that must roll back and the set of checkpoints up to which rollback must occur. In [Ng88], the authors propose a commit protocol for checkpointing distributed transactions. Although the domino effect is possible here, it is shown that the lost work can be reduced by reusing portions of completed computations. In [RS88], synchronized clocks have been used for checkpointing and rollback recovery; these clocks coupled with the idea of a pseudo-recovery block approach [SL84] are used to develop a checkpointing algorithm. Independent checkpointing is also done in [MPS91]. However, no domino effect is possible here because checkpointing is done just as an optimization and the state of a process can fully be recovered from information stored at other processes.

In the other main approach, processes coordinate with each other to checkpoint [BS83, KT87, LB89, TS84]. Typically, the processes use a two-phase commit protocol to checkpoint, thus ensuring that the set of checkpoints stored is consistent. In this scheme, two checkpoints need to be stored at any time: a permanent checkpoint that cannot be undone and a tentative checkpoint that can be undone or changed to a permanent checkpoint. Note that even with the coordinated checkpointing, there is a need for some synchronization. In the absence of such synchronization, processes cannot all restore their checkpoints simultaneously and *livelocks*, in which processes endlessly cycle, can be introduced [KT87]. To avoid this, the recovery is again done in two phases. In the first phase, a request to restart from a checkpoint is sent; in the second, a decision to restart is propagated.

Message Logging

Independent checkpointing can be enhanced by the use of message logging in a technique sometimes called *optimistic recovery* [JZ90, SW89, SY85]. In these schemes, processes checkpoint independently and log input messages along with some dependency information in stable storage. Recovery then consists of (a) restoring an earlier possible state of the failed process using a checkpoint from the stable store plus potentially replaying the logged messages, (b) recognizing the set of processes whose states depend on lost states using the dependency information and rolling them back, and (c) committing messages to the outside when it is known that the states that generated the messages will never need to be undone. The logging of messages can also be done on volatile storage as has been shown in

[JZ87, PBS89, SY85]. In this case, messages are logged on the volatile storage of other processes and then replayed to the recovering process at the time of recovery.

9 Common Dependencies

In the preceding sections, we have described some of the key abstractions that have proved important for constructing dependable distributed systems, focusing on both the fundamental properties of each abstraction and the most important approaches used to realize these properties. In doing so, however, each service was treated largely as an isolated entity without concern for how it might interact with other services. The reality of the situation, of course, is that any given system usually contains a number of these fault-tolerant services that interact in various ways and use one another to implement their functionality. For example, in Section 4 we outlined how some approaches to multicast use the functionality provided by a common global time service to implement a consistent message ordering. Following [Cri91], we term such relationships *dependencies*, where a service u depends on a service v if the correctness of u depends on the correctness of v; in this case, we will informally refer to u as the *higher-level service* and v as the *lower-level service*. This notion of dependency can be viewed as generalizing the kind of level-structuring that has long been common in operating system design [Dij68].

As might be expected, the dependencies exhibited by a given system vary based on the programming paradigm used, and other details of the design and implementation. However, some dependencies are essentially independent of a specific system or implementation, and hence, are more common than others. In this situation, the lower-level service usually provides some fundamental function without which the higher-level service cannot be implemented. [3] The goal of this section is to identify and explain some of these common dependencies.

The graph in Figure 3 shows some of the common dependencies among the programming paradigms and fault-tolerant services we have described. In this figure, the rectangles are paradigms or services, and edges are dependencies. The edge labels indicate the property (or properties) that induce the dependency; that is, they indicate properties of the higher-level entity that require the functionality of the lower-level entity to be realized. Note that a new abstraction called “Atomic Actions (shadow updating)” has been introduced. We simply use this term to refer to single-machine atomic actions where recoverability is implemented by the shadow updating (or two-copy) approach described in Section 7.2; this particular implementation technique for atomic actions is being separated from the others in order to clarify certain dependencies. Two additional caveats are also in order before describing the graph in more detail. First, it should be emphasized that this graph is not intended to capture all possible combinations of services or dependencies, but only certain common patterns. Second, the graph should be considered speculative at best, since these services often interact in subtle ways that are only now beginning to be understood.

Programming Paradigms

Many of the direct dependencies from the four programming paradigms to lower-level fault-tolerant services should be clear based on their descriptions in Section 2 and subsequent discussions. The object/action model, of course, uses atomic actions as one of its fundamental concepts, and also relies on RPC for communication between objects. A conversation is a collection of resilient processes that interact by message-passing, making that edge its fundamental dependency to a fault-tolerant service.

[3] Whether these two services can in fact be identified as separate in the implementation is a different issue.

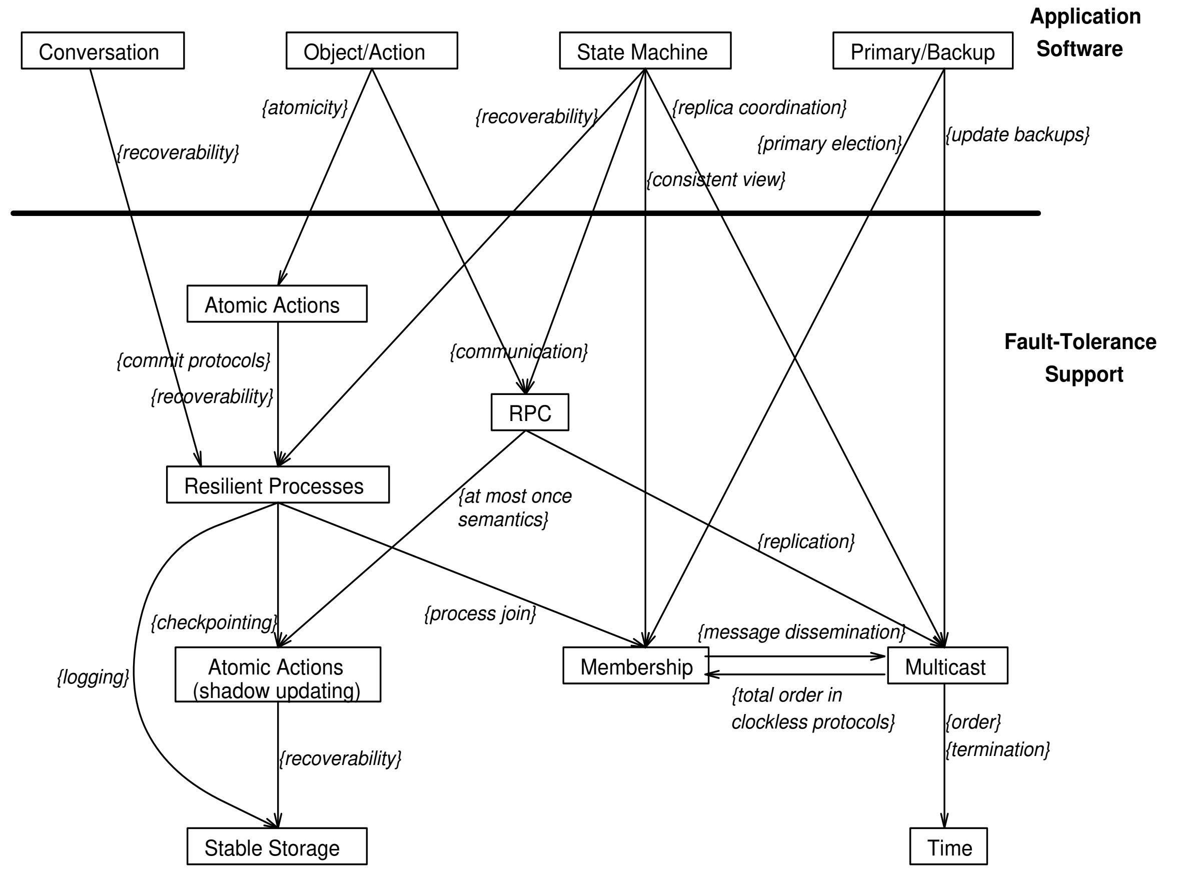

Figure 3: Common dependencies

The state machine approach uses a number of the abstractions directly. Multicast is often used to coordinate replicas, usually a variant with at least atomicity and ordering properties. Membership is also used to provide consistent information on which replicas are functioning at any given time. In some systems, replicas are reintegrated into the system upon recovery following a failure, which means that each replica is a resilient process. Finally, interaction between replicas of different state machines is sometimes implemented using either regular or replicated RPC.

The primary/backup approach commonly depends on both multicast and membership services. Multicast is useful for updating backup processes to reduce the amount of recomputation required when the primary fails, while membership is needed to recognize the failure of the primary. Another common dependency, although not shown explicitly as an edge on the graph, is to the lower-level atomic actions; this occurs if the primary employs checkpointing to save intermediate states for a backup to use should a failure occur.

Atomic Actions

The major dependency edge for atomic actions is to the resilient process abstraction since many of the relevant implementation techniques require that a process execute some recovery action upon restart. For example, the use of in-place updating for realizing recoverability on a single machine requires that the modified data elements be restored to their original values after recovery. Another example is found in the two-phase commit protocol for coordinating actions across multiple machines; in this technique, the coordinator process reads a commit record following recovery and then either commits or aborts the action.

Remote Procedure Call

Some of the variants of RPC have dependencies to other fault-tolerant services. For example, an RPC mechanism defined to have At Most Once semantics is dependent on the atomic action abstraction because of the need to implement orphan removal; that is, the remote procedure needs to be executed as an atomic action by the server to guarantee that any effects are undone and that call ordering is not violated should the process become orphaned. In the graph, we have drawn this edge to the lower-level atomic action since checkpointing is the most common implementation technique; alternatively, the edge could be to the higher-level variant if logging is used instead. Another variant of RPC that has dependencies is replicated RPC. Specifically, this type of RPC often depends on the multicast abstraction to ensure that messages are delivered to a collection of processes reliably and in some consistent ordering.

Resilient Processes

The resilient process abstraction has several properties that can induce dependencies. One that has already been mentioned is recovery of a previous state following failure; this requires writing checkpoints and/or log entries to stable storage. Given that checkpoints are usually large, the dependency here is to lower-level atomic actions, which implement the atomicity aspect of the write. Log records are usually on the order of a single value that can be written atomically by stable storage directly, so the dependency in this case is directly to stable storage. Another relevant property when a collection of processes are cooperating is ensuring that the recovering process is reincorporated back into the group. Doing this consistently across all processes is the task of the membership service, leading again to a dependency relationship between abstractions.

Atomic Actions (shadow updating)

The basic dependency for this type of atomic actions is to stable storage, for two reasons. One is simply that atomic actions at this level are typically used to manipulate long-lived data of the type that is typically stored on stable storage. The other is that the shadow updating technique uses redundancy in the form of extra data stored on stable storage as the basis for implementing recoverability. Specifically, two copies of the data are kept on stable storage, along with an indicator of which is current.

Multicast and Membership

The dependencies involving the multicast and membership services tend to differ depending on whether the protocols are clock-driven or clockless, that is, whether or not they assume a common global time base implemented by synchronized clocks. In both the clock-driven and clockless approaches, the time service is used to provide consistent message ordering; however, in the clock-driven approach, synchronized clocks are used, while logical clocks are used in the clockless approach. Multicast services that satisfy the termination property, which are typically clock-driven, are also commonly implemented using global time to achieve that property.

The biggest difference comes when considering the relationship between the multicast and membership services themselves. In clock-driven systems, multicast can determine a consistent order for received messages independent of an explicit membership protocol by using the synchronized clocks and assumptions about the synchrony of the network and processors. Specifically, for a message sent at a given time and timestamped with that value, the receiving process need only wait long enough to ensure no earlier messages will arrive. This approach is not sufficient with a clockless protocol, however, since the logical timestamp included on a message puts no bound on how long the receiver must wait to ensure no earlier messages. To overcome this, acknowledgements (implicit or explicit) are used, which allows the receiver to conclude no earlier messages will arrive after it has received a message from every other functioning process with a timestamp at the same or greater (logical) time. [4] But knowing "every other functioning process" consistently across machines requires membership, thus leading to a dependency edge from multicast to membership for clockless protocols. The alternative clockless algorithms that use a funnel process also need membership to detect the failure of the funnel process and subsequent election of a new funnel process.

There is a more fundamental reason for the dependency of multicast on membership in clockless protocols. As shown in [DDS83], reaching agreement in the presence of failures in a distributed system is possible if both the communication system and processors (processes) are synchronous. Clockless protocols approximate this first property given an asynchronous network by using low-level message acknowledgments and retransmission. However, the second property of synchronous processors is still needed to reach agreement, in this case on the consistent total order of messages to be enforced by multicast. It is this property that is supplied by the membership service. Specifically, the failure detection aspect of membership approximates this property by putting an upper bound on the amount of time processes can execute before they are considered to have failed. Thus, in clockless protocols, multicast depends on the membership service to realize the synchronous process requirement needed to reach agreement.

10 Fault-Tolerant Systems

The abstractions outlined in the previous sections have been used as fundamental components in a number of fault-tolerant distributed systems that have been designed and/or implemented over the past decade. Here, we briefly outline some representative examples: the Advanced Automation System (AAS) [BDD+89, CDD90], Argus [LS83, Lis88], Consul [MPS91], Delta-4 [PSB+88], ISIS [BJ87, BSS91], and MARS [KM85, KDK+89]; our specific emphasis is on describing how each fits into the framework developed above. Others systems of interest include ADS [IM84], ANSA [Tea91, OOW91], Arjuna [SDP91], Avalon [DHW88], Chorus [BFG+85], and Clouds [LW85].

10.1 Advanced Automation System

AAS is a fault-tolerant distributed system currently being developed by IBM as the next-generation air-traffic control system for the U.S. Federal Aviation Administration. The system is structured as a collection of Area Control Computer Complexes (ACCC), each of which manages one of the 23 areas into which U.S. airspace is divided. An ACCC is, in turn, structured as a distributed system of workstations and mainframes connected by a local-area network; among its tasks are to provide air traffic controllers with display information concerning the location of aircraft within the area based on radio and radar input, process flight plans, and interpret commands from air traffic controllers. Each ACCC also communicates with other computing complexes, including other ACCCs to implement transfer of aircraft between areas, airport tower complexes to coordinate takeoffs and landings, and weather computer systems. As might be expected, the availability requirements for this system are

[4] A message for which acknowledgments have been received is called a *stable message* in [PBS89] and *fully acknowledged* in [Sch82].

stringent; for example, certain critical services are not supposed to be unavailable for more than 3 seconds per year. Such specifications have lead to the extensive use of fault-tolerance techniques.

The software associated with an ACCC is organized as a collection of services, each of which implements a particular function. For example, the Surveillance Processes and Correlation (SPC) Service tracks aircraft based on radar input. To increase availability in the face of failures, services are implemented by groups of redundant server processes executing on separate physical machines. These groups are organized using either the state machine approach or primary/backup, with the choice depending primarily on the real-time requirements of the particular application. For example, the SPC service has strict requirements in this regard and so uses the state machine approach, while the Flight Planning Service uses primary/backup since a longer interval of unavailability can be tolerated. Group management functions—for example, coordinating promotion of a backup to a primary—are localized in a Group Service Availability Management (GSAM) Service, which is itself implemented by redundant processes using the state machine approach. The failure model assumed throughout corresponds to performance failures.

To support these paradigms, AAS uses a number of the abstractions described in previous sections, including a time service, multicast, membership, RPC, and resilient processes. The time service is provided by synchronizing clocks using the probabilistic scheme mentioned earlier [Cri89]. The multicast service, termed atomic broadcast here, provides atomicity, consistent total order, and termination. Interestingly, it and the membership service are used only in the implementation of the GSAM service, with a separate replicated RPC service being used to implement group communication for the other services; this RPC provides either At Least Once or At Most Once semantics depending on the situation. The checkpointing and message-logging techniques associated with resilient processes are used to implement redundant server processes organized according to the primary/backup approach.

The dependency structure of the fault-tolerant services essentially follows the clock-driven organization, with the time service realized using clock synchronization as the lowest layer. On top of this are, successively, multicast, membership, and then the GSAM service. The replicated RPC, as mentioned, is implemented separately, and so depends on none of these. The GSAM also depends on recovery to realize the primary/backup approach.

10.2 Argus

Argus is a programming language and system for constructing fault-tolerant distributed programs that has been designed and implemented at MIT [LS83, Lis88]. The system is oriented towards applications in which preserving the consistency of long-lived data is the primary concern, such as would be found, for example, in banking or airline reservation systems. Thus, the emphasis has been on developing mechanisms to efficiently manipulate such data, while controlling the consistency-destroying potential of concurrent access and failures. A general distributed architecture is assumed, together with crash failure semantics for the processors and performance failure semantics for the network.

To support this type of application, Argus provides a programming model based on objects and actions. Objects in Argus are dynamically-created entities called *guardians*, which are the units of distribution that encapsulate data and export *handlers* that can be invoked from within other guardians to manipulate the data. Guardians also contain processes; these are created to execute incoming handler invocations, or are *background* processes that operate independently of invocations. Data within a guardian can be declared to be *stable*, in which case it is stored on stable storage to facilitate recovery in the event of a processor crash; data not declared to be stable is stored in volatile storage, and is therefore lost should a failure occur. In addition, there are provisions for specifying recovery code to be executed upon restart of a guardian following failure.

Argus also supports actions, which are computations that exhibit the serializability and recoverability properties described in Section 7. More precisely, an action preserves these properties for any built-in or user-defined *atomic object* that it manipulates during execution; these objects—for example, atomic arrays or records—typically contain the shared data that is being accessed concurrently by multiple actions. Actions are denoted syntactically within the code of background processes of guardians, and cross machine boundaries by using RPC to invoke the handlers of other guardians. *Nested* or *subactions* are also supported [Mos85]. These are used both to increase concurrency and to limit the amount of computation lost due to failure; the former comes from provisions for concurrent spawning of multiple subactions, while the latter follows from the property that a failure during the execution of a subaction only aborts the subaction and not the entire top-level action. The call to a handler automatically runs as a subaction, which gives it the semantics of At Most Once RPC.

Of the abstractions described above, those that are used in the implementation of Argus include atomic actions, RPC, and stable storage. For atomic actions, two-phase locking is used to realize serializability, with shadow updating of stable data and the two-phase commit being the basis for recoverability. As already mentioned, the RPC in Argus implements At Most Once semantics, and also also handles the detection and elimination of orphans. Stable storage is used to store data declared to be stable by the user and in the implementation of the two-phase commit protocol. The dependency structure is similar to that shown in Figure 3.

10.3 Consul

Consul is a collection of protocols developed at The University of Arizona for implementing fault-tolerant distributed programs based on the replicated state machine approach. As such, it provides support for consistently ordering input messages submitted to the state machine, for maintaining a consistent view of group membership despite process failure and recovery, and for reestablishing a consistent state for a process upon recovery. At the heart of Consul is Psync, a multicast protocol that maintains the partial (or causal) ordering of messages exchanged among replicas in the form of an explicit graph that is made available to the application and other protocols [PBS89]. This graph is replicated by Psync on all processors on which participating processes reside. Consul is implemented in the *x*-kernel, an operating system kernel designed to facilitate the implementation of communication protocols [HP91]. This platform makes Consul highly-configurable in the sense that it is simple to construct an instance of the system oriented towards a particular application given a preexisting library of protocols. The failure model assumed corresponds to performance failures.

Of the abstractions described in previous sections, Consul includes multicast, time, membership, resilient processes, and stable storage. For multicast, the basic functionality plus consistent partial ordering are realized directly by Psync; additional protocols that use Psync's message graph give semantic-dependent and total ordering. The time service is also provided by Psync in the form of logical clocks. Membership is implemented by two protocols, one that does failure detection and another that realizes agreement on group membership in the event of failure or recovery. The state machine replicas are resilient processes that can use a combination of checkpointing and message logging to recover following a failure; of special interest here is that the messages need not be logged explicitly to stable storage since the replication of the message graph in the volatile memory of multiple processors by Psync automatically implements similar functionality.

In Consul, the dependencies are as follows. The multicast service depends on the time service to provide the various types of consistent message orderings. Membership and multicast, in turn, depend on each other; membership uses the partial order provided by the multicast in its agreement algorithms, while multicast uses membership to establish total ordering. Finally, resilient processes depend on the membership service to consistently incorporate the recovering process into the collection of replicas, and on stable storage to do checkpointing. [5]

10.4 Delta-4

The Delta-4 project is a European effort whose goal is to define a general system architecture for dependable, distributed systems. This project, which began in 1986 and involves both industrial and academic organizations, has addressed a number of different problems ranging from communication protocols and application support environments to protocol validation, dependability modeling, and fault-injection. The applications targeted by the project are correspondingly broad, and include process control, computer integrated manufacturing, distributed databases, transaction processing, and scientific computation. The project officially ended in 1991, although work continues on various aspects under follow-on projects.

Two complementary architectures have been developed during the course of the project: an Open System Architecture (OSA) and an Extra Performance Architecture (XPA). As its name implies, the OSA is intended to work in an open environment in the sense that it conforms to Open Systems Interconnection (OSI) reference models and, hence, is able to work with heterogeneous architectures, proprietary operating systems, etc. It also supports the widest variety of applications and fault-tolerance techniques. The XPA, on the other hand, is an architecture designed explicitly for constructing fault-tolerant distributed programs that also have critical real-time requirements. The two share the same design philosophy and overall structure.

The Delta-4 hardware system model consists of a number of *host* machines connected by a communication network. Interposed between each host and the network is a specialized processor called a Network Attachment Controller (NAC) that executes the communications software. The failure model assumed for hosts in the OSA can be either crash failures or arbitrary failures, with the latter naturally requiring more expensive protocols and fault-tolerance techniques; the failure model for hosts in the XPA is crash failures. In both architectures, the NACs are assumed to suffer only crash failures

Delta-4 supports a number of different software structuring paradigms to achieve fault-tolerance. These include the state-machine approach, primary/backup, and a variant of the state-machine called *leader/follower* in which replicas remain synchronized, but a single process is deemed responsible for making all decisions that affect replica determinism, such as message receipt order and preemption times. [6] The fundamental support for all of these techniques comes from two different multicast protocols executed by the NAC. The first, called AMp (Atomic Multicast protocol), is used in the OSA; it provides atomicity, total ordering, and termination. The second, called xAMp, is used by the XPA; it extends AMp to provide a range of services ranging from unreliable datagrams to reliable (unordered) multicast to full atomic multicast [RV91]. xAMp also includes a membership service in the form of a Group Management layer.

In terms of the abstractions defined in earlier sections, Delta-4 includes multicast, membership, and a time service. As already mentioned, the more general xAMp provides a variety of services, including all of the possible message orderings described in Section 4. Membership allows processes to join and leave process groups, as well as handling the detection and removal of failed processes. Two time services are provided, one implemented by synchronizing physical clocks and the other by logical clocks; the former is intended primarily for use by applications, while the latter is used in the multicast

[5] A rudimentary single-processor atomic action service is implemented as part of stable storage, so it can be argued that there is also a dependency to atomic actions similar to that in Figure 3.

[6] This approach can be viewed as a generalization of the use of a funnel process described in Section 4.

services for messages orderings [VR91]. The dependency structure generally follows that shown in Figure 3, although the exact relationship between membership and multicast is not entirely clear.

10.5 ISIS

ISIS is a toolkit developed at Cornell University to support the construction of distributed applications, including those that have fault-tolerance requirements. The software is oriented around a programming model based on *virtually synchronous process groups*, which are groups in which member processes see a consistently-ordered stream of events such as multicast messages, group membership changes, and failure notifications. This property makes the distributed system appear to be synchronous to the user, and hence, easier to program as compared with a system in which events occur asynchronously. [7] This programming model is most attuned to the state machine approach for constructing fault-tolerant programs, but experience has also shown it to be useful for structuring distributed applications in other ways [BC91]. The toolkit is currently implemented as a library on top of UNIX, with plans underway to integrate key features into the Mach operating system at a lower level to improve performance.

The fundamental fault-tolerant abstraction provided by ISIS is a multicast service made up of a collection of different protocols. The most important of these is CBCAST ("Causal Broadcast"), which provides delivery atomicity and partial ordering of messages; among its unique features is that it preserves causal ordering among multiple groups that have overlapping memberships. An atomic broadcast protocol called ABCAST, which extends CBCAST to a consistent total ordering, is also provided; this protocol uses the funnel process approach described in Section 4. Other services found within ISIS include a membership service, which is used to agree on when events corresponding to the leaving and joining of processes occur, and a time service based on logical clocks.

Dependencies among services that can be identified in ISIS include the following. The multicast service depends on the time service to provide both partial and total ordering of messages, and on the membership service to manage the failure of the funnel process used for total ordering. Membership also depends on the multicast service. In particular, the instances of the membership protocol executing on different machines use a variant of multicast to communicate among themselves; in this protocol, messages are delivered atomically but with no consistent ordering guarantee.

10.6 MARS

MARS (MAintainable Real-time System) is a system being developed at Institut für Technische Informatik, Vienna for use with distributed process control applications such as steel rolling mills and railroads. In this application area, the control system must meet stringent real-time guarantees, in addition to being functionally correct and fault-tolerant. The system model used in MARS is hierarchical: closely-cooperating machines are connected by a bus into *clusters*, which are themselves interconnected. Unlike many of the other projects, MARS is a combined hardware and software effort. Custom hardware that has been developed include the bus controller chip used for intracluster communication and a clock synchronization unit.

The general software organization can be characterized as a *periodic system* in which components are initiated periodically at predetermined intervals to handle external events that may have occurred since the last time they were executed. The computation required to respond to a given event potentially involves processes on several machines within a cluster; these computations are termed *real-time transactions*, although fault-tolerance is handled by replication, implying that the computing model has

[7] The meanings of synchronous and asynchronous in this case are somewhat different than the way in which they are used in Section 2.

more in common with the state machine approach than a pure object/action model. The failure model corresponds to crash failures; explicit assumptions about the number of possible concurrent failures are also made.

As might be expected given its periodic nature, the various fault-tolerant services are organized along the lines of a clock-driven system. Accordingly, a time service lies at the heart of the system; this service is realized by the hardware clock synchronization units alluded to above, which synchronize the clocks relative to an external time source. The fundamental multicast service for communicating within a cluster is based on a TDMA (Time-Division Multiple-Access) approach in which the bus is conceptually divided into slots along the time dimension and machines are allocated time slots using a round-robin scheduling algorithm. Each message is sent multiple time in succession to deal with the potential for lost messages. The combination of these strategies gives a service that realizes total order and termination based on the definitions given in Section 4.

MARS also has a membership service, which is implemented by logically including a machine's current membership set in every message. Interestingly, the effect of combining this service with the basic multicast is a multicast with atomicity, since any machine not receiving a message—and there can be at most one due to their failure assumptions—is removed from the membership set on the next TDMA cycle. In addition, the membership service is used in turn to construct an enhanced multicast service, which the system's developers term an atomic broadcast. However, the atomicity realized by this service is somewhat different than that defined in Section 4; in particular, the sender specifies a collection of destination machines, and the multicast succeeds only if that set is a subset of the membership at the time the multicast originates. Thus, if one of the machines in the destination set has failed, all receiving machines will discard the message.

The service dependencies in MARS are those implied by the above discussion. The basic multicast service depends on the time service, while membership depends on multicast. The MARS atomic multicast then depends on membership.

11 Summary

In this paper, we have described a number of fault-tolerant services that simplify the task of constructing dependable distributed systems. The usefulness of these system abstractions comes in a general sense from the support they provide for implementing the various programming paradigms that have been developed for building this type of system, including the object/action model, the primary/backup approach, the replicated state machine approach, and conversations. Some of these abstractions, such as stable storage, atomic actions, resilient processes, and RPC, can be viewed as analogous in many ways to standard functions, but with improved semantics when failures occur. Others, such as common global time, multicast, and membership, are more oriented towards making consistent information available to all machines in a distributed system despite the lack of physically shared memory. The specific algorithms used to realize these services are sensitive to the programming paradigm being supported and the failure model that is assumed. There are also significant differences based on the overall organizational strategy used for the system, with the two most common being termed clock-driven and clockless. In the literature, the former have often been referred to as synchronous and the latter as asynchronous, although we attempted to argue that, strictly-speaking, all systems are actually based on some type of synchrony assumption.

In addition to outlining key properties of these abstractions in isolation, different ways in which fault-tolerant services interact have been identified. This was done in two ways. First, we argued that certain interactions are fundamental when considering the properties of the various abstractions, which led to a hierarchy of common dependencies. Second, we described the way in which these abstractions are actually organized and interact in various fault-tolerant systems, specifically AAS, Argus, Consul, Delta-4, ISIS, and MARS. While by no means an exhaustive survey, these systems represent in many ways the current state-of-the-art in dependable distributed systems.

Finally, we note that the programming paradigms and fault-tolerant services described in this paper by no means close the book on the problems associated with constructing dependable distributed systems. Further work is needed in many areas, ranging from designing additional techniques for implementing existing abstractions, to improving our understanding of dependencies between abstractions, to developing new and better programming paradigms and supporting abstractions. All of these have the potential to help meet the increasingly inevitable need for computer systems that can be relied on provide dependable service.

Acknowledgments

We would like to thank D. Bakken, F. Cristian, M. Hiltunen, P. Homer, and V. Thomas for reading earlier versions of this paper and providing valuable feedback. Special thanks also to V. Thomas for preparing the figures and for contributing to the discussions on dependencies that led to the material in Section 9. This work has been supported in part by NSF Grant CCR-9003161 and ONR Grant N00014-91J-1015.

References

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of Second International Conference on Software Engineering*, pages 627–644, Oct 1976.

[AFdR80] K. Apt, N. Francez, and W. de Roever. A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, Jul 1980.

[Bad79] D. Z. Badal. Correctness of concurrency control and implications in distributed databases. In *Proceedings of IEEE COMPSAC Conference*, pages 588–593, Nov 1979.

[BALL90] B. Bershad, T. Anderson, E. Lazokska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 6(1):37–55, Feb 1990.

[BC91] K. Birman and R. Cooper. The ISIS project: Real experience with a fault-tolerant programming system. *Operating Systems Review*, 25(2):103—107, Apr 1991.

[BD85] O. Babaoglu and R. Drummond. Streets of Byzantine: Network architectures for fast reliable broadcast. *IEEE Transactions on Software Engineering*, SE-11(6):546–554, Jun 1985.

[BDD+89] R. A. Benel, R. D. Dancey, J. D. Dehn, J.C. Gutmann, and D.M. Smith. Advanced automation system design. *Proceedings of the IEEE*, 77(11):1653–1660, Nov 1989.

[BFG+85] J.S. Banino, J.C. Fabre, M. Guillemont, G. Morisset, and M. Rozier. Some fault-tolerant aspects of the Chorus distributed system. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 430–437, May 1985.

[BG81] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, Jun 1981.

[BGL83] P. A. Bernstein, N. Goodman, and M. Y. Lai. Analyzing concurrency control when user and system operations differ. *IEEE Transactions on Software Engineering*, SE-9(3):233–239, May 1983.

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

[BJ87] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb 1987.

[BL88] B. Bhargava and S. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems — An optimistic approach. In *Proceedings of the Seventh Symposium on Reliable Distributed Computing*, pages 3–12, Columbus, Ohio, Oct 1988.

[BMD91] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-tolerant computing. Technical Report 91-40, Department of Computer Sciences, University of Texas at Austin, 1991.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb 1984.

[Bor81] A. Borr. Transaction monitoring in ENCOMPASS: Reliable distributed transaction processing. In *Proceedings of International Conference on Very Large Data Bases*, Sep 1981.

[BS83] G. Barigazzi and L. Strigini. Application-transparent setting of recovery points. In *Proceedings of the Thirteenth Symposium on Fault Tolerant Computing*, Jun 1983.

[BSR80] P. A. Bernstein, D. W. Shipman, and J. B. Jr. Rothnie. Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 5(1):18–25, Mar 1980.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.

[BSW79] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3):203–216, May 1979.

[Cas81] M. A. Casanova. The concurrency control problem of database systems. In *Lecture Notes in Computer Science 116*. Springer-Verlag, Berlin, 1981.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[CC85] B. Chor and B. A. Coan. A simple and efficient Byzantine agreement algorithm. *IEEE Transactions on Software Engineering*, SE-11(6):531–539, Jun 1985.

[CDD90] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in the Advanced Automation System. In *Proceedings of the Twentieth Symposium on Fault-Tolerant Computing*, pages 6–17, Newcastle-upon-Tyne, UK, Jun 1990.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, Jun 1971.

[CGR88] R.F. Cmelik, N.H. Gehani, and W. D. Roome. Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs. In *Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing*, pages 55–61, Tokyo, June 1988.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.

[Coo85] E. C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 63–78, Orcas Island, WA, 1985.

[Cou81] Courier. Courier: The remote procedure call protocol. Technical Report XSIS 038112, Xerox System Integration Standard, Stamford, CT, Dec 1981.

[Cri84] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, SE-10(2):163–174, Mar 1984.

[Cri88] F. Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Proceedings of the Eighteenth International Conference on Fault-tolerant Computing*, pages 206–211, Tokyo, Jun 1988.

[Cri89] F. Cristian. Probabilistic clock synchronization. In *Proceedings of the Ninth International Symposium on Distributed Computing Systems*, pages 288–296, Newport Beach, CA, Jun 1989.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.

[DDS83] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. In *Proceedings of 24th Annual Symposium on Foundations of Computer Science*, Tucson, AZ, Nov 1983.

[DHS86] D. Dolev, J. Y. Halpern, and R. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Science*, 32(2):230–250, 1986.

[DHW88] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/c++. *IEEE Computer*, 21(12):57–69, Dec 1988.

[Dij68] E. W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[DIW89] S. Davidson, L. Insup, and V. Wolfe. A protocol for timed atomic commitment. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 199–206, Newport Beach, CA, Jun 1989.

[DS83] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal of Computing*, 12:656–666, Nov 1983.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, Nov 1976.

[EL90] P. D. Ezhilchelvan and R. Lemos. A robust group membership algorithm for distributed real-time system. In *Proceedings of the Eleventh Real-Time Systems Symposium*, pages 173–179, Lake Buena Vista, Florida, Dec 1990.

[Fid88] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Eleventh Australian Computer Science Conference*, 1988.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[GMS91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, Aug 1991.

[Goo75] J. Goodenough. Exception handling, issues, and a proposed notation. *Communications of the ACM*, 18(12):683–696, Dec 1975.

[Gra78] J. Gray. Operating systems: An advanced course. In *Lecture Notes in Computer Science 60*, pages 393–481. Springer-Verlag, Berlin, 1978.

[Gra86] J. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, Jun 1986.

[GS80] V. D. Gligor and S. H. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, SE-6(5):435–440, Sep 1980.

[Had82] V. Hadzilacos. An algorithm for minimizing rollback cost. In *Proceedings of the First ACM Symposium on Principles of Distributed Computing*, pages 93–97, Ottawa, Canada, 1982.

[Hol72] R. C. Holt. Some deadlock properties in computer systems. *ACM Computing Surveys*, 4(3):179–196, Sep 1972.

[HP91] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[HS80] M. Hammer and D. Shipman. Reliability mechanisms for SDD-1. *ACM Transactions on Data Base Systems*, Dec 1980.

[HSSD84] J. Y. Halpern, B. Simons, R. Strong, and D. Dolev. Fault-tolerant clock synchronization. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 89–102, Vancouver, Canada, Aug 1984.

[HY86] T. Hadzilacos and M. Yannakakis. Deleting completed transactions. In *Proceedings of Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 43–47, Cambridge, MA, Mar 1986.

[IM84] H. Ihara and M. Mori. Autonomous decentralized computer control systems. *IEEE Computer*, 17(8):57–66, Aug 1984.

[JZ87] D. Johnson and W. Zwaenopoel. Sender based message logging. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, pages 14–19, Pittsburgh, PA, Jun 1987.

[JZ90] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, pages 462–491, 1990.

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, Feb 1989.

[KGR91] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avizienis and J.C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 411–429. Springer-Verlag, Wien, 1991.

[Kim78] K. H. Kim. An approach to program-transparent coordination of recovering parallel processes and its efficient implementation rules. In *Proceedings of 1978 International Conference on Parallel Processing*, Aug 1978.

[KM85] H. Kopetz and W. Merker. The architecture of MARS. In *Proceedings of the Fifteenth Symposium on Fault-Tolerant Computing*, pages 274–279, Ann Arbor, Mi, Jun 1985.

[KO87] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed, real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, Aug 1987.

[Koh81] W. H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys*, 13(2):149–183, Jun 1981.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, Jun 1981.

[KT87] R. Koo and S. Toueg. Checkpointing and rolback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Jan 1987.

[KTHB89] M. F. Kaashoek, A. Tanenbaum, S. F. Hummel, and H. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, Oct 1989.

[KYA86] K. H. Kim, J. h. You, and A. Abouelnaga. A scheme for coordinated execution of independently designed recoverable distributed processes. In *Proceedings of Sixteenth Symposium on Fault Tolerant Computing*, Jun 1986.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, Jul 1978.

[Lam81] B. Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

[Lap92] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Vienna, 1992.

[LB89] P. Leu and B. Bhargava. A model for concurrent checkpointing and recovery using transactions. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 423–430, Newport Beach, California, Jun 1989.

[LeL78] G. LeLann. Algorithms for distributed data-sharing which use tickets. In *Proceedings of Third Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 259–272, Berkeley, CA, Aug 1978.

[LG81] G. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica*, 15:281–302, 1981.

[LG85] K. J. Lin and J. D. Gannon. Atomic remote procedure call. *IEEE Transactions on Software Engineering*, SE-11(10):1126–1135, Oct 1985.

[Lis85] B. Liskov. The Argus language and system. In M. Paul and H.J. Siegert, editors, *Distributed Systems: Methods and Tools for Specification, Lecture Notes in Computer Science, Volume 190*, chapter 7, pages 343–430. Springer-Verlag, Berlin, 1985.

[Lis88] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, Mar 1988.

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, Jan 1985.

[LS76] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. Technical report, Computer Science Laboratory, Xerox, Palo Alto Research Center, Palo Alto, CA, 1976.

[LS83] B. Liskov and R. W. Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, Jul 1983.

[LSM82] L. Lamport, R. Shostak, and Pease M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, Jul 1982.

[LW85] R. J. LeBlanc and C. T. Wilkes. Systems programming with objects and actions. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 132–139, Denver, Colorado, May 1985.

[LWL88] J. Lundelius-Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

[Mar76] P. M. Marci. Deadlock detection and resolution in a CODASYL based data management system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 45–49, Washington, D.C., Jun 1976.

[Mar84] K. Marzullo. *Maintaining the Time in a Distributed System*. PhD thesis, Stanford University, Department of Electrical Engineering, Mar 1984.

[Mat89] F. Mattern. Time and global states in distributed system. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, North-Holland, 1989.

[Mil79] M. Milenkovic. Update synchronization in multiaccess database systems. Technical Report PhD dissertation, Dept of Electrical & Computer Engineering, University of Massachusetts, Amherst, MA, 1979.

[ML83] C Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transaction. In *Proceedings of the Second ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug 1983.

[MLO86] C. Mohan, B. Lindsay, and R. Obermarck. Transactions management in the R* distributed database management system. *ACM Transaction on Database Systems*, 11(4):378–396, Dec 1986.

[Mos85] J. E. B Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, 1985.

[MPS89] S. Mishra, L. Peterson, and R. Schlichting. Implementing replicated objects using Psync. In *Proceedings of the Eighth Symposium on Reliable Distributed Computing*, pages 42–52, Seattle, Washington, Oct 1989.

[MPS91] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report TR 91-32, Dept of Computer Science, University of Arizona, Tucson, AZ, 1991.

[MPS92] S. Mishra, L. Peterson, and R. Schlichting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, Wien, 1992.

[MR79] D. A. Menasce and Muntz R. R. Locking and deadlock detection in distributed databases. *IEEE Transactions on Software Engineering*, SE-5(3):195–201, May 1979.

[MSF83] C Mohan, R. Strong, and S. Finkelstein. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In *Proceedings of the Second ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug 1983.

[MSM89] P. M. Melliar-Smith and L. E. Moser. Fault-tolerant distributed systems based on broadcast communication. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 129–134, Newport Beach, CA, Jun 1989.

[NCN88] S. Navaratnam, S. Chanson, and G. Neufeld. Reliable group communication in distributed systems. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 439–446, San Jose, California, Jun 1988.

[Nel81] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[Neu91] P.G. Neumann. Illustrative risks to the public in the use of computer systems and related technology. *Software Engineering Notes*, 16(1):2–9, Jan 1991.

[Ng88] P. Ng. A commit protocol for checkpointing transactions. In *Proceedings of the Seventh Symposium on Reliable Distributed Computing*, pages 22–31, Columbus, Ohio, Oct 1988.

[NS89] T. P. Ng and S. Shi. Replicated transactions. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 474–480, Newport Beach, CA, Jun 1989.

[OOW91] M. Olsen, E. Oskiewicz, and J. Warne. A model for interface groups. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 98–107, Pisa, Italy, Sep 1991.

[Pap79] C. H. Papadimitriou. Serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct 1979.

[PBS89] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug 1989.

[Per85] K. J. Perry. Randomized Byzantine agreement. *IEEE Transactions on Software Engineering*, SE-11(6):539–545, Jun 1985.

[PS88] F. Panzieri and S. K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, SE-14(1):30–37, Jan 1988.

[PSB+88] D Powell, D. Seaton, G. Bonn, P. Verissimo, and F. Waeselynk. The Delta-4 approach to dependability in open distributed computing systems. In *Proceedings of the Eighteenth Symposium on Fault-Tolerant Computing*, Tokyo, Jun 1988.

[Rab76] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–39. Academic Press, 1976.

[Rab83] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of Twenty-fourth Annual Symposium on Foundations of Computer Science*, Tucson, AZ, Nov 1983.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, Jun 1975.

[RB91] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal, Quebec, Canada, Aug 1991.

[RC89] K. Ravindran and S. T. Chanson. Failure transparency in remote procedure calls. *IEEE Transactions on Computers*, 38(8):1173–1187, Aug 1989.

[RLT78] B. Randell, P.A. Lee, and P.C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–166, Jun 1978.

[RS88] P. Ramanathan and K. G. Shin. Checkpointing and rollback recovery in a distributed system using common time base. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 13–21, Columbus, OH, Oct 1988.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, Feb 1978.

[RSB90] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, Oct 1990.

[RSL78] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, Jun 1978.

[Rus80] D. L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, Mar 1980.

[RV91] L. Rodrigues and P. Verissimo. *x*AMp: A multi-primitive group communications service. Technical report, INESC, Lisboa, Portugal, Sep 1991.

[SB90] M. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 6(1):1–17, Feb 1990.

[SC90] J. W. Stamos and F. Cristian. A low-cost atomic commit protocol. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, pages 66–75, Oct 1990.

[Sch82] F. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, Apr 1982.

[Sch87] F. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Dept of Computer Science, Cornell University, Aug 1987.

[Sch90] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec 1990.

[SDP91] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, Jan 1991.

[Ske82a] D. Skeen. Nonblocking commit protocols. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 133–147, Orlando, FL, Jun 1982.

[Ske82b] D. Skeen. A quorum based commit protocol. In *Proceedings of Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, Berkeley, CA, Feb 1982.

[SL84] K. G. Shin and Y. H. Lee. Evaluation of recovery blocks used for checkpointing processes. *IEEE Transactions on Software Engineering*, SE-10(6):692–700, Nov 1984.

[SM77] R. M. Shapiro and R. E. Millstein. Reliability and fault recovery in distributed processing. In *Oceans'77 Conference Record, Vol II*, Los Angeles, 1977.

[SMR88] S. K. Shrivastava, L. V. Mancini, and B. Randell. On the duality of fault tolerant system structures. In J. Nehmer, editor, *Experiences with Distributed Systems*, volume 309. LNCS Springer-Verlag, 1988.

[SS83] R. Schlichting and F. Schneider. Fail-stop processors: An approach to designing fault tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, Aug 1983.

[SS84] R. Schlichting and F. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, Jul 1984.

[SSCA87] R. Strong, D. Skeen, F. Cristian, and H. Aghili. Handshake protocols. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 521–528, Berlin, Sep 1987.

[ST87] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, Jul 1987.

[Svo84] L. Svobodova. Resilient distributed computing. *IEEE Transactions on Software Engineering*, SE-10(3):257–268, May 1984.

[SW89] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proceedings of Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, Edmonton, Canada, Aug 1989.

[SY85] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug 1985.

[Tea91] ISA Project Core Team. ANSA: Assumptions, principles, and structures. In *Conference proceedings of Software Engineering Environments*, University College of Wales, Aberystwyth, Wales, Mar 1991.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, Jun 1979.

[TS84] Y. Tamir and C. H. Sequin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the Thirteenth International Conference on Parallel Processing*, Aug 1984.

[Ver90] P. Verrissimo. Real-time data management with clockless reliable broadcast protocols. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20–24, Houston, TX, Nov 1990.

[VM90] P. Verissimo and J. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, pages 54–63, Huntsville, AL, oct 1990.

[VR91] P. Verissimo and L. Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. Technical report, INESC, Lisboa, Portugal, Nov 1991.

[VRB89] P. Verissimo, L. Rodrigues, and M. Baptista. Amp: A highly parallel atomic multicast protocol. In *SIGCOMM'89*, pages 83–93, Austin, TX, Sep 1989.

[Wei89] W. Weihl. Using transactions in distributed applications. In Sape Mullender, editor, *Distributed Systems*, pages 215–235. Addison-Wesley Publishing Company, ACM Press, New York, New York, 1989.