Position Paper: Fault-Tolerance Support in Distributed Systems

Richard D. Schlichting

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

The general goal of our research program for a number of years has been to devise abstractions, mechanisms, and formal methods that facilitate the construction of fault-tolerant programs, especially those designed for execution on distributed systems. Most recently, we we have been pursuing on a collection of related projects a variety of areas, ranging from communication protocols to formal verification. In particular, we have been investigating the following specific topics:

- Operating system support for fault-tolerant, distributed programming.
- Programming language abstractions that facilitate construction of fault-tolerant, distributed systems.
- Verification of fault-tolerant programs.

Much of this research is directly applicable to the topic of the workshop, especially the issues of formal methods, appropriate programming paradigms, and packaging of fault-tolerance primitives. In the sections that follow, we briefly outline the nature of our investigations in these three areas. Relevant papers are cited following the description.

1. Operating System Support

Ensuring a consistent order on message delivery among a set of participants—that is, ensuring that each process sees broadcast messages in the same order—is widely recognized as fundamental to constructing certain types of fault-tolerant, distributed programs. We have designed a new communication abstraction targeted at the problem of maintaining a consistent ordering of messages in the presence of communication and processor failures. The abstraction, called a *conversation*, preserves the partial ordering of messages multicast among a collection of processes. The novelty of our work is that we give an efficient mechanism for explicitly encoding a distributed application's logical clock in a low-level IPC mechanism and making it directly available to the application. Because of the fundamental nature of this partial ordering of messages, conversations offer a general solution to a wide range of communication problems: they support elegant and efficient implementations of conventional communication paradigms, they allow applications to directly access information not made available by other mechanisms, and they facilitate recovery from processor failure.

We have implemented a prototype IPC mechanism, called Psync, that supports the conversation abstraction. Psync implements the abstraction with a *context graph* that maintains the partial ordering of messages. A copy of the context graph is replicated on multiple hosts using an optimistic algorithm that transmits the context in which a given message was sent (i.e., the set of messages that precede the message) only when that context is missing. Studies indicate that the performance of Psync is comparable to other low-level IPC mechanisms such as TCP.

Features have been designed into Psync to facilitate the construction of fault-tolerant applications. For example, there is a restart primitive that can be used by a process to initiate recovery following a failure; its execution reconstructs the local copy of the context graph and

notifies the other processes of the recovery. This primitive can also be used to reexecute the participant from its initial state or from an intermediate saved state. In this option, the sequence of messages received by the process upon reexecution will be identical to the sequence it originally received, thus facilitating reconstruction of an appropriate internal state. This functionality is similar to recovery techniques based on message-logging, although with the advantage of being an integrated part of the IPC mechanism rather than an additional system component.

These aspects of Psync form only the basic foundation needed for constructing fault-tolerant applications. In general, such programs are easier to structure if higher level abstractions are also provided. Since these abstractions are typically constructed above the Psync layer in our scheme, we refer to them as *application-level protocols*. Examples include atomic broadcast, membership protocols, and recovery protocols, as well as storage-based abstractions such as stable storage, checkpoints, and logs.

We have recently started to investigate issues related to the design and implementation of application-level protocols in the context of a replicated object application built on top of Psync. One specific question we are addressing involves the interrelationship of the various protocols and appropriate structuring techniques. As currently implemented, all of the protocols are essentially independent even though it is clear that some of them could be simplified if they could use other protocols as lower-level abstractions. For example, it appears that both the recovery and membership protocol could use some variant of an order protocol to provide the appropriate partial ordering of messages.

To achieve this kind of functionality, we are currently exploring ways to decompose protocols into smaller, more fundamental "micro-protocols." For example, atomic broadcast has two orthogonal aspects: ensuring a consistent message order and atomicity of delivery. If it is possible to construct separate protocols that guarantee each property individually, we may be able to come to a clearer understanding of the specific property or properties of atomic broadcast that are required for a given higher-level protocol. Or, as another example, determining a common denominator in the way different applications use transactions could shed light on the exact relationship between such applications and different transaction protocols.

The ability to decompose a protocol into its fundamental components also opens the way to other interesting research possibilities. One is the investigation of new protocols that compose the pieces in unique and interesting ways. For example, composing an atomicity micro-protocol together with a partial ordering micro-protocol might result in an interesting and useful variant of atomic broadcast that is inherently more efficient. Another is work on constructing a collection of micro-protocols so that an applications designer could, for instance, use different versions of a protocol depending on the nature of the application, the specific architecture being used, or the assumed fault model. For example, an atomicity micro-protocol assuming only fail-stop failures would be different (and simplier) than one in which arbitrary (i.e., Byzantine) failures are allowed. This work on composing protocols will be supported by the meta-protocol capabilities of the *x*-kernel, the operating system nucleus upon which Psync is implemented.

2. Programming Language Abstractions

The ease with which distributed system software can be implemented depends to a certain extent on the programming language being used and the particular abstractions it provides. For example, the use of conventional sequential languages such as C or Pascal augmented with message-passing primitives provides very little in the way of support for distributed programming in general, and practically no support for fault-tolerant programming. High-level distributed or concurrent programming languages such as Ada, Concurrent Euclid, Mesa, Modula-2, and SR are in many ways ideal for programming such systems, yet in general lack sufficient support for fault-tolerant programming. The problems associated with using existing languages to construct robust systems has been reinforced by the shortcomings we perceived while using SR to build a

prototype of the Saguaro distributed operating system.

Accordingly, we have been investigating the design and implementation of distributed programming languages specifically intended for building fault-tolerant, distributed systems. The emphasis on system software is significant: we are interested in determining what mix of language features are useful for *implementing* application-level abstractions such as transactions or replicated servers rather than in designing languages which themselves contain such features. To achieve this goal, we have been working on language-based approaches for constructing system software that can deal with the asynchronous nature of failures in a systematic manner. Our approach is based on the observation that the occurrence of failures can be considered as events (i.e., state transitions) caused spontaneously by the *adverse environment* of the system. With this view, the failure of a processor is treated logically within the software as a concurrent event that is generated and signaled in real-time by an underlying processor membership service that detects the failure. The interprocess synchronization and communication mechanism provided by the language (e.g., semaphores, condition variables, messages) is then used to wait for the occurrence of a failure signal and synchronize its activity with normal processing. The specific extensions we have proposed involve the failure and recovery of fail-stop processors and are designed for the SR distributed programming language. The approach can easily be generalized to other languages and to handle other types of events, however.

In this work, one of our stated objectives is to design the mechanisms to mesh with SR so as to have minimal impact on the existing language. That is, our goal is to show how existing distributed languages can be adapted for fault-tolerant computing, rather than to develop a new language. Our subsequent plans are to lift this restriction and to use the experience gained in extending SR to rethink the entire collection of abstractions supported by the language. One obvious goal in doing so is to develop specific suggestions for future versions of SR. However, a second, more important goal is to devise general guidelines for designing and implementing abstractions that will lead to better support for fault-tolerant system programming in the next generation of programming languages. Our plan is to investigate a variety of different abstractions with an eye towards determining which are appropriate to provide within the context of a system programming language.

3. Verification Techniques

We have recently been investigating formal methods for reasoning about fault-tolerant, distributed programs that have timing constraints. Our approach is based on applying two methods previously used in other specification and validation tasks in a unique way: temporal logic and fuzzy logic. Temporal logic—first-order logic augmented with operators that allow reasoning about sequences of execution states—has been applied successfully to the task of proving liveness properties of sequential and concurrent programs. On the other hand, fuzzy logic—a logic that allows for a degree of uncertainty within formal reasoning—has been proposed for reasoning about non-determinism, particularly in artificial intelligence and database applications. Our approach is to combine these two methods into a single system that supports reasoning about timing and reliability properties separately and in tandem. In particular, an "extended" temporal logic is used to reason about timing behavior, while a fuzzy logic "valuation" applied to such temporal logic formulae is used to reason about reliability. Our eventual objective is to allow properties such as "The actuator will be adjusted $\leq M$ time units after the sensors are read with probability .99" to be expressed and verified given certain underlying probability distributions.

Over the past year, we have been working on developing the extended temporal logic portion of the logic. This logic, called CSTL (Clock State Temporal Logic), is based on the addition of a time component to the standard temporal operators "henceforth" ($\Box$) and "eventually" ($\Diamond$) to allow reasoning about lower and upper bounds on execution times. Specifically, we define the following abbreviations

$\Diamond_{[j,k]}P \quad ::= \exists i : j \le i \le k : \bigcirc^i P$

$\Box_{[j,k]}P \quad ::= \forall i : j \le i \le k : \bigcirc^i P$

$\Delta_{[j,k]}P \quad ::= \neg\Diamond_{[0,j-1]}P \wedge \Diamond_{[j,k]}P$

$\nabla_{[j,k]}P \quad ::= \neg\Diamond_{[0,j-1]}P \wedge \Box_{[j,k]}P$

for any temporal formula P and natural numbers j and k. The abbreviations are read informally as follows.

$\Diamond_{[j,k]}P$: P is true in at least one of the states between the j^{th} state and the k^{th} state.

$\Box_{[j,k]}P$: P is true in all of the states between the j^{th} state and the k^{th} state.

$\Delta_{[j,k]}P$: P is not true in the first $j-1$ states and then is true in at least one of the states between the j^{th} state and the k^{th} state.

$\nabla_{[j,k]}P$: P is not true in the first $j-1$ states and then is true in all the states between the j^{th} state and the k^{th} state.

Given these abbreviations, we have developed a programming model and a proof theory for reasoning about the timing properties of distributed programs. Two programs abstracted from existing applications have served as examples to test the application of our approach.

To deal with failures and recoveries in a formal way, our current plan is to develop a logic based on CSTL that includes a fuzzy "valuation" or probability for all temporal formulae. The first step is to add a transition probability matrix to the model for CSTL. This matrix represents the probabilistic nature of state transitions, where, for example, the entry representing the transition from a state s_i to a distinguished state s_{fail} would be the probability of a processor failure while in in state s_i. The next step is to use these matrix entries and the notion of satisfiability of predicates to construct a fuzzy valuation for formulae in the logic. That is, a valuation based on satisfiability is defined for predicates and extended inductively to cover immediate assertions involving the standard logical connectives, "not," "or" and "and." A valuation based on the appropriate entry in the transition probability matrix is then defined for use with the "next" ($\bigcirc$) operator. Finally, the valuations of formulae involving the bounded "henceforth" ($\Box$), "eventually" ($\Diamond$), "delta" (Δ), and "grad" (∇) can then be constructed from the CSTL definitions for these operators and the fuzzy valuation for their components. Note, however, that there are many possible fuzzy valuations; this approach seems to model the properties we wish to specify and validate most successfully, and so will be used as the starting point for further investigations.

4. Relevant Papers

[Hay89] Hay, K., Manchanda, S., and Schlichting, R.D. A temporal logic for proving real-time properties of distributed programs. Technical Report 88-40a, Dept. of Computer Science, The University of Arizona, May 1989.

[Mish89] Mishra, S., Peterson, L., and Schlichting, R.D. Implementing fault-tolerant replicated objects using Psync. *Proc. 8th Symp. on Reliable Dist. Sys.*, Seattle (Oct. 1989), 42-52.

[Pete89] Peterson, L., Buchholz, N., and Schlichting, R.D. Preserving and using context information in interprocess communication. *ACM Trans. on Comp. Sys.* 7, 3 (Aug. 1989), 217-246.

[Schl89] Schlichting, R.D., Cristian, F. and Purdin, T. A linguistic approach to failure handling in distributed systems. *Proc. IFIP Int. Working Conf. on Dependable Computing for Critical Applications*, Santa Barbara, CA (Aug. 1989), 159-166.