Components for Software Fault Tolerance and Rejuvenation

Yennun Huang
Chandra M. R. Kintala
Lawrence Bernstein
Yi-Min Wang

Software fault tolerance is the task of detecting and recovering from failures that are not handled in the underlying hardware or operating system layers of an application. Software rejuvenation prevents failures by periodically, and gracefully, terminating an application and restarting it at a clean internal state. This paper describes five reusable software components that provide these capabilities. They perform automatic detection and restart of failed processes, checkpointing and recovery of data in memory, replication and synchronization of files, and software rejuvenation. These components, which have been ported to a number of UNIX* platforms, can be used in any application with minimal programming effort. The fault tolerance capabilities of several communication products and services in AT&T have been enhanced by incorporating these components. Experience with these products to date indicates that the components provide efficient, economical means to increase the level of fault tolerance in an application.

Introduction

In a telecommunications network, switching systems are known to require the highest degree of reliability—in other words, availability and data consistency. However, hundreds of other systems are needed to provision, process, operate, administer, and maintain a large, reliable telecommunications network and its services. If the network as a whole is to be reliable, each of its components must also be reliable. The challenge is to do this with reasonable cost and effort.[1]

Traditionally, reliability is provided through fault tolerance technology in the hardware, operating system, and database layers of the computer system executing the application software. Two trends emerging in the marketplace are changing this tradition of providing fault tolerance. First, standard commercial hardware and operating systems are becoming more reliable, distributed, and inexpensive. These items are now off-the-shelf commodities with open and evolving standards and interfaces. Second, the proportion of failures that can be attributed to faults in the application software is increasing along with the size and complexity of the software being deployed.[2] Failures that result from those software faults are discussed in depth below.

Transient Failures and Software Dynamics. Most software systems, especially telecommunications software products and services, are thoroughly verified and validated (V&V) before they are deployed in the field. However, the size and complexity of the software in these systems are increasing and becoming distributed in such a way that even the most advanced V&V tools and methods cannot remove all possible faults, thereby leaving behind residual faults.

In software, all faults are design and coding faults, and all are permanent faults. However, a failure exhibited by these faults can be *transient;* in other words, the failure may not recur if the software is reexecuted using the same input. During a program's

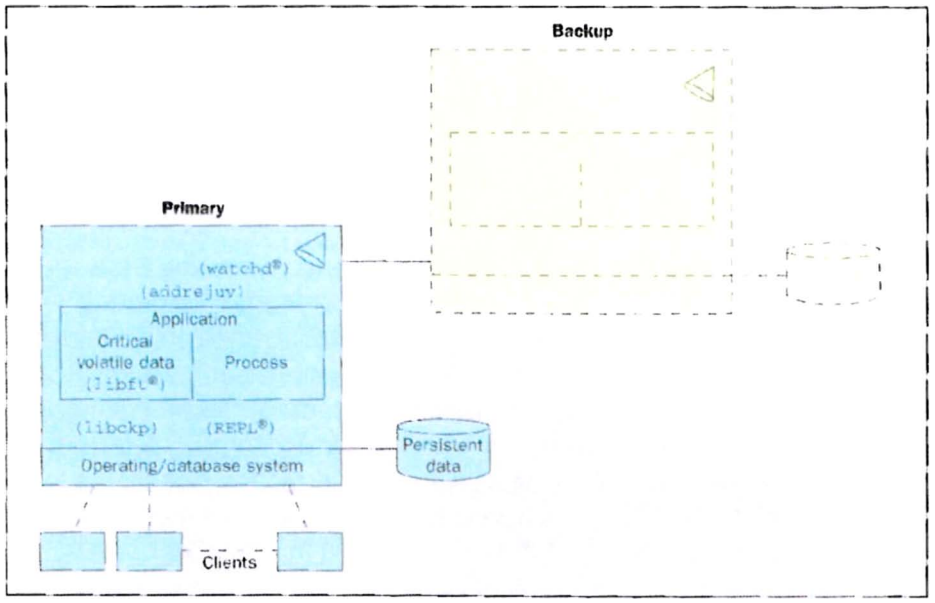

Figure 1. Software fault tolerance platform and components.

execution, its behavior, especially that of a distributed application, depends not only on the input data and message contents, but also on the timing and interleaving of messages, signals, and interrupts received from the resources. It also depends on shared variables and other "state" values in the operating environment of the application. As a result, residual faults are occasionally triggered, leading to transient failures, also called Heisen bugs.[2]

To understand software failures, the focus must be shifted from the programs themselves to programs in execution, or *processes*. This shift in focus is part of a new direction for software research—*software dynamics*. Software dynamics is the study of the dynamic behavior of software as it is being executed. The concluding section of this paper contains more general remarks on software dynamics.

Handling Transient Software Failures. This shift in focus leads to a collection of approaches for recovering from transient software failures[3] and providing increasingly higher levels of reliability. At the minimum, a mechanism to detect and restart failed processes should be in place. The next level is to *checkpoint*—that is, to make a copy of the application data—and recover the internal state of a process when it fails. In addition, messages may also be logged and replayed. During recovery and replay, some part of the environment may change, keeping the process from failing when it is reexecuted. The messages may also be reordered during replay, thereby masking errors caused by unexpected event sequences. In addition to the previous tasks, the next level is on-line replication of application files at a remote site.

The methods described are reactive in nature—that is, they attempt to recover the application after a failure has occurred. A complementary proactive approach, called *software rejuvenation*, is used to handle transient software failures. Software rejuvenation prevents failures from occurring by periodically, and gracefully, terminating an application and immediately restarting it at a clean internal state. An application is restarted by queuing the incoming messages, restarting the application processes at a predefined state, reinitializing the in-memory volatile data structures, and logging administrative records.[4]

A Platform. Individually implementing these software fault tolerance and rejuvenation tasks in each application requires expertise in reliability. It should not be done ad hoc. A middleware platform containing a set of reusable software components—`watchd`,® `libft`,® `REPL`,® `libckp`, and `addrejuv`—has been developed to perform those tasks, as shown in Figure 1.

The hardware platform on which these reusable software components are based is a network of standard computers. Each computer serves as a backup for another computer on the network. The components provide mechanisms to checkpoint, log messages, watch, detect, rollback, restart, recover from failures, and rejuvenate software processes to avoid failures.

Reusable components make it easy, efficient, and economical for software developers to embed high availability in an application. The components also provide flexibility in determining the underlying computing platform and the amount and level of fault tolerance in the application running on that platform. The sections that

follow describe these components, and the section entitled "Applications" discusses some applications that have increased their reliability using these components.

Watchd

`Watchd` is a watchdog daemon process that runs either on a single machine or on a network of machines to detect application process failures and machine crashes. It detects a process crash either by polling the process (using `kill(0,pid)`) or by receiving a `SIGCHLD` signal from the process.

`Watchd` determines whether a process is hung by using one of two methods. In the first method, `watchd` sends a "ping"-like command to the local application process using interprocess communication (IPC) facilities on the local node. It then checks the return value. If it cannot make the connection, it waits for the amount of time specified by the application, and tries again. If it fails after the second attempt, `watchd` interprets the failure to mean that the process is hung. In the second method, the application process periodically sends a heartbeat message to `watchd`, which checks the heartbeat. If the heartbeat message from the application is not received by a specified time, `watchd` assumes that the application is hung. `Libft`, discussed in the next section, provides a `hbeat()` function that enables applications to send heartbeats to `watchd`.

When `watchd` detects that an application process has crashed or is hung, it recovers that application at an initial internal state or at the last checkpointed state. The application process is recovered on the primary node, if that node has not crashed, or on the backup node, as specified in a configuration file. If `libft` is also used, `watchd` sets the restarted application to process all the logged messages from the log file generated by `libft`.

`Watchd` runs on each machine and watches one neighboring `watchd` in a circular fashion to detect machine failures. This method is similar to the one used in the adaptive distributed diagnosis algorithm.[5] If the `watchd` running on a machine fails to respond to a polling request from the neighboring `watchd`, a machine crash is assumed. When a machine failure is detected, the neighboring `watchd` can execute a user-defined recovery action, which may include migrating the application from the failed machine to another machine. After the machine is repaired, it can rejoin the network by simply starting the `watchd` daemon. To distinguish machine failures from communication link failures, although not definitively, `watchd` can use two communication links for polling a neighboring machine. It reports a machine failure only when it fails to contact a neighboring machine through both links.

`Watchd` also facilitates restoring the saved values and reexecuting the logged events. It has facilities for rejuvenation, remote execution, error reporting, remote copy, distributed election, and status report production. Several commands are also provided for operating, administering, and maintaining a network using `watchd` daemons.

Libft

`Libft` is a user-level library of C functions used in application programs to specify and checkpoint critical data, recover the checkpointed data, log events, and locate and reconnect the failed client processes to a backup server. It provides a set of functions, such as `critical()`, to specify critical volatile data in an application. These items are allocated in a reserved region of the virtual memory and are periodically checkpointed on primary and backup nodes. The concept of saving only critical data in an application is analogous to the Recovery Box concept in Sprite.[6] This technique allows critical data structures to be saved without traversing them.

The `ftread()` and `ftwrite()` functions in `libft` log messages automatically. In a normal condition, when the `ftread()` function is called by a process, data is read from a channel and automatically logged on a file. The logged data is then duplicated and logged by the `watchd` daemon on a backup machine. The replication of logged data is necessary to enable a process to recover from a primary machine failure. When the `ftread()` function is called by a process that is recovering from a failure in a recovery mode, the input data is read from the logged file before any data is read from a regular input channel. Similarly, the `ftwrite()` function logs output data before it is sent out. The output data is also duplicated and logged by the `watchd` daemon on a backup machine. The log files created by the `ftread()` and `ftwrite()` functions are truncated after a `checkpoint()` function is successfully executed. There is a slight possibility that some messages may get lost during

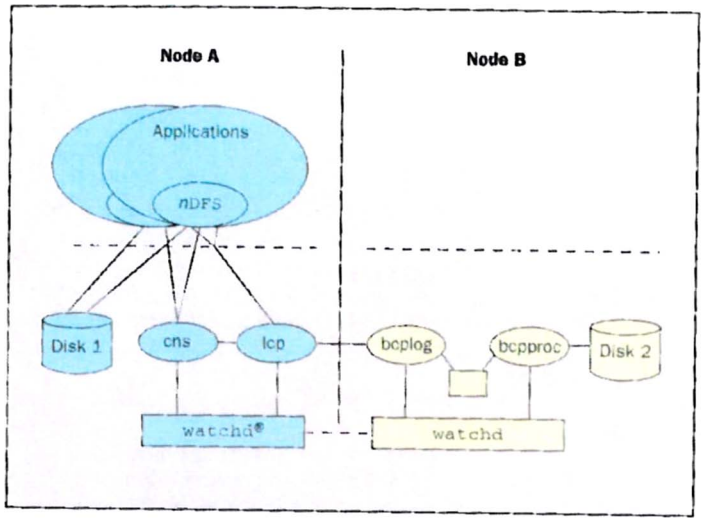

Figure 2. Software architecture of the REPL mechanism.

the automatic restart procedure. If this might affect an application, an additional message synchronization mechanism can be built into the application to check and retransmit lost messages.

The following functions in `libft` perform fault-tolerant versions of the network and file system calls:

- `getsvrloc()`, `getsvrport()`, `ftconnect()`, and `ftbind()` intercept standard socket function calls, help locate servers, and reconnect clients to a backup server when a network failure is detected.
- `ftfopen()`, `ftfclose()`, `ftcommit()`, and `ftabort()` help to commit and abort file updates. Files updated using `ftfopen()` can be committed only by calling `ftfclose()` or `ftcommit()`. In the case of process rollback recovery, file updates can be rolled back to the last commit point.

As it is currently implemented, the `libft` checkpoint mechanism is not fully transparent to programmers. A transparent checkpointing library, `libckp`, is described later in this paper. However, `libft` checkpoints only critical data and thus has less checkpointing overhead. In addition, `libft` does not require a new language, a new preprocessor, or complex declarations and computations to save critical data structures.[2] Sacrificing transparency for speed has proved useful in many projects using `libft`.

REPL

REPL, a file replication mechanism running on a pair of machines, replicates critical files of an application on-line. It often uses the shared library *n*DFS,[7] but it is not always necessary. The mechanism intercepts file system calls by using dynamic shared libraries. When a user program issues a file update, the shared library intercepts the request, performs the update locally, and passes the update message to a remote REPL server. When the message is received, the remote REPL server replays it and performs the file update. Critical files are specified through a UNIX* shell environment variable. Because REPL is built on top of standard file systems, its use requires no change to the underlying operating system. Speed, robustness, and replication transparency are the primary design goals of the REPL replication mechanism.

As shown in Figure 2, REPL consists of four main components:

- Relay server (`lcp`).
- Connection server (`cns`).
- Log server (`bcplog`), and
- Process server (`bcpproc`).

Relay server (`lcp`). When the remote node B is up, the relay server (`lcp`) establishes a connection to `bcplog` (a REPL logging process on the remote backup node described on the next page), reads messages from applications (linked with the *n*DFS library), and passes

```
/* fileapp contains three integers 1, 2 and 3 */
/* a checkpoint is transparently taken here */
fp = fopen("fileapp", "a"); /* for append */
fprintf(fp, "%d", 4);
fclose(fp);
/* failure occurs, roll back */
unlink("fileapp"); /* remove the file */
```

Figure 3. Sample illustrating the need for correct rollback of user files.

those messages to the remote `bcplog`. When the remote is down, the relay server creates a log file and saves data to it.

Connection server (`cns`). The connection server (`cns`) establishes a connection to the `lcp`, maintains a file descriptor for the connection, and sends the file descriptor to applications.

Log server (`bcplog`). The log server (`bcplog`) receives update messages from the `lcp` on primary node A and logs the messages onto a log file.

Process server (`bcpproc`). The process server (`bcpproc`) normally reads the log file generated by `bcplog` and replays the update messages. In recovery, it copies files from the primary node for file resynchronization.

Libckp

`Libckp` is a user-transparent checkpointing library for UNIX applications. It can be linked with a user's program to periodically save the program state on stable storage without modifying the source code. The checkpointed program state includes the program counter, stack pointer, program stack, open file descriptors, global/static variables and dynamically allocated memory of the program, and libraries linked with the program.

Compared to other existing UNIX checkpointing libraries,[8,9] `libckp` has two unique features. First, the library allows a user to include files as part of the process state that is checkpointed and recovered. More specifically, when a process rolls back, all the modifications that it has made to the files since the last checkpoint are undone, to ensure that the states of the files are consistent with the checkpointed volatile state. Other checkpointing libraries either do not support the rollback of user files or only provide this capability to a limited extent.

A straightforward but inconsistent way of extending volatile state checkpointing to user files is to record the file size of each open file when the volatile state is checkpointed. If a rollback is initiated, each file is truncated to the recorded size. Figure 3 shows an example in which this simple approach will result in an inconsistency. Here, the file size of `fileapp` was not recorded at the time of checkpointing (which occurred before the first `fopen()` call was made) because the file was not open at that time. If a rollback is initiated after the `fclose()` call, `fileapp` will not be truncated and the second `fprintf()` call will incorrectly append another `4` to that file.

To checkpoint user files consistently and efficiently, a lazy approach is adopted, which delays the actual checkpointing action until it is necessary. For example, in Figure 3, the size of `fileapp` is recorded (on stable storage) at `fopen()`. Then `fileapp` can be truncated at rollback to the correct size to undo the effect of `fprintf()`. If the failure does not occur, a shadow copy of `fileapp` will be generated at `unlink()`. If a failure occurs later, the shadow copy and the recorded size can be used to restore both the contents and size of `fileapp`.

The second unique feature of `libckp` is its ability to provide a nontransparent mode for flexible execution control. Functions `ckpcheckpoint()` and `ckprollback()` provide application-initiated checkpoint and rollback facilities within a program. The rollback function rolls back the process to a location in the program at which the previous checkpoint was made. These two function calls can be viewed as generalizations of the UNIX system calls `setjmp()` and `longjmp()`, which can restore global/static variables, dynamically allocated memory, and user files.

Addrejuv

`Addrejuv`, an added feature of `watchd`, can rejuvenate software by stopping and restarting a process at a certain interval or when a particular event occurs in the application process. The interval or event for periodic

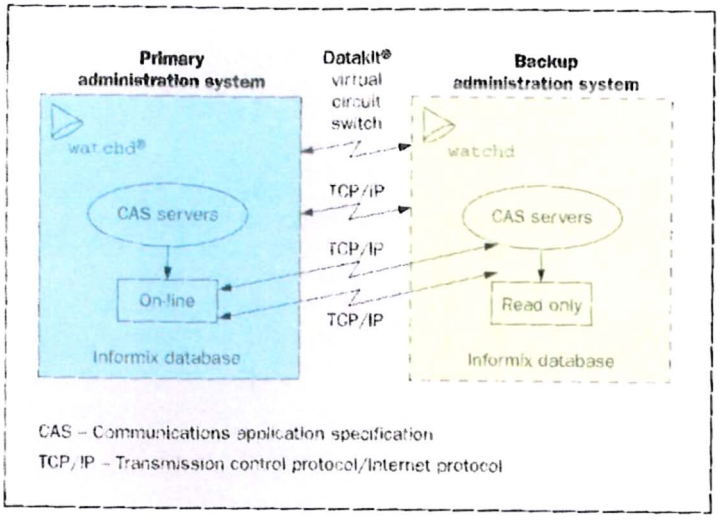

Figure 4. A network administration system using `watchd`.

rejuvenation is determined through analysis and experience with the application.[4,10,11]

When the `addrejuv` feature is used, `watchd` creates a rejuvenation shell script and registers the starting time or the event for execution of that script with the `cron` daemon, which can rejuvenate the process. The shell script stops the process in three steps. First, a signal or a command, specified as the first argument to the `addrejuv` feature, is sent to the process to kill it. Fifteen seconds later, a second signal or command, specified as the second argument to the `addrejuv` feature, is sent to the process. Finally, fifteen seconds later, a SIGKILL signal is sent to the process to make sure that the process is really terminated. The fifteen-second interval between the two signals allows the process to clean up its state before being terminated. The default value of fifteen seconds can be changed by the application. Once the process is terminated, `watchd` takes a recovery action to restore the process exactly as it does when it detects a failure.

Applications

These software fault tolerance components have been used in a wide spectrum of AT&T applications, products, and services to enhance their availability. The applications have been used in Communication Services, for a large-800 customer service system; in Operations Systems, for a network facility administration system; in Transmission Systems, for the DACS VI-2000; and in Switching Systems, for a back-end billing data collector in BNS-2000. Under a license agreement, Tandem Computers Inc. is selling a product named Integrity HATS,* which incorporates `watchd`, `libft`, and REPL. Tandem Computers also has a second set of license agreements to use that software on their Windows NT* platforms.

Administrative Processes. `Watchd` and `libft` increase the availability of administrative processes in network facility administration systems such as those shown in Figure 4.

In these types of systems, a script registers application processes with `watchd`. As soon as `watchd` detects a process failure (crash or hang), it executes a script to recover the processes. The recovery script usually:

- Shuts down the failed machine (to make sure that it is really down);
- Notifies system administrators;
- Restarts the failed services on the backup machine;
- Performs an Internet protocol (IP) address failover, to rebind and reconnect sockets, if necessary;
- Notifies the necessary party to signal the completion of the failover; and
- Resumes services.

`Libft` checkpoints the process states and sends

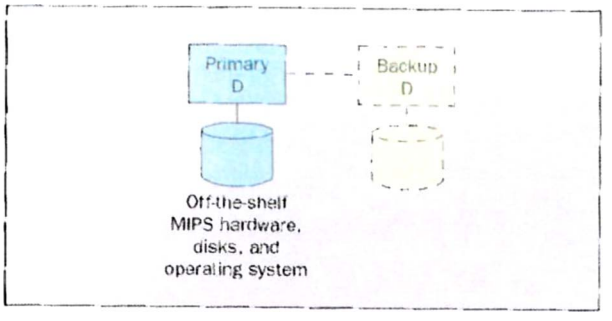

Figure 5. A cross-connection switch using REPL.

heartbeats to `watchd`. To save the critical state of a process, a user specifies the data critical to recovery, and `libft` checkpoints only that critical data. At the process recovery time, it can then restore its critical data to make the recovery transparent to service requesters or to bypass the long initialization procedure that the process may otherwise have to go through.

A Cross-Connection System. Along with `watchd` and `libft`, `REPL` is also used in systems such as DACS-VI 2000, a cross-connection switch for international transmission networks (see Figure 5).

These systems use duplex components for CPUs and disks. To prevent any single point of failure, all the modifications to one disk are duplicated to another disk. `REPL` provides a solution that is simpler and less expensive than using mirrored disks or disk arrays.

Long-Running Programs. `Libckp` is used in several long-running computer-aided design (CAD) programs, simulation programs, and signal processing applications. Their execution times range from 2 to 17 hours; the checkpoint sizes range from 0.3 to 40.0 MB. With a 30-minute default checkpoint interval, the checkpoint overhead typically ranges from 0 to 7 percent, which is acceptable in most applications. `Libckp` has also been integrated into a load-sharing migration server called CosMiC. When a user submits a job, CosMiC finds a remote idle workstation to run it; when the owner of the workstation starts using his or her machine, CosMiC kills the job and automatically migrates it to another idle workstation. If the job is linked with `libckp`, the migration will restart the job from the previous checkpoint to avoid a total loss of useful work.

A Billing Data Collection System. Software rejuvenation is implemented in the BILLDATS II® Collector, a billing data collection system deployed throughout the AT&T long distance network, and in several of the Regional Bell Operating Companies (RBOCs) and Independent Telephone Companies (ITCOs). Based on predeployment laboratory testing with longevity runs approximating two weeks, the rejuvenation interval in that system is conservatively set to one week for field installations. After several years of field operation of the BILLDATS II Collector with rejuvenation, not a single incident of the type of failures that affect longevity has been encountered to date. Memory and process rejuvenation is also implemented in some CAD and speech recognition applications.

About three years ago, a telecommunications traffic network monitoring system periodically fell into a nonresponsive state. Whenever this happened, the field administrator used a secondary channel to connect to the system and kill the user processes. This freed buffers and allowed them to be used for remote login connections on the primary channel. After about 6 months of that mode of operation and intermittent investigation to trace the bug, it was found that a certain module in the system was ignoring the return value from the `canput()` routine, causing the STREAMS I/O buffer pool to overflow. (STREAMS is a feature of UNIX System V that provides a standard way of dynamically building and passing messages up and down a protocol stack.) The bug was fixed and the system is now operating without that problem. If the software rejuvenation module had been in place then, it probably would have periodically rejuvenated the system by sweeping out the offending user processes and releasing the buffers automatically, without requiring manual intervention.

Conclusion

Most of the theory for today's software technology focuses on the static behavior of the software, such as analysis of source code. There is little theory about the dynamic behavior of software executing under varying load conditions. To avoid serious network problems, for example, software systems are often overengineered, with enough bandwidth for two or three times the expected load. Without analysis of their dynamic behavior, there is no way to determine the resources an

application will need once it is working. As an analogy, until feedback control theory was developed, electronic systems had to be hand-crafted and tuned to prevent them from failing intermittently. A similar theory is needed for software.

To relate this to chaos theory, one cannot be certain that a small change in software will result in a small change in system performance. The April 25, 1994, issue of *Forbes Magazine* points out that a three-line change to a 2-million line program caused multiple failures that were traced back to a single fault. The concepts and technologies described in this paper point to a new direction for software research, one called software dynamics.

An important part of software dynamics is the software stability theory. Even after a bug is found and fixed, it is difficult to design a system that can restore software to a known stable state, in which its operation can be tested. One exception to this may be custom-designed systems, but those are known to be error prone. When a system is restored to a stable state, ensuring that the system behaves predictably under varying load conditions is a major aspect of this stability theory. Software behaves like a nonlinear, nonstationary process. Rejuvenation forces the software states to be periodic. This method does not allow problems such as roundoff errors, buffer and file overflows, and protocol retries to build up and corrupt the execution of the software systems in untested domains of operation.

The concepts and technologies described in this paper point to software dynamics as a new direction for software research. Topics in software dynamics include software fault tolerance (the subject of this paper), safety-critical software, and software testing and performance analysis. The codification of these technologies will enable software developers to understand how software performs under load. With that information at hand, they can reliably predict system performance for a range of offered loads and define the point at which the software will fail. This body of knowledge will not only move the software industry toward quantitative analysis, component modeling, and specifications, but will also mark its maturation.

Acknowledgments

The authors would like to thank Dave Korn, Glenn Flower, and Herman Rao for their help in the design and development of `REPL`, and Sharmila Deshpande and John Eldridge for describing the system shown in Figure 4.

***Trademarks**

Integrity HATS is a registered trademark of Tandem Computers, Inc.

UNIX is a registered trademark of Novell in the United States and other countries, licensed exclusively through X/Open Company Limited.

Windows NT is a trademark of Microsoft Corporation.

(Manuscript approved March 1996)

References

1. L. Bernstein, "Innovative Technologies for Preventing Network Outages," *AT&T Technical Journal*, Vol. 72, No. 4, July/August 1993, pp. 4-10.
2. J. Gray and D. P. Siewiorek, "High Availability Computer Systems," *IEEE Computer*, Vol. 24, No. 9, September 1991, pp. 39-48.
3. Y. Huang and C. M. R. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," *Proceedings of 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*. Toulouse, France, June 1993, pp. 2-9. Also appeared as a chapter in the book *Software Fault Tolerance*, M. Lyu (Ed.), John Wiley & Sons, New York, March 1995.
4. Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proceedings of 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, Pasadena, California, June 1995, pp. 381-390.
5. R. Bianchini, Jr., and R. Buskens, "An Adaptive Distributed System-Level Diagnosis Algorithm and Its Implementation," *Proceedings of 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, July 1991, pp. 222-229.
6. M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," *Proceedings of Summer USENIX*, June 1992, pp. 31-43.
7. G. S. Fowler, Y. Huang, D. G. Korn, and H. Rao, "A User-Level Replicated File System," *Proceedings of Summer USENIX*, June 1993, pp. 279-290.
8. M. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel," *Proceedings of Winter USENIX Conference*, San Francisco, California, January 1992, pp. 283-290.
9. J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under UNIX," *Proceedings of Winter USENIX*

Technical Conference, New Orleans, Louisiana, January 1995, pp. 213-224.
10. Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. M. R. Kintala, "Checkpointing and Its Applications," *Proceedings of 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, Pasadena, California, June 1995, pp. 22-31.
11. E. J. Weyuker and A. Avritzer, "Estimating Software Reliability of Smoothly Degrading Systems," *Proceedings of the 5th International Symposium on Software Reliability Engineering*, November 1994, pp. 168-174.

Yennun Huang *is a member of technical staff in the Distributed Software Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He develops software tools and techniques that improve system reliability and availability. Mr. Huang received a B.S. in electrical engineering from National Taiwan University, Taipei, and M.S. and Ph.D. degrees in computer science from the University of Maryland at College Park. He joined AT&T in 1989.*

Chandra M. R. Kintala *is head of the Distributed Software Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is responsible for software technology in distributed applications, with emphasis on software fault tolerance. Mr. Kintala received an M.Tech in electrical engineering from the Indian Institute of Technologies, Kanpur, India, and a Ph.D. in computer science from Pennsylvania State University, University Park. He joined AT&T in 1980.*

Lawrence Bernstein *is Network Operations Platform Vice President of AT&T Network Systems and Executive Director at AT&T Bell Laboratories. He directs the automation of telephone company business operations and database software for network control. Mr. Bernstein is also responsible for development, software, engineering project management, and marketing. He received a B.S. from Rensselaer Polytechnic Institute, Troy, New York, and an M.S. from New York University, New York City, both in electrical engineering. He is a fellow of the IEEE and an Industrial Fellow of Ball State Center for Information and Communication Sciences. Mr. Bernstein holds one patent for logic design and four for software. He joined AT&T in 1961.*

Yi-Min Wang *is a member of technical staff in the Distributed Software Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He designs and develops reusable software components for enhancing the availability of software systems. Mr. Wang received a B.S.E.E. from National Taiwan University, Taipei, and M.S. and Ph.D. degrees in computer engineering from the University of Illinois at Urbana-Champaign. He joined AT&T in 1993.*