10

Software Fault Tolerance in the Application Layer

YENNUN HUANG and CHANDRA KINTALA
AT&T Bell Laboratories

ABSTRACT

By software fault tolerance in the application layer, we mean a set of application level software components to detect and recover from faults that are not handled in the hardware or operating system layers of a computer system. We consider those faults that cause an application process to crash or hang; they include application software faults as well as faults in the underlying hardware and operating system layers if they are undetected in those layers. We define four levels of software fault tolerance based on availability and data consistency of an application in the presence of such faults. We describe three reusable software components that provide up to the third level of software fault tolerance. Those components perform automatic detection and restart of failed processes, periodic checkpointing and recovery of critical volatile data, and replication and synchronization of persistent data in an application software system. These components have been ported to a number of UNIX[2] platforms and can be used in any application with minimal programming effort.

Some telecommunications products in AT&T have already been enhanced for fault-tolerance capability using these three components. Experience with those products to date indicates that these modules provide efficient and economical means to increase the level of fault tolerance in a software system. The performance overhead due to these components depends on the level and varies from 0.1% to 14% based on the amount of critical data being checkpointed and replicated.

[1] This is an expanded version of the paper "Software Implemented Fault Tolerance: Technologies and Experience" in *Proceedings of 23rd Intl. Symposium on Fault Tolerant Computing (FTCS-23)*, Toulouse, France, pages 2–9, June 1993.
[2] UNIX is now a registered trademark of X/Open Co.

10.1 INTRODUCTION

There are increasing demands to make the application software systems we build today more tolerant to faults. From a user's point of view, fault tolerance has two dimensions: *availability* and *data consistency* of the application. For example, users of telephone switching systems demand continuous availability whereas bank teller machine customers demand the highest degree of data consistency. Safety critical real-time systems such as nuclear power reactors and flight control systems need highest levels in both availability and data consistency. Most other applications have lower degrees of requirements for fault-tolerance in both dimensions; see Figure 10.1. But, the trend is to increase those degrees as the costs, performance, technologies and other engineering considerations permit.

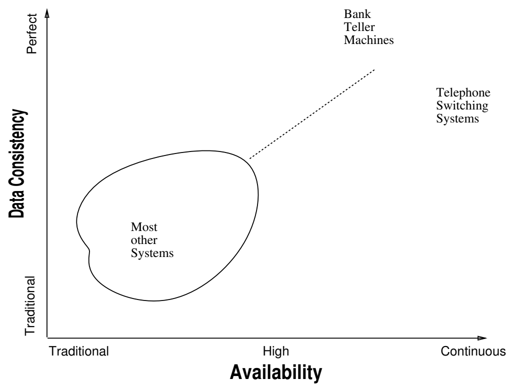

Figure 10.1 Dimensions of fault tolerance

Availability and data consistency in an application is traditionally provided through fault-tolerant hardware and operating system used by the application for its execution. New trends are emerging in the marketplace that are changing this tradition. Standard commercial hardware and operating systems are becoming highly reliable, distributed and inexpensive to the extent that they are now off-the-shelf commodity items. New application software systems are increasingly networked and distributed, i.e. mostly client-server systems. Many of those applications are also built from reusable components whose sources are unknown to the application developers. Due to this complexity in application software, the proportion of failures due to faults in the application software is increasing. The *End-to-End* type of arguments imply that one needs fault tolerance in the application software itself to handle such failures. Also, as the society's dependence on such diverse and distributed applications grows, demands for more reliable and yet economical fault-tolerant software will grow.

In this paper, we discuss three cost-effective reusable software components, `watchd`, `libft`, and `REPL`, to raise the degree of fault tolerance in an application's availability and data consistency dimensions. We discuss the background concepts of software faults, failures and fault tolerance in Section 10.2 and then present a model for providing fault tolerance through software in Section 10.3. The three components are described in Section 10.4 and the experience with applications in using those technologies are discussed in Section 10.5, followed by some concluding remarks in Section 10.6.

10.2 BACKGROUND

10.2.1 Software Faults and Failures

Following Cristian [Cri91], we consider distributed software applications that provide a "service" to clients. The applications in turn use the services provided by the underlying operating or database systems which in turn use the computing and network communication services provided by the underlying hardware; see Figure 10.2.

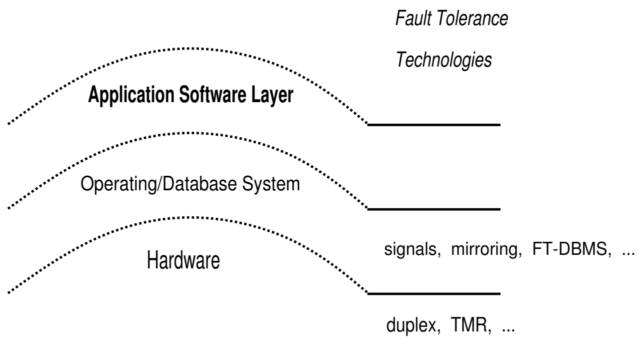
Fault Tolerance
Technologies
Application Software Layer
Operating/Database System
Hardware
signals, mirroring, FT-DBMS, ...
duplex, TMR, ...

Figure 10.2 Layers of fault tolerance

Due to the complex and temporal nature of interleaving of messages and computations in distributed software systems, no amount of verification, validation and testing can eliminate all faults in an application and give complete confidence in the availability and data consistency of that application. So, those faults occasionally manifest themselves as failures, causing application processes to crash or hang. A process is said to be *crashed* if the working process image is no longer present in the system. A process is said to be *hung* if the process image is alive, its entry is still present in the process table but the process is not making any progress from a user's point of view.

Tolerating faults in such applications involves detecting a failure, gathering knowledge about the failure and recovering from that failure. Traditionally, these fault tolerance actions are performed in the hardware, operating or database systems used in the underlying layers of the application software. Hardware fault tolerance is provided using Duplex, Triple-Module-Redundancy or other techniques [Pra86]. Fault tolerance in the operating and database layers is often provided using replicated file systems [Sat90], exception handling [Shr85], disk shadowing [Bit88], transaction-based checkpointing and recovery [Nan92], and other system routines. These methods and technologies handle faults occurring in the underlying hardware, operating and database system layers only.

Increasing number of faults are however occurring in the application software layer causing application processes to crash or hang [Sie92]. Software, unlike hardware, has no physical properties. So, the only kind of faults it has are design and coding faults. Due to the permanent nature of such design faults, it has been generally assumed that the failures caused by software faults are also permanent. This belief led to the use of design diversity for supporting fault tolerance. With design diversity, if a module cannot provide its service, then another module which has a different design is used to provide the required service. The two well known methods for design diversity are the recovery block approach (see Chapter 1), and the N-version programming approach (see Chapter 2).

However, the failures exhibited by those software faults can be *transient*, i.e. the failure may not recur if the software is reexecuted on the same input [Gra91, Wan93]; this is a frequently used technique in hardware to mask transient hardware failures. Sullivan and Chillarege [Sul92] also showed that a large percentage of software errors are triggered by peak conditions in workload, exception handling and timing. Such errors are likely to disappear when the software is reexecuted after a certain amount of clean-up and reinitialization [Ber92]. This is because the behavior of a program, especially a client server application running on a distributed system, depends not only the input data and message contents but also on the timing and interleaving of messages, shared variables and other "state" values in the operating environment of the application [Cri91, Hua94].

10.2.2 Software Fault Tolerance

It is possible to detect a software failure and restart the application at a checkpointed state through operating system facilities, as in IBM's MVS [Sie92]. In their chapter on *End-to-End Arguments* [Sal84], Saltzer et. al. claim that such hardware and operating system based methods to detect and recover from software failures are necessarily incomplete. They show that fault tolerance cannot be complete without the knowledge and help from the endpoints of an application, i.e., the application software itself has to be engaged to provide complete end-to-end fault tolerance. We claim that such hardware and operating system based methods, i.e. services at a lower layer detecting and recovering from failures at a higher layer, may also be inefficient. For example, file replication on a mirrored disk through a facility in the operating system is more inefficient in execution time and space usage than replicating only the "critical" files of the application in the application layer since the operating system has no internal knowledge of that application. Similarly, generalized checkpointing schemes in an operating system checkpoint entire in-memory data of an application whereas application-assisted methods checkpoint only the critical data [Lon92, Bak92].

A common but misleading argument against embedding checkpointing, recovery and other fault tolerance schemes inside an application is that such schemes are not efficient or reliable because they are coded by application programmers. We claim that well-tested and efficient fault tolerance methods can be built as libraries or reusable software components that can be linked into an application and that they are as efficient as some of the operating system based methods. They may not be as transparent to the application as the other methods but they are much more portable across many hardware and operating system platforms since they are in the application layer. All the three components discussed in this chapter are efficient, reliable and portable across many platforms.

The above observations lead to our notion of software fault tolerance as:

> a set of software components executing in the application layer of a computer system to detect and recover from faults that are not handled in the underlying hardware or operating system layers.

We consider all faults that cause an application process to crash or hang which include software faults as described earlier as well as faults in the underlying hardware and operating system layers if they are undetected in those layers. Thus, if the underlying hardware and operating system are not fault-tolerant due to performance/cost trade-offs or other engineering considerations in an application system, then that system's availability can be increased cost effectively through software fault tolerance components described in this chapter.

10.3 MODEL

For simplicity in the following discussions, we consider only client-server based applications running in a network of computers (nodes) in a distributed system[3]. Such an application has a server process and several client processes executing in the user level (application layer) on top of vendor supplied hardware and operating systems. To get services, client processes send messages to the server process. In each of those message processing steps, the server process performs the required computation and data processing and sends back a response if necessary. We sometimes call the server process *the application*. For fault tolerance purposes, the nodes in the distributed system are viewed as being in a circular configuration so that each node is a backup node for its left neighbor in that circular list. As shown in Figure 10.3, each application is executing primarily on one of the nodes in the network, called the primary

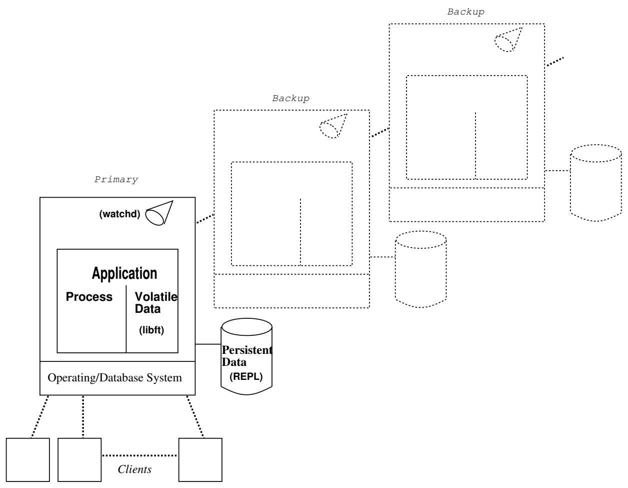

Figure 10.3 Model for software fault tolerance in the application layer

node for that application. Each executing application has *process text* (the compiled code), *volatile data* (variables, structures, pointers and all the bytes in the static and dynamic memory segments of the process image) and *persistent data* (the application files being referred to and updated by the executing process).

10.3.1 Modified Primary-Site Approach

We modify the primary-site approach to software fault tolerance [Als76] slightly in our model. In the primary site approach, the service to be made fault tolerant is replicated at many nodes,

[3] These discussions also apply to other kinds of applications. Indeed, the components described in the next section have been used in compute-intensive non-distributed applications.

one of which is designated as primary and the others as backups. All the requests for the service are sent to the primary site. The primary site periodically checkpoints its state on the backups. If the primary fails, one of the backups takes over as primary. This model for fault tolerance has been analyzed for frequency of checkpointing, degree of service replication and the effect on response time by Huang and Jalote [Hua89, Hua92]. This model is slightly modified, as described below, to build the three technologies described in this chapter.

- Each node has its left (or right) neighbor in the circular configuration designated as a backup node.
- The application is active only on its primary node; the application is inactive (i.e. process image is available but not executing) on its backup node.
- A watchdog process (called `watchd` in our technologies), is running on the primary node watching for application crashes or hangs.
- Another watchdog process is running on the backup node watching for primary node crashes.
- A routine (supplied by `libft` library in our technologies) is periodically checkpointing the critical volatile data in the application and logging the client messages to the application between checkpoints.
- A replication mechanism (called `REPL` in our technologies) is running on the primary and the backup nodes to duplicate application's persistent data on the backup node.
- When the application on the primary node crashes or hangs, it is restarted on the primary node, if possible, or on the backup node, otherwise.
- The application is restored to its latest possible internal state before the failure using the checkpointed data structures and message log.
- The application is connected to the replicated files on the backup node if the application restarts on the backup.

Observe that these software fault tolerance tasks can be used in addition to other methods such as N-version programming [Avi85] or recovery blocks [Ran75] inside an application program. Observe also that the application process on the backup node need not be running until it is started by the watchdog process and hence there are no consistency and concurrency concerns; this is unlike in the process-pair model [Gra91] where the backup process is actively running even during normal operations.

10.3.2 Levels of Software Fault Tolerance

The degree to which the above software fault tolerance tasks are used in an application determines the availability and data consistency of that application. It is, therefore, useful to establish a classification of the different levels of software fault tolerance. We define the following 4 levels based on our experience in AT&T. Applications illustrating these levels are described in Section 10.5.

Level 0: *No tolerance to faults in the application software:*
In this level, when the executing application process dies or hangs, it has to be manually restarted from an initial internal state. The application may leave its data in an incorrect or inconsistent state due to the timing of the crash and may take a long time to restart due to elaborate initialization procedures.

Level 1: *Automatic detection and restart:*
When the application dies or hangs, the error is detected and the application is restarted

from an initial internal state on the same processor, if possible, or on a backup processor if available. In this level, the internal state of the application is not saved and, hence, the process restarts at the initial internal state. As stated above, restart along with reinitialization is slow. The restarted internal state may not reflect all the messages that have been processed in the previous execution, and therefore, may not be consistent with the persistent data. The difference between Levels 0 and 1 is that the detection and restart are automatic in Level 1, and therefore, the application availability is higher in Level 1 than in Level 0.

Level 2: *Level 1 plus periodic checkpointing, logging and recovery of internal state:*
In addition to what is available in Level 1, the internal state of the application process is periodically checkpointed, i.e. the critical volatile data is saved, and the messages to the application are logged. After a failure is detected, the application is restarted at the most recent checkpointed internal state and the logged messages are reprocessed to bring the application close to the state at which it crashed. The application availability and volatile data consistency are higher in Level 2 than those in Level 1.

Level 3: *Level 2 plus persistent data recovery:*
In addition to what is available in Level 2, the persistent data of the application is replicated on a backup disk connected to a backup node, and is kept consistent with the data on the primary node throughout the normal operation of the application. In case of a fault and resulting recovery of the application on the backup node, the backup disk brings the application's persistent data as close to the state at which the application crashed as possible. The data consistency of the application in Level 3 is higher than that in Level 2.

Level 4: *Continuous operation without any interruption:*
This level of fault tolerance in software guarantees the highest degree of availability and data consistency as required, for example, in safety critical real-time systems. Often, this is provided by replicated processing of the application on "hot" spares, such as the recovery block in Chapter 1 or the N-version software in Chapter 2. The technologies we describe in this chapter do not provide this level of fault tolerance and hence we do not recommend them to be exclusively used in such applications.

10.4 TECHNOLOGIES

Sometimes, the fault tolerance tasks described in the previous section are individually implemented in an application in an *ad hoc* manner. We developed three generic and reusable components (`watchd`, `libft` and `REPL`)[4] to embed those tasks in any application with minimal programming effort.

10.4.1 Watchd

10.4.1.1 PROCESS RECOVERY

`Watchd` is a watchdog daemon process that runs on a single machine or on a network of machines. It continually watches the life of a local application process by periodically sending a null signal to the process and checking the return value to detect whether that process is alive or dead. It detects whether that process is hung or not by using one of the following two methods specified by the application. In the first method, `watchd` sends a null message to

[4] `watchd`, `libft` and `REPL` are registered trademarks of AT&T Bell Laboratories.

the local application process using IPC (Inter Process Communication) facilities on the local node and checks for a response. If `watchd` cannot make the connection, it waits for some 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 sends a heartbeat message to `watchd` periodically and `watchd` periodically 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` provides the function `hbeat()` for applications to send heartbeats to `watchd`. The `hbeat()` function has an argument whose value specifies the duration for the next heartbeat to arrive.

When it detects that the application process crashed or hung, `watchd` recovers that application at an initial internal state or at the last checkpointed state. The application is recovered on the primary node if that node has not crashed, otherwise on the backup node for the primary 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`.

10.4.1.2 PROCESSOR RECOVERY

`Watchd` also watches one neighboring `watchd` (left or right) in a circular fashion to detect node failures; this circular arrangement is similar to the adaptive distributed diagnosis algorithm [Bia91]. When a node failure is detected, `watchd` can execute user-defined recovery commands and reconfigure the network. Observe that neighboring `watchd`s cannot fully differentiate between node failures and link failures. In general, this is the problem of attaining common knowledge in the presence of communication failures which is provably unsolvable [Hal90]. However, to minimize the problem, `watchd` can use two communication links for polling a neighboring node. Only when it can not reach the neighboring node by both links, `watchd` reports a node failure; an example is given in Section 10.5, Level 3.

10.4.1.3 SELF RECOVERY

`Watchd` also watches itself. A self-recovery mechanism is built into `watchd` in such a way that it can recover itself from an unexpected software failure. When `watchd` finishes initialization, it forks a backup `watchd`. The backup `watchd` executes a loop and keeps polling the primary `watchd`. If the primary `watchd` fails, the backup `watchd` breaks the polling loop and resumes the primary `watchd`'s task by itself becoming the primary. It also spawns a new backup `watchd` for watching itself, the new primary `watchd`. If the backup `watchd` fails, the primary `watchd` gets a signal from operating system since the backup `watchd` is always a child process of the primary `watchd`.

`Watchd` also facilitates restarting a failed process, restoring the saved values and reexecuting the logged events and provides facilities for remote execution, remote copy, distributed election, and status report production.

10.4.2 Libft

`Libft` is a user-level library of C functions that can be used in application programs to specify and checkpoint critical data, recover the checkpointed data, log events, locate and reconnect to a server, do exception handling, do N-version programming (NVP), and use recovery block techniques.

10.4.2.1 FUNCTIONS

`Libft` provides a set of functions, described below, to specify critical volatile data in an application. Those critical data items are allocated in a reserved region of the virtual memory and are periodically checkpointed. The reserved region is saved using a single system call to the memory copy function (`memcpy()`); we thus avoid traversing complex, application-dependent data structures. When an application does a checkpoint, its critical data is saved on the primary and backup nodes. Unlike other checkpointing methods [Lon92], the overhead in our checkpointing mechanism is minimized by saving only critical data and avoiding data-structure traversals. This idea of saving only critical data in an application is analogous to the Recovery Box concept in Sprite [Bak92].

Data structure checkpointing, recovery, fault-tolerant network communication and file operations are done using the following functions in `libft`.

- `ft_start()` reserves a block of critical memory. The function takes two arguments — the size of the critical memory and the file name for checkpoint data. When in recovery, `ft_start()` restores the data structures from the critical memory in reserved address space.
- `t_critical()` declares critical global variables along with an `id` to identify the thread that made the call; function `critical()` is similar to `t_critical()` without the identifier. Both functions take a list of variables and their sizes as input arguments.
- `t_checkpoint()` and `checkpoint()` save the values of critical variables and the critical memory onto a file.
- `t_recover()` and `recover()` restore the values of critical variables and critical memory.
- `ftmalloc()`, `ftcalloc()` and `ftrealloc()` are used to allocate space from the critical memory and function `ftfree()` is used to free space to critical memory.
- `getsvrloc()`, `getsvrport()`, `ftconnect()` and `ftbind()` are used by clients to locate server processes and reconnect to servers in a network environment.
- `ftfopen()`, `ftfclose()`, `ftcommit()` and `ftabort()` help in committing and aborting file updates. Files updated using `ftfopen()` can be committed only by calling `ftfclose()` or `ftcommit()`. Therefore, in the case of process rollback recovery, file updates can be rolled back to the last commit point.

`Libft` also provides `ftread()` and `ftwrite()` functions to automatically log messages. When the `ftread()` function is called by a process in a normal condition, the data are read from a channel and automatically logged on a file. The logged data then are duplicated and logged by the `watchd` daemon on a backup machine. The replication of logged data is necessary for a process to recover from a primary machine failure. When the `ftread()` function is called by a process which is recovering from a failure in a recovery situation, the input data are read from the logged file before any data can be read from a regular input channel. Similarly, the `ftwrite()` function logs output data before they are 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. Using functions `checkpoint()`, `ftread()` and `ftwrite()`, one can implement either a sender-based or a receiver-based logging and recovery scheme [Jal89]. There is a slight possibility that some messages during the automatic restart procedure may get lost. If this is a concern to an application, an additional message synchronization mechanism can be built into the application to check and retransmit lost messages.

The exception handling, NVP and recovery block facilities are implemented using C macros and standard C library functions. These facilities can be used by any application without changing the underlying operating system or adding new C preprocessors.

Speed and portability are primary concerns in implementing `libft`. The `libft` checkpoint mechanism is not fully transparent to programmers as in the Condor system [Lit88]. However, `libft` does not require a new language, a new preprocessor or complex declarations and computations to save data structures [Gra91]. The sacrifice of transparency for speed has been proven to be useful in some projects to adopt `libft`. The installation of `libft` doesn't require any change to a UNIX-based operating system; it has been ported to several platforms.

`Watchd` and `libft` separate fault detection and volatile data recovery facilities from the application functions. They provide those facilities as reusable components which can be combined with any application to make it fault tolerant. Since the messages received at the server site (active node) are logged and only the server process is recovered in this scheme, the consistency problems that occur in recovering multiple processes [Jal89] are not issues in this implementation.

10.4.2.2 EXAMPLE

The following program is an example of a server program using `libft` library for checkpointing. The server program reads a number from a client and pushes the number onto the top of a stack. The stack is implemented using a linked list.

```
#include <ft.h>
...
struct llist {
    int data;
    struct llist *link;
    ...
}
...
main(){
    struct llist *pHead=NULL, *ptmp;
    int s, indata;
    ...
    ft_start("/tmp/examp1",16384);
    critical(&pHead, sizeof(pHead),0);
    ...
    for (;;) {
        ...
        if (in_recovery()) recover(INFILE);
        if (application decides to checkpoint due to
            a change in its state) checkpoint(INFILE);
        ...
        s=accept(..);
        read(s,indata,MaxLen);
        ptmp=(struct llist *) ftmalloc(sizeof(struct llist));
        ptmp->link=pHead;
        ptmp->data=indata;
        pHead=ptmp;
        ...
    }
}
```

The critical data in the above program is the stack itself; to save it, the pointer to the top of the stack, pHead, and the stack size are declared to be critical. To save the contents of the stack, the stack elements are assigned from the critical memory. A critical memory of size 16K bytes is created by the `ft_start()` function. The size of the critical memory can be dynamically increased as needed. `in_recovery()` function returns 1 or 0 indicating whether the program is in recovery state.

10.4.2.3 OTHER CONSTRUCTS

`Libft` provides `C`-style constructs to do N-version programming, recovery block, exception handling and program retry block. All the constructs are implemented using macros. Therefore, no new `C` preprocessor or compiler is needed. The syntax of each construct is listed below.

Recovery block:

```
#include <ftmacros.h>
...
ENSURE(accept-test) {
       primary block;
} ELSEBY {
       secondary block 1;
} ELSEBY {
       secondary block 2:
}
...
ENSURE;
```

In the above program, `accept-test` is a condition statement which should return 0 if the condition fails.

N-version programming:

```
#include <ftmacros.h>
...
NVP
VERSION{
      block 1;
      SENDVOTE(v-pointer, v-size);
}
VERSION{
        block 2;
        SENDVOTE(v-pointer, v-size);
}
...
ENDVERSION(timeout,v-size);
if (!agreeon(v-pointer)) error_handler();
ENDNVP;
```

The `v-pointer` is a pointer to a critical variable containing recovery block's output data to be voted upon. Function `SENDVOTE()` sends that data to a voting registrar. The function `agreeon()`, the default registrar, returns 1 if a majority of the returned data agree. In this case, the result of the voting is stored in the function argument. Otherwise, function `agreeon()` returns 0.

Exception handling:

```
#include <ftmacros.h>
...
exception name1, ...;
TRY
    statements;
EXCEPT(name1) {
    handler routine 1;
}
EXCEPT(name2) {
    handler routine 2;
}
...
ENDTRY;
```

To raise an exception, `THROW(name)` construct is used.

Retry block:

```
#include <ftmacros.h>
...
START(max_no) {
     statements;
}
FINISHBY(post-condition);
```

The program stops if the retry block can not satisfy the `post-condition` after `max_no` of retries.

10.4.3 REPL

REPL file replication technology provides facilities for on-line replication of user specified files on a backup node. The implementation of REPL uses a shared library to intercept file system calls, as in *n*`DFS` [Fow93], and is built on top of UNIX file systems. So, it runs entirely in the application layer and requires no change to the underlying file system or operating system. Speed, robustness and replication transparency are the overriding goals in the design and implementation of REPL.

10.4.3.1 COMPONENTS

REPL technology contains two parts — a shared library `librepl.a` and a `REPL` system of processes to run on the primary and backup nodes. Applications are linked with `librepl.a` library. The library intercepts all the system calls that operate on the specified critical files of that application, generates file update messages and routes those systems calls to the underlying file system on the primary as in a normal system call. The `REPL` system on the primary transports the generated file update messages to the `REPL` system on the backup host using the available transport mechanisms such as sockets. The `REPL` system on the backup host receives and logs the update messages from the primary and performs the corresponding updates on the backup files asynchronously. The shared library can be linked with the application either dynamically if the underlying UNIX supports dynamic shared libraries (e.g., Sun's OS 4.3 and higher, Solaris IRIX 5.1 and higher) or can be linked with the application during compilation. Thus, REPL has five major components; see Figure 10.4. They are:

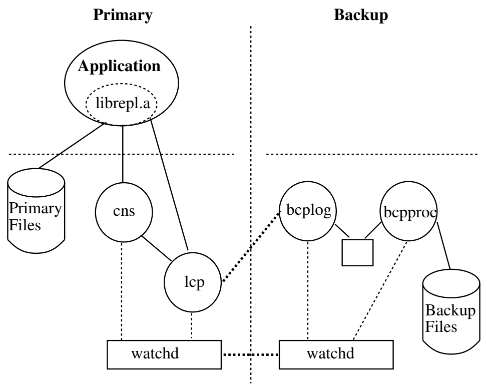

Figure 10.4 Software architecture of REPL

- `librepl.a`, the shared library that intercepts file system calls from the application,
- `cns`, the connection server on the primary; it creates a child process, `lcp`, maintains a file descriptor for that connection to `lcp` and sends that file descriptor to the application,
- `lcp`, the child process of `cns`; it opens a connection to `bcplog` on the backup host, receives data corresponding to the intercepted file system operations in the application from `librepl.a`, sends messages about those update operations to `bcplog`,
- `bcplog`, the backup log server that receives update messages from `lcp` on primary and logs them onto a log file and
- `bcpproc`, the backup process server that reads the log file, processes the update messages and performs those operations on the backup files.

10.4.3.2 PRIMARY FAILURE RECOVERY

`watchd` and `libft` together detect and recover failures of any of those components in REPL. If the primary node fails or the application running on the primary fails, then `watchd` and `libft` recover the application on the backup node as explained in the previous section.

The recovered application on the backup node gets access to the replicated files. REPL itself uses `watchd` and `libft` for fault detection and recovery from failures in its mechanism. If one of the components of REPL fails (i.e. a software failure in REPL) or if the backup file system fails, then REPL recovers itself from such failures as explained below. A crashed backup file system, after it is repaired, can catch up with the primary file system without appreciably slowing down the applications running on the primary. The failure and the recovery are transparent to applications and users.

10.4.3.3 BACKUP FAILURE RECOVERY

If the backup node fails, `watchd` running on the primary detects the failure in about 20 seconds and sends a signal (`SIGPIPE`) to `lcp`. Then, `lcp` creates a local log file, `ftopenlog`, and writes all the incoming data to the log file while the backup node is down. After the backup node is repaired and rebooted, `watchd`, `bcplog` and `bcpproc` are restarted on the backup. The new `bcplog` sends a signal to the `lcp` on the primary node, and forces the `lcp` to connect to the new `bcplog`. Once the connection is established, `lcp` sends new file update messages to that `bcplog`. In addition, the backup `bcpproc` gets the logged file, `ftopenlog`, from the primary node and then processes the update messages on the backup file system in order to catch up with state of the primary file system. While the backup node is down, if the size of the `ftopenlog` file becomes too large, `lcp` stops logging the operations and puts a flag at the beginning of the log file. When the backup node is rebooted and `bcpproc` is restarted, `bcpproc` copies all the critical files from the primary node. At the same time, `bcplog` logs the updates coming from the primary node. When the file copy is complete, `bcpproc` processes the log files created by `bcplog` to catch up with the primary file system. Eventually, all the log files are processed and the recovery is complete. We assume that the relative loads on the primary and backup nodes are such that they can process these file recovery operations without appreciable degradation of the normal application processing.

10.5 EXPERIENCE

Fault tolerance in some of the telecommunications network management products in AT&T has been enhanced using `watchd`, `libft` and `REPL`. Experience with those products to date indicates that these technologies are indeed economical and effective means to increase the level of fault tolerance in application software. The performance overhead due to these components depends on the level of fault tolerance, the amount of critical volatile data being checkpointed, frequency of checkpointing, and the amount of persistent data being replicated. The overhead varies from 0.1% to 14%. We describe some of those products to illustrate the availability, flexibility and efficiency in providing software fault tolerance through these 3 components. To protect the proprietary information of those products, we use generic terms and titles in the descriptions.

10.5.1 Example 1

Level 1 - Failure detection and restart using `watchd`:

Application C monitors and analyzes data in a special purpose on-line billing system on AT&T's network. Application C uses `watchd` to check the "liveness" of some service daemon processes in C at 10 second intervals. When any of those processes fails, i.e. crashes or hangs, `watchd` restarts that process at its initial state. It took 2 people 3 hours to embed and configure `watchd` for this level of fault tolerance in application C.

Another example is a cross-connection system which consists of several processes using shared memory for interprocess communication. One of these processes is a writer process which may modify some data structures in the shared memory and the others are reader processes which only read the data structures. Because of a hideous software bug, there is a slight chance that a reader may be reading a data structure while the writer is modifying it (e.g., manipulating the pointers for inserting a new data node). Consequently, the reader may receive a segmentation violation fault if the reader happens to read the pointer (a byte) while the writer is modifying it. In such a case, the reader will be rolled back and restarted by *watchd*. Once the reader is restarted, it will access the same pointer again. This time, however, the read operation will succeed because the writer has finished the modification.

Other potential uses of this kind of fault tolerance are in in general purpose local area computing environments for state-less network services such as `lpr`, `fingerd` or `inetd` daemons. Providing higher levels of fault tolerance in those services would be unnecessary.

10.5.2 Example 2

Level 2 - Failure detection, checkpointing, restart and recovery using `watchd` *and* `libft`:

Application N maintains a certain segment of the 800 number call routing information on a Sun server; maintenance operators use workstations running N's client processes communicating with N's server process using *sockets*. The server process in N was crashing or hanging for unknown reasons. During such failures, the system administrators had to manually bring back the server process, but they could not do so immediately because of the UNIX delay in cleaning up the socket table. Moreover, the maintenance operators had to restart client interactions from an initial state. Replacing the server node with fault tolerant hardware would have increased their capital and development costs by a factor of 4. Even then, all their problems would not have been solved; for example, saving the client states of interactions. Using `watchd` and `libft`, system N is now able to tolerate such failures. `Watchd` also detects primary server failures and restarts it on the backup server. Location transparency is obtained using `getsvrloc()` and `getsvrport()` calls in client programs and `ftbind()` in server program. `Libft`'s checkpoint and recovery mechanisms are used to save and recover all critical data. Checkpointing and recovery overheads are below 2%. Installing and integrating the two components into the application took 2 people 3 days. This application is running on 5 maintenance centers across the country.

10.5.3 Example 3

Level 3 - Failure detection, checkpointing, replication, restart and recovery using `watchd`, `libft` *and* `REPL`:

Application U is a real-time telecommunication network element which collects data from a switch, filters that data and stores them on a disk for several days. Other off-line operations systems access the stored data for billing and various other purposes. In addition to the previous requirements for fault tolerance, this product needed to get its persistent files on-line immediately after recovery of the failed application on a backup node. During normal operations on the primary server, `REPL` replicates all the critical persistent files on a backup server with an expected overhead of less than 14%. When the primary server fails, `watchd` starts the application U on the backup node and automatically connects it to the backup disk on which the persistent files were replicated. To distinguish a node failure from a link failure, `watchd` was configured to use an ethernet and a datakit connection for polling; see Figure 10.5. A fail-

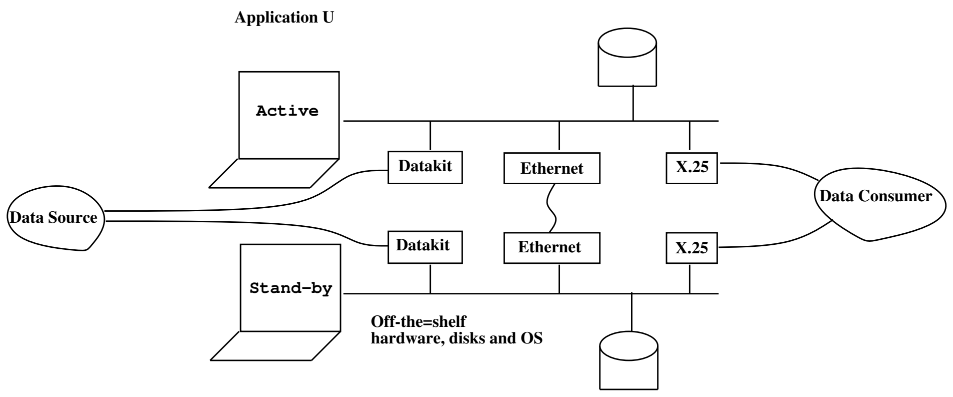

Figure 10.5 Architecture of application U

over takes place only when `watchd` on the backup site can not poll the primary site using both ethernet and datakit connections. The fail-over takes about 30 seconds to complete.

10.5.4 Other Possible Uses

The three software components, `watchd`, `libft` and `REPL`, can be used not only to increase the level of fault-tolerance in an application, as described above, but also to aid in other operations unrelated to fault-tolerance as described below.

- *On-line upgrading of software:* One can install a new version of software for an application without interrupting the service provided by the older version. This can be done by first loading the new version on the backup node, simulating a fault on the primary and then letting `watchd` dynamically move the service location to the backup node. This method assumes that the two versions are compatible at the application level client-server protocol.
- *Using checkpoint states and message logs to aid in debugging distributed applications:* In `libft`, all the checkpointed states, i.e. values in the critical data, and message logs can optionally be saved in a journal file. This journal can be used to aid in analyzing failures in distributed applications.

10.6 CONCLUSIONS

We defined a role, a taxonomy and tasks for software fault tolerance in the application layer based on availability and data consistency requirements of an application. We then described three software components, `watchd`, `libft`, and `REPL` to perform these tasks. These three components are flexible, portable and reusable; they can be embedded in any UNIX-based application software to provide different levels of fault tolerance with minimal programming effort. [5]

Experience in using these three components in some telecommunication products has shown that these components indeed increase the level of fault tolerance with acceptable increases in performance overhead.

ACKNOWLEDGMENTS

Many thanks to Lawrence Bernstein who suggested defining levels for fault tolerance, provided leadership to transfer this technology rapidly and encouraged using these components in a wide range of AT&T products and services. The authors have benefited from discussions, contributions and comments from several colleagues, particularly, Rao Arimilli, David Belanger, Marilyn Chiang, Glenn Fowler, Kent Fuchs, Pankaj Jalote, Robin Knight, David Korn, Herman Rao and Yi-Min Wang.

REFERENCES

[Als76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed services. In *Proc. of 2nd International Conference on Software Engineering*, pages 562–570, October 1976.

[Avi85] A. Avižienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[Bak92] M. Baker and M. Sullivan. The recovery box: using fast recovery to provide high availability in the UNIX environment. In *Proc. of Summer USENIX*, pages 31–43, June 1992.

[Ber92] L. Bernstein. On software discipline and the war of 1812. *ACM Software Engineering Notes*, 18, October 1992.

[Bia91] R. Bianchini, Jr. and R. Buskens. An adaptive distributed system-level diagnosis algorithm and its implementation. In *Proc. of 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 222–229, July 1991.

[Bit88] D. Bitton and J. Gray. Disk shadowing. In *Proc. of 14th Conference on Very Large Data Bases*, pages 331–338, September 1988.

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

[Fow93] G. S. Fowler, Y. Huang, D. G. Korn and H. Rao. A user-level replicated file system. In *Proc. of Summer USENIX*, pages 279–290, June, 1993.

[Gra91] J. Gray and D. P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, 1991.

[Hal90] J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the ACM*, 37(3):549–587, 1990.

[Hua89] Y. Huang and P. Jalote. Analytic models for the primary site approach to fault-tolerance. *Acta Informatica*, 26:543–557, 1989.

[Hua92] Y. Huang and P. Jalote. Effect of fault tolerance on response time — analysis of the primary site approach. *IEEE Transactions on Computers*, 41(4):420–428, 1992.

[Hua94] Y. Huang, P. Jalote and C. M. R. Kintala, Two techniques for transient software error Recovery. In M. Banâtre and P. A. Lee (Eds.), *Hardware and Software Architectures for Fault Tolerance: Experience and Perspectives*, Lecture Notes in Computer Science, No. 774, Springer Verlag, pages 159–170, 1994.

[Jal89] P. Jalote. Fault tolerant processes. *Distributed Computing*, 3:187–195, 1989.

[5] As of this writing (October 1994), these three components have been used within AT&T and are also available from Tandem Computers Incorporated as a product named HATS (High-Availability Transforming Software). Interested readers should contact the authors ([cmk,yen]@research.att.com) or Tandem Computers for further information.

[Lit88] M. Litxkow, M. Livny, and M Mutka. Condor — a hunter of idle workstations. In *Proc. of 8th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, June 1988.

[Lon92] J. Long, W. K. Fuchs and J. A. Abraham. Compiler-assisted static checkpoint insertion. In *Proc. of 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 58–65, July 1992.

[Nan92] A. Nangia and D. Finker. Transaction-based fault-tolerant computing in distributed systems. In *Proc. of 1992 IEEE Workshop on Fault-tolerant Parallel and Distributed Systems*, pages 92–97, July 1992.

[Pra86] D. K. Pradhan (ed.). *Fault-Tolerant Computing: Theory and Techniques*, volumes 1 and 2, Prentice-Hall, 1986.

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

[Sal84] J. H. Saltzer, D. P. Reed and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[Sat90] M. Satyanarayanan. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, C-39:447–459, 1990.

[Shr85] S. K. Shrivastava (ed.). *Reliable Computer Systems*, Chapter 3, Springer-Verlag, 1985.

[Sie92] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems Design and Implementation*, Chapter 7, Digital Press, 1992.

[Sul92] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *Proc. of 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 475–484, June 1992.

[Wan93] Y. M. Wang, Y. Huang and W. K. Fuchs. Progressive retry for software error recovery in distributed systems. In *Proc. of 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 138–144, June 1993.