



DISTRIBUTED COMPUTING

[DS HOME](#) | [ARCHIVES](#) | [ABOUT US](#) | [SUBSCRIBE](#) | [SEARCH](#) | [C](#)[Back to Article](#)

Fault-Tolerant Distributed Computing: Evolution and Issues

K. H. (Kane) Kim • DREAM Laboratory, University of California, Irvine

At this early phase of the 21st century, distributed computing (DC) is not just an advanced mode of computing; it is the main mode. Moreover, the technological advances that DC fosters—in terms of scale and complexity—are occurring faster than at any point in DC's 25-year history. From the beginning of the DC era, fault tolerance had been touted as one of the main advantages of shifting system design from centralized structures into more distributed structures. As the 21st century has begun, it seems fitting to reflect on the evolution of fault-tolerant DC technology.

This article aims to examine the major unresolved issues of the 20th century, especially those that the research community must address for fault-tolerant DC to advance to the next level. Some issues might not be sufficiently resolved by the end of this decade. One emerging trend is the prevalence of real-time applications and embedded systems. Thus, this article will also address the issues that must be resolved to advance the real-time, fault-tolerant DC field.

Liveliness of the fault-tolerant DC field

Hardware components' reliability has significantly improved over the past two decades. For example, by the mid-1990s, the reliability of desktop workstations and supporting file servers steadily improved, which diminished concerns about the reliability of non-real-time systems. However, general-purpose database management system (DBMS) vendors successfully incorporated the ability to protect the integrity of data despite component failure.

Mainstream computing

In addition to the data backup and relatively basic types of DBMS transaction¹ mechanisms, the software support needed for higher-coverage fault-tolerant computing—that is, the mode of computing in which a broader range of fault types and rates are tolerated—did not advance into maturity in the 20th century. In the 20th century, real-time applications remained a small segment of the computing industry and did not attract any serious attention from mainstream vendors.

The mainstream computing industry's concerns mainly entailed maintaining DBMS integrity. On the whole, most vendors settled for *clean abort* of transactions whenever errors in

intermediate computation occurred. Most vendors did not feel that additional execution overhead—plus the extra hardware and software costs involved in facilitating higher-coverage fault-tolerant computing—was worthwhile. If a company did attempt to incorporate these features, its products would not be competitive in the market at that time.

The mainstream computing industry is now becoming interested in higher-coverage fault-tolerant DC mainly because of the rapid growth of the Web server market and the resulting demand for high-availability Web servers. The rapid growth of real-time computing applications that started around the mid-1990s, especially the demand for computer-embedded intelligent devices, helped further motivate that interest.

The mainstream computing industry is trying to meet these demands by incorporating cost-effective fault-tolerant DC mechanisms largely within the framework of general-purpose hardware—including redundant arrays of inexpensive disks (RAIDs) and redundant switch-router complexes—and general-purpose operating system (OS) architectures.

Explosive growth

With the explosive growth of e-commerce, the Web server market appears to be becoming the most important market for major computing vendors. Some say that end users lose patience when Web sites take longer than eight seconds to show results. This means that such Web sites must be up all the time—that is, they must meet high-availability requirements—and they must respond quickly even when a large number of clients concurrently access them.

High-availability Web servers must be capable of doing more than cleanly aborting failed transactions. If a node crashes, the impacted ongoing transactions must be cleanly aborted and the server function of the crashed node must be resurrected in another node in a reasonable amount of time. The high-availability server approach seeks to achieve higher coverage in fault-tolerant DC than the conventional server model that relies solely on clean aborts. The former leads clients to experience disconnection from the server for shorter duration and incurs less application costs than the latter.

Moreover, when companies use wireless network components in Web server applications, the fault rate and the needs for fault tolerance mechanisms tend to become more significant. In real-time applications, cleanly aborting transactions is rarely an acceptable approach because a transaction abort typically leads to abandoning the application.

Real-time computing applications—such as video conferencing, voice over IP, factory automation, and defense applications—require even higher coverage in fault-tolerant DC than high-availability Web servers do. Attempts for automatic failed-transaction retries or concurrent-redundant tries are usually essential techniques to avoid losing any transaction.

The mainstream computing industry's interests in fault-tolerant DC technology has steadily advanced over the past two decades through the course of clean abort technologies, high-availability server technologies, and real-time recovery technologies. Overall, the degree of interests has fluctuated as changes have occurred in component failure rates, computing environment factors, costs of computing failures, and the costs of redundancy.

Advances in fault-tolerant computing

Spectacular advances in integrating logic components—the early building-blocks of computer systems—into a smaller number of modules have reduced concerns about reliability. In addition, the computer industry achieved significant advances in the late 1970s and early

1980s in producing specially hardened hardware modules that were attractive building-blocks of high-reliability computing systems in safety-critical applications. Representative examples of such hardened modules include hardened processor modules,² RAID systems, and error-detection and error-correction coding subunits.

The industry generally feels that these technologies have matured. But because standard general-purpose hardware modules have become so reliable and powerful, much of the mainstream industry has growing interest in using software techniques instead of specially hardened hardware components to realize acceptable levels of system reliability.

Fault detection and network surveillance

The computing industry established several approaches for fault detection in the 20th century, including timeout (enforced by using watch-dog timers), redundant execution to compare the results, error-detection and error-correction codes, and acceptance tests or reasonableness checks.^{3,4} The first three techniques have generally matured, while the last technique—reasonableness checking—has also been used extensively, although there is still much research that needs to be done on the subject.

Another important category of fault detection techniques is *network surveillance*, which is also called node-link membership maintenance.⁵ Network surveillance is basically a decentralized mode of detecting DC components' faulty or repaired status. This technique is aimed at minimizing the periods during which faulty components—for example, the processing nodes and communication links—are lurking in DC systems.

There are only a few techniques that are practical that also yield rigorous quantitative performance analysis. The important metric in this area is the detection latency bound. Researchers have demonstrated some techniques for RT network surveillance in bus-LAN based systems⁵⁻⁷ and some for use in point-to-point network based systems.⁸

Transaction

The database research community established the scheme for transaction structuring on the basis of atomicity and sphere of control.⁹ The basic transaction model targets properties such as atomicity, consistency, isolation, and durability in spite of component failure.¹ A transaction must end with either commitment of an update to the database—in case of a successful execution—or clean abortion in case of an execution failure. Once a database aborts a transaction, the client that requested the transaction might repeat the request or take an alternative course of action. The database must observe the atomicity and clean-abort rules regardless of whether the database is concentrated in one site or distributed over multiple sites.

Researchers have developed log-based schemes for efficiently aborting and committing transactions and have developed various schemes for concurrent multiple-transaction execution. The industry has also largely integrated the transaction scheme with disk-mirroring approaches and the use of RAIDs. In short, basic transaction technology has matured. While researchers have attempted to extend the basic transaction scheme,¹ the industry has mainly implemented only basic transaction schemes.

Checkpointing and recovery line

Researchers developed *rollback-retry*—also called *checkpointing recovery*—in the 1960s to increase the probability of successful completion of a sequential atomic real-time computation segment. Checkpointing takes a snapshot of the state of a computation and saves the snapshot in a safe storage device. In executing a database transaction, the act of

saving a snapshot is often called a *save-point establishment*.

In a system of interacting processes, each of which performs checkpointing at various execution points independent of other checkpointing processes, a process rollback could cause an avalanche of rollbacks of interacting processes. This phenomenon is called the *domino effect*.³ Numerous researchers have also studied how to make interacting processes establish checkpoints in a coordinated manner to prevent a domino effect.⁴

A *recovery line* for a process, for example, P1, is a set of checkpoints possessing the following property: Each checkpoint belongs to a different process that will not be crossed by a rollback of any process caused by P1's failure. Over the past 25 years, researchers have extensively studied the techniques for managing recovery lines, but it appears that practitioners have not yet accepted these techniques.

Replication

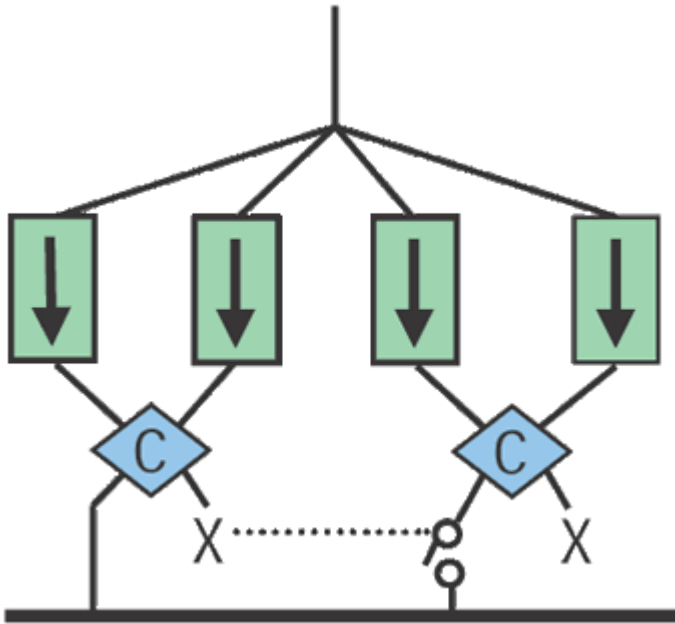
Researchers have studied database and process replication for the past three decades. Replicated databases, in which every transaction is executed on the replica designed as the primary and a subsequent update command is sent to other replicas, represent the simplest type of replicated databases.¹⁰ The falling costs of RAIDs have produced incentives for reducing the degree of replication. Researchers have studied other types of replicated databases extensively, but these techniques have not yet met with wide acceptance.

For process replication, the types of needs and the types of replication structures are more diverse. Early distributed computing research established approximately six basic types of replication structures.⁷ Other replication structures are variations of these. A combination of replicated processes and node facilities is called a *fault-tolerant computing station*, of which there are six basic types.

Comparing pair and rollback. Two replicas of a process run on two different nodes in parallel, and the system compares the results of their replicated task execution.² If a mismatch occurs, then the replicas roll back and make a retry.

Pair of self-checking processing nodes (PSP). There are two types of this structure. In the first type, a pair of single processor nodes uses an application-independent fault detection software component. Here, two replicas of a process run on two different nodes in parallel. The system validates computational results of task execution on each node by the lack of signals from the fault detection hardware and OS and by the execution of a common fault-detection component. A typical fault detection software component is one that does a consistency check on the data structures in the OS, as occurs in many telephone switching systems. In the second type of this structure, called the *pair of comparing pairs* (PCP), the replicas of a process run in parallel on four different nodes, organized as two pairs—each of which is in turn a comparing pair (as Figure 1 shows).

Figure 1. A pair-of-comparing-pairs station in which the replicas of a process run in parallel on four different nodes.



Distributed recovery block (DRB) station. The structure of the DRB station^{7,11} (as Figure 2 shows) is essentially a PSP station and one or both of the following two types of optional software components: an acceptance test that is an application-dependent fault-detection software component; or alternate application algorithms, also called try blocks. System designers can incorporate the acceptance test and try blocks using an elegant language construct called a *recovery block*.^{3,4} When developers use two try blocks, the operating rule is that the primary node tries to execute the primary try block whenever possible, whereas the shadow node tries to execute the alternate try block.

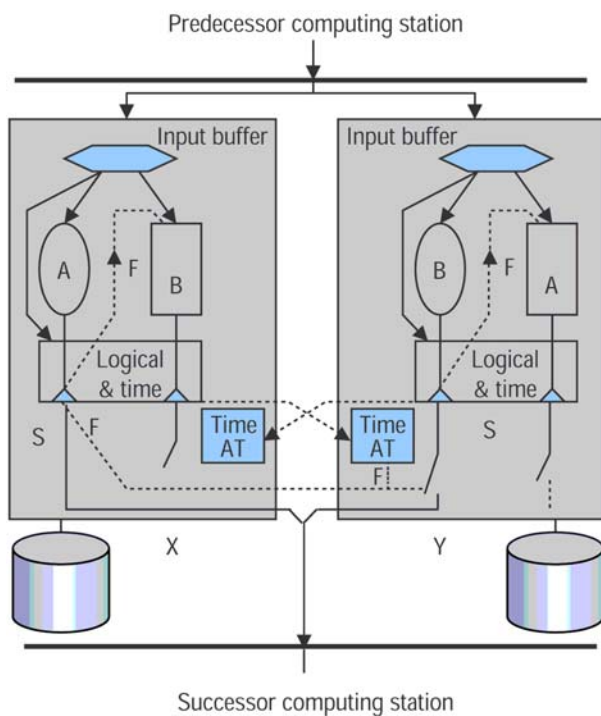


Figure 2. A distributed recovery block (DRB) station. This is a pair of single processor nodes station with optional acceptance test and try blocks.

In Figure 2, primary node *X* uses try block *A* as the initial first try block, whereas shadow node *Y* uses try block *B* as the initial first try block. The two nodes pick up the same input data and process the data in parallel using their current first try blocks. They then execute the same acceptance test to check if their computational results are reasonable. If the primary node passes the acceptance test, it performs an output of its results without delay.

This output action includes updating its local database and sending a data message to the successor computing station. If one of the two nodes fails before or at the acceptance test, the other node takes the role of the primary as soon as it discovers its partner's failure. Meanwhile, the failed node attempts to become a healthy shadow node without disturbing the new primary node; it attempts to roll back and retry with its second try block to bring its application computation state, including local database, up to date.

Researchers have developed approaches for using more than two try blocks or more than two processing nodes in a DRB station—and for using the same node-pair to form multiple virtual DRB stations.¹² The DRB scheme can accomplish forward recovery regardless of whether a node fails because of hardware faults or software faults. Furthermore, the DRB scheme does not require the two try blocks to produce identical results. The second try block does not need to be as sophisticated as the first try block. If software fault tolerance is not a design goal, the application task designer need not provide alternate algorithms or acceptance tests.

Voting triple modular redundancy (TMR) station. Three replicas of a process run on three different nodes, respectively, and they take a vote with their execution results.² When there is discrepancy, the system uses the result that receives a majority vote.

N-Version Programming station. As shown in Figure 3, the NVP station is essentially a TMR station plus three or more different application algorithms for each application task.¹³ While the voting logic is application-independent (a significant advantage), the scheme requires designing multiple versions that generate truly identical computation results. This could be a restriction in cases where task logic complexity is high.

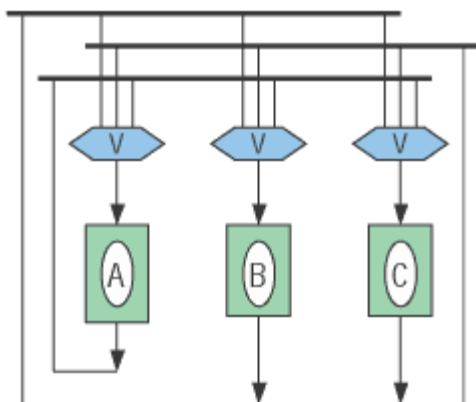


Figure 3. An *N*-version programming station.

Quantitative treatment

Fault tolerance in DC is realized through redundant computation which consumes resources. Effective resource allocation is thus impossible without quantitative characterizations of fault tolerance schemes. The efforts made by the fault tolerance research community in the 20th

century are grossly inadequate.

In the application environments in which clean abort is the recovery goal and high-availability servers are desirable, the most important metrics are fault types and rates tolerated; the extra hardware costs; and the extra time costs, including fault detection overhead, abortion time, and server down-time.

In the application environments in which real-time recovery is desirable, the most important metrics include fault types, rates tolerated, and recovery time, which is the maximum difference between a normal execution time for a task and the time for a task execution involving fault detection and recovery events. In some applications, extra hardware costs are also an equally important metric.

It is easy to suggest that fault tolerance approaches not yielding easy quantitative analyses are not safe to use and that using such approaches is a blind exercise. In light of this, the research community must emphasize the quantitative elements of fault-tolerant DC in this century. In addition, the software reliability problem is fundamentally one of obtaining analyzable software. Ideally, those in the industry must use analyzable node OS, analyzable middleware, and analyzable application software to realize reliable DC systems. Adding fault tolerance capabilities cannot be an excuse for violating this principle.

Fair and unfair fault-source modeling

A component of a DC system exhibits faulty behavior as a function of its internal organization, the reliability of its subcomponents, and its operating environment. Because a nontrivial component can exhibit faulty behavior in an almost unlimited number of ways, designers must have a manageable model of faulty component behavior. I call such component models *faulty behavior models* and a combination or an abstraction of a combination of the faulty behavior models of the components used in composing a DC system *fault-source models*.

Researchers have proposed and used numerous fault-source models in the 20th century.^{14,15} Unfortunately, more often than not, some research used subjective and unscientific reasoning in adopting and assessing the reasonableness of fault-source models. This reasoning often led to unproductive discussions in the research community. A good fault-source model must be a characterization of all non-negligible patterns of fault occurrences. Such models must be based on good faulty behavior models of the components used in the system.

In the research dealing with the analysis of fault-tolerant DC algorithms, researchers have most often modeled replaceable components as units of two extreme types: either an idealistic faulty behavior model or a malicious unit model. An idealistic faulty behavior model, the simplest of conceivable models, is the *fail-silent unit* model. An FSU can exhibit only absence of an explicit output when an internal fault occurs. Such units do not explicitly send out erroneous values. The model at the other extreme end is the *malicious unit* model, also called the Byzantine unit model. An MaU is capable not only of sending out erroneous values but also of sending out sequences of values as if they were carefully manufactured to cause troubles to monitoring and diagnosing units.

The FSU model should be taken as a design goal for each replaceable component. It is an entirely different matter whether it is appropriate to represent a component used in a real system as an FSU. It is often dangerous to treat existing components as FSUs when either building complex systems with them or validating complex systems containing them. The MaU model appears to be unbalanced. The probability of malicious behavior occurring in a

real system is much smaller than the probability of an assumption—adopted in conjunction with the MaU model—being violated. Typically an MaU model is accompanied by an assumption, such as “the number of nodes that can fail is limited to one third of the total number of nodes in the system.”

Given this, an ideal direction for advancing the state of the art in this important area is to develop a systematic method for fair distribution of concerns over possible occurrences of anomalous events. One useful technique is to lay out possible occurrences of all types of anomalous events in the space of occurrence probabilities and then choose the subset of the possible events to contend with.¹¹ Figure 4 shows such a view.

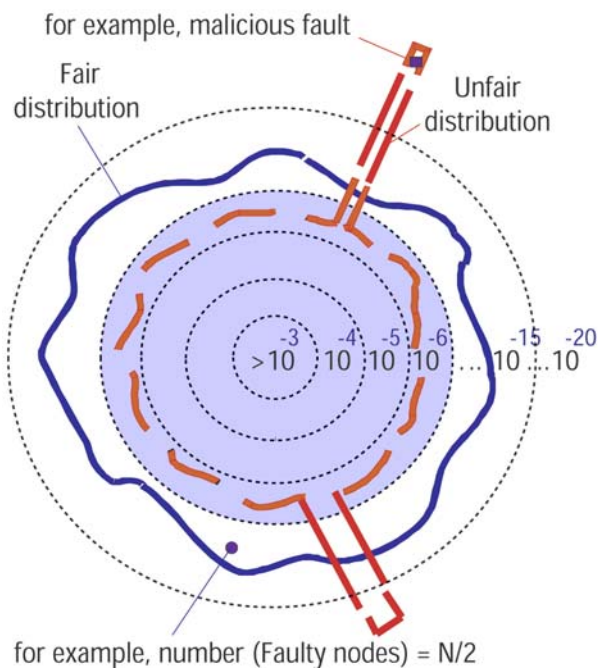


Figure 4. Fair distribution of concerns in which the space of occurrence probabilities can be divided into concentric rings.

The space of occurrence probabilities can be divided into concentric rings. Any event that has the occurrence probability of at least 10^{-3} is positioned within the innermost ring. Events that have occurrence probabilities between 10^{-4} and 10^{-3} must be positioned in the space between the innermost ring and the second innermost ring, and so forth.

Formulating a fault source model amounts to selecting a set of anomalous events that will be of concern. The way these selected events are spread in the space of occurrence probabilities determines the fairness or unfairness of the distribution of concerns. As illustrated by the solid rugged circle in Figure 4, it should be possible to draw a circular boundary in the space of occurrence probabilities that meet these fair distribution requirements:

- FDR1—all non-negligible events that the designer can envision are placed within the encircled space;
- FDR2—Events that are placed outside the circular boundary have occurrence probabilities that are more than several magnitudes of order—10 to 100 times—greater than the occurrence probability of any event placed within the encircled space.

Although fair distribution might sound like a natural thing to do, its effective practice requires considerable research. Among other things, more statistical data on various types of

component failures—including correlation among different component failures and correlation between internal faults of components and observed failures—must be obtained.

As mentioned earlier, the mainstream computing industry is now becoming better motivated to explore high-availability server technologies and real-time recovery technologies rather than just clean abort technologies. The demands for quantitative treatment of server down-times and recovery time bounds are now growing.

Real-time fault-tolerant DC

In real-time fault-tolerant DC systems, fault detection and recovery actions should ideally be executed in timely, effective manners so that intended output actions of real-time computations always take place on time and even in spite of fault occurrences. When this is not feasible, the system must attempt the fault tolerance actions that lead to the least damages to the application's mission and the system's users. To be useful in DC systems, a fault detection technique must at least yield a tightly bounded detection latency, and a recovery technique must at least yield a tightly bounded recovery time. Only a small fraction of the research conducted to date dealt with rigorous analysis of *detection latency bounds* and *recovery time bounds*.

Even in non-real-time systems, such as many Web server applications, detection latency and recovery time are becoming important performance metrics because they contribute to server down-time. However, in such environments, average down-times are more relevant than tight bounds on the down-time. In fact, there are signs that the major OS and communication infrastructure vendors are gradually increasing their efforts to make the timing behavior of their products more predictable. This will blur the boundary between fault-tolerant DC approaches used in non-real-time systems and those used in real-time systems.

Currently, the main challenge in development of the real-time fault-tolerant DC technology appears to be integration. There are two major types of integration that must be developed: real-time fault-tolerant computing stations and network surveillance and reconfiguration; and fault detection and replication principles and object-oriented real-time DC structuring techniques.

Researchers can use the six fundamental types of replication structures described earlier or their variations to construct real-time fault-tolerant computing stations, each dedicated to executing one or a few types of real-time tasks. Developers must mobilize not only these computing station construction techniques but also network surveillance techniques to construct real-time fault-tolerant DC systems cost effectively. They must also use techniques for fast reconfiguration, which includes functional amputation of faulty components and redistribution of tasks to existing, newly incorporated, and repaired nodes.

In the 20th century, developers coupled nearly all real-time fault tolerance techniques with the approaches for process structuring of real-time DC software. One of the several cutting-edge technology movements initiated in the 1990s in software engineering is real-time OO programming. The most important goal of that movement is to instigate a productivity jump in software engineering for real-time DC systems. The movement is still in its youth but has already started to have an impact. Its potential is now much more clearly and widely recognized than it was in mid-1990s.

Adapting the existing real-time fault tolerance techniques for integration into powerful real-time OO DC structures is an important integration issue. Another major issue in real-time

fault-tolerant DC is scalability. The complexity of real-time fault-tolerant DC systems is now growing more quickly than before at this early phase of the 21st century. Wide area network infrastructure is increasingly used in new applications. To cope with this trend, the research community must enhance the scalability of network surveillance techniques and recovery techniques.

Additionally, one fundamental approach in real-time DC—that only a tiny fraction of the computer science community has explored—is the time-based coordination of distributed actions.¹⁶ This approach can generate many effective techniques for real-time fault tolerance, including network surveillance, state consistency among replicas, and real-time fault-tolerant multicasts.

Reliable multicast

Researchers have worked on group communication in real-time DC systems for almost two decades, but it is not yet a mature technological field.^{16,17} Different research communities—such as the computer network research community and the fault-tolerance computing research community—have addressed this topic from different view points. The main challenge in establishing group communication protocols is dealing with possible fault occurrences.

Conversely, multicasts without the possibility of component failures are simple programming problems. It is therefore sensible to cast group communication protocols as distributed applications that are supported by execution engines using established real-time fault-tolerance techniques. Such engines must effectively handle failures of low-level components, such as processors, paths in communication, interconnection networks, processor-network interfaces, and OS components.

Challenging issues in implementing real-time fault-tolerant multicasts include ensuring that the sender and all receivers reach the same correct conclusion without excessive delay and that all the receivers correctly received the message. While some promising approaches have emerged in recent years, real-time fault-tolerant multicasts still appear to be a fresh research topic.

Object-oriented fault-tolerant DC

In recent years, the OO DC movement—including CORBA, Java-based DC, COM, and .Net—has become a major force. Most major industrial developers have joined the movement. Even the real-time OO DC branch of the movement has become significant. Justifications for process-structured DC have started losing strength. Fault-tolerant OO DC technology is becoming an active research field.^{18,19} The challenging research issue is to exploit intra-object concurrency while still enabling high-coverage fault-tolerant computing, such as real-time recovery.

Middleware mechanisms

Modern sizable DC systems consist of various subsystems that form multiple layers: application software, middleware, node OS kernel, and hardware. Developers can incorporate fault-tolerance mechanisms into some or all of these layers. Figure 5 shows the major types of applicable approaches in various layers. The figure also indicates that in terms of the technical difficulties of realizing real-time fault tolerance, the middleware-level and the application-software-level approaches are more difficult to implement than hardware

approaches and the kernel-level approaches. However, various cost factors put middleware approaches in a favorable position, which is why the middleware support for real-time fault-tolerant DC is becoming a popular area of research.

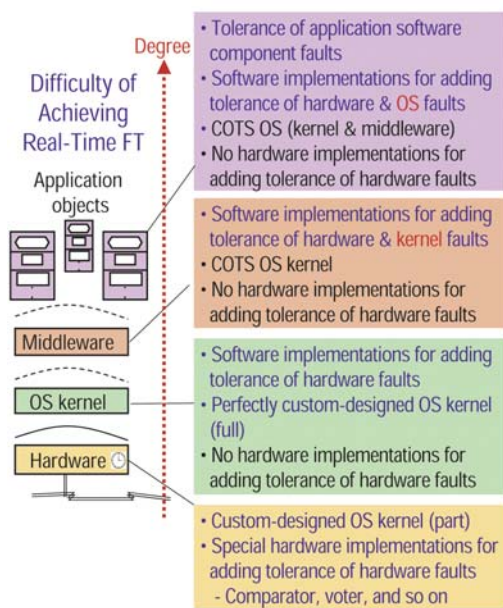


Figure 5. Approaches applicable in different layers and relative difficulties of creating real-time fault tolerance.

Developers pursue middleware approaches for two major reasons. First, such approaches can adapt to different hardware platforms. Second, middleware researchers assume that it will be some time before mainstream OS designers can produce efficient support mechanisms for fault-tolerant DC. Middleware approaches target tolerating not only hardware faults but also OS kernel faults. Middleware instantiations running on different nodes might cooperatively detect the crash of a certain node. They might even detect the crash of a certain hardware subsystem of a certain node, or a hardware node's temporary deafness. These deafness symptoms might be caused by hardware malfunctioning or OS kernel imperfections. Middleware implementations of various network surveillance schemes are therefore becoming an active field of research. Often, recovery actions taken by a cooperating group of middleware instantiations might be independent of whether faults originated from node hardware or OS kernels.

Middleware should also detect conditions, which, according to the parameters supplied by the application software, are errors. For example, the application software might notify the middleware that a call for a certain method in a remote object must be completed within 50 milliseconds under fault-free circumstances. The application software might also notify the middleware that upon detection of a certain error condition, a signal must be sent to a thread executing an object replica. Or the middleware might be told that objects 7 and 9 are replicas that will run fully independently, including independent delivery of results to receiving objects. The middleware must then compare the results from object 7 and 9 before letting each receiving object take the results.

Therefore, the middleware must obey the application-specific demands that come from the application software, which are related to detection of application-level errors and to actions devised to produce application-level recovery effects. Such services that can be made

available to DC application software, and sets of relevant APIs, are important research topics.

Software fault tolerance

Because software faults are design faults, using design redundancy is the logical choice for tackling the problem.⁴ Design faults are often found in both OS and application software. Although researchers formulated promising approaches—such as the recovery block scheme, the DRB scheme, and the NVP scheme—roughly 18 to 27 years ago, the industry still lacks an effective method for validating schemes aimed at handling design faults.

The main difficulty is creating realistic fault conditions. In fact, the research community has not yet fulfilled its mission of demonstrating the software fault-tolerance capability in convincing application contexts. Some research has successfully demonstrated the ability to detect symptoms of design faults at runtime, but these approaches do not include any successful real-time recovery actions other than stopping the system operation. Successfully accomplishing this goal will require long-term persistent research efforts. The DC system used must be considerably complex because, otherwise, the convincing cases of software fault occurrences are not likely to be encountered.

Because of the size and duration of any serious research effort needed for software fault tolerance, the size of the research community dealing with this area has been in a steady decline in the past 10 years. Yet the problems of software faults are real. No prudent manager responsible for constructing a large-scale safety-critical system wants to assume that software faults will not exhibit symptoms during the system's operational life. Hopefully, government agencies will begin to pay more attention to the growing neglect of software fault-tolerance research.

Although researchers in the 20th century built a basic technical foundation for fault-tolerant DC, major gaps in the established foundation must be closed in this new century. Future emphasis by the fault-tolerant DC research community on quantitative treatment of design techniques and protocols—including scientific assessment of fault source models—will lead to accelerated advances in fault-tolerant DC technologies.

Acknowledgments

Research work was supported in part by the US Defense Advanced Research Project Agency under Contract N66001-97-C-8516, monitored by SPAWAR and Contract F33615-01-C-1902 (NEST) monitored by AFRL; in part by the NSF Next-Generation Software (NGS) Program under Grant 99-75053; and in part by the NSF Information Technology Research (ITR) program under Grant 00-86147.

References

1. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufman Publishers, 1994.
2. W.N. Toy, "Fault-Tolerant Computing," *Advances in Computers*, vol. 26, 1987, pp. 201–279.
3. B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering*, IEEE CS Press, Los Alamitos, Calif., vol. SE-1, no. 2, June 1975, pp. 220–232.
4. B. Randell and J. Xu, "The Evolution of the Recovery Block Concept," *Software Fault*

- Tolerance*, John Wiley & Sons, New York, 1995, pp. 1–21.
5. H. Kopetz et al., "Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System," *Dependable Computing and Fault-Tolerant Systems*, Vol. 4, Springer Verlag, Berlin, 1989, pp. 167–174.
 6. H. Kopetz and G. Gruensteidl, "TTP: A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems," *Computer*, vol. 27, no. 1, 1994, pp. 14–23.
 7. K.H. Kim, "Action-Level Fault Tolerance," *Advances in Real-Time Systems*, Prentice Hall, Upper Saddle River, 1994, pp. 415–434.
 8. K.H. Kim and C. Subbaraman, "Dynamic Configuration Management in Reliable Distributed Real-Time Information Systems," *IEEE Trans. Knowledge and Data*, IEEE CS Press, Los Alamitos, Calif., vol. 11, Jan./Feb. 1999, pp. 239–254.
 9. C.T. Davies, "Recovery Semantics for a DB/DC System," *Proc. ACM Annual Conf.*, ACM Press, New York, 1973, pp. 136–141.
 10. B. Bhargava, *Concurrency Control and Reliability in Distributed Systems*, North-Holland Pub., Amsterdam, 1987.
 11. K.H. Kim, "Fair Modeling of Fault-Tolerant Distributed Systems," *Computer Comm.*, vol. 17, no. 10, Oct. 1994, pp. 699–707.
 12. K.H. Kim, "The Distributed Recovery Block Scheme," *Software Fault Tolerance*, 1995, pp. 189–209.
 13. A. Avizienis, M.R. Lyu, and W. Schutz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," *Proc. IEEE CS 18th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 15–22.
 14. R.B. Haverkort et al., *Performability Modeling: Techniques and Tools*, John Wiley & Sons, New York, 2001.
 15. P. Powell, "Failure Mode Assumptions and Assumption Coverage," *Proc. 22nd IEEE Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 386–395.
 16. H. Kopetz, *Real-Time Systems*, Kluwer, Dordrecht, the Netherlands, 1997.
 17. M. Mock, N. Edgar, and S. Schemmer, "Efficient Reliable Real-Time Group Communication for Wireless Local Area Networks," *Lecture Notes in Computer Science*, vol. 1667, Springer Verlag, Berlin, 1999, pp. 380–400.
 18. P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith, "Using Interceptors to Enhance CORBA," *Computer*, July 1999, pp. 62–68.
 19. J. Xu, A. Romanovsky, and B. Randell, "Concurrent Exception Handling and Resolution in Distributed Object Systems," *IEEE Trans. Parallel & Distributed Systems*, IEEE CS Press, Los Alamitos, Calif., vol. 11, no. 10, Oct. 2000, pp. 1019–1032.

K. H. (Kane) Kim is director of the DREAM Laboratory at the University of California, Irvine. His research interests include real-time object-oriented distributed programming and software engineering, real-time fault-tolerant computing, distributed and parallel real-time simulation, reliable Web service applications, and real-time embedded computing applications including multimedia and control applications. He received a PhD in Computer Science from the University of California, Berkeley in 1974. Contact him at ET544E, University of California, Irvine, CA 92697; khkim@uci.edu; <http://dream.eng.uci.edu>.



Feedback? Send comments to dsonline@computer.org.

This site and all contents (unless otherwise noted) are Copyright ©2002, Institute of Electrical & Electronics Engineers, Inc. All rights reserved.