Available online at www.sciencedirect.com
SCIENCE DIRECT®

The Journal of Systems and Software 71 (2004) 127–141

The Journal of Systems and Software

www.elsevier.com/locate/jss

NT-SwiFT: software implemented fault tolerance on Windows NT

Deron Liang [a,*], P. Emerald Chung [b,1], Yennun Huang [c], Chandra Kintala [d], Woei-Jyh Lee [e], Timothy K. Tsai [d], Chung-Yih Wang [c]

[a] *Department of Computer Science, National Taiwan Ocean University, Keelung 202, Taiwan*
[b] *Siebel Systems, Suite 2100, 411 108th Ave NE, Bellevue, WA 98004, USA*
[c] *AT&T Research Labs, 180 Park Avenue, P.O. Box 971, Florham Park, NJ 07932, USA*
[d] *Network Software Research, Avaya Labs, 233 Mount Airy Road, Basking Ridge, NJ 07920, USA*
[e] *Department of Computer Science, University of Maryland, College Park, MD 20740, USA*

Received 1 September 2001; received in revised form 6 February 2002; accepted 2 May 2002

Abstract

Today, there are increasing demands to make application software more tolerant to failures. Fault-tolerant applications detect and recover from failures that are not handled by the application's underlying hardware or operating system. In recent years, an increasing number of highly available applications are being implemented on Windows NT. However, the current version of Windows (NT4.0, 2000) and its utilities, such as Microsoft Cluster Server (*MSCS*), do not provide some facilities (such as transparent checkpointing, and message logging) that are needed to implement fault-tolerant applications. In this paper, we describe a set of reusable software components collectively named software implemented fault tolerance (*NT-SwiFT*) that facilitates building fault-tolerant and highly available applications on Windows NT, 2000. *NT-SwiFT* provides components for *automatic error detection and recovery, checkpointing, event logging and replay, and communication error recovery,* and *incremental data replication*. Using *NT-SwiFT*, we conducted fault injection experiments on three commercial server applications—Apache web server, Microsoft IIS web server, and Microsoft SQL—to study the failure coverage and the overhead of *NT-SwiFT* components. Preliminary results show that *NT-SwiFT* can detect and recover more application failures than *MSCS* does in all three applications.
© 2002 Elsevier Inc. All rights reserved.

Keywords: Windows NT; Microsoft Cluster Server; Software implemented fault tolerance; Automatic error detection and recovery, checkpointing, event logging and replay, communication error recovery, and incremental data replications

1. Introduction

To achieve high reliability and availability in an application, two types of techniques are usually used, namely, transaction processing (Gray and Reuter, 1993), and process replication (Birman, 1996; Huang and Kintala, 1993). These two techniques are not mutually exclusive and can be used together while needed. Transaction processing is the most widely used technique for fault tolerance among commercial fault-tolerant products, in particular, for applications that are concerned with data consistency. For example, applications in the financial industry belong to this category. With a transactional processing system, applications usually have a well-defined transaction boundary, such as updating a record or establishing a communication channel. When a fault occurs, both the client and server abort the on-going transaction and each remains at the previous (consistent) state.

Process replication technique is suitable for applications that concern with the availability and reliability of a continuing service. This technique often assumes that there is a group of identical servers running in the networks. The implementation of process replication normally relies on one of two techniques, *atomic multicasting* and *checkpoint/message-logging*, to make process states consistent. *Atomic multicasting* assures that all servers in the group receive the same sequence of client

[*] Corresponding author. Tel.: +886-2-24-622-192; fax: +886-2-24-623-249.
E-mail address: drliang@iis.sinica.edu.tw (D. Liang).
[1] This work was done while these authors were with Bell Laboratories, Lucent Technologies, Inc.

0164-1212/$ - see front matter © 2002 Elsevier Inc. All rights reserved.
doi:10.1016/S0164-1212(02)00154-1

invocations. If the replicas are identical, it is expected that they all arrive at the same (consistent) state after executing the same sequence of invocations if they started with the same state. With *checkpoint/message-logging*, the state of a server is checkpointed onto backup servers or on stable storage from time to time. The received messages may also be logged for recreating state change. When a failure occurs, the failed server process is stopped. Then, either a backup server is promoted to the primary, or a new process is created to be the new primary. The state is recovered by loading its last checkpoint and by replaying its logged messages. These two techniques address three replication management schemes: *cold, warm* and *hot* (Liang et al., 1999; OMG, 1999).

Cold replication. With a *cold replication* scheme, there is only one active copy of a fault-tolerant process, called primary replica, in the replica group. If the primary fails, a backup server is promoted to the primary (a *fail-over*), and the state of the new primary is restored to the previous checkpointed state.

Warm replication. With a *warm replication* scheme, one or more backup processes run on a network, and the primary process periodically checkpoints its state to its backup processes. Only the primary process can provide services to client applications; the backup processes can only receive checkpoint messages from the primary process. If the primary process fails, one of the backups quickly becomes the primary and resumes services.

Hot replication. [2] With a *hot replication* scheme, all of replicas in the group are considered as primary. This technique usually assumes the replicated servers in the group are identical and have deterministic behavior. Furthermore, it usually requires that there exist an atomic broadcast mechanism so that the operations of replica servers are running synchronously with synchronized messages. When a failure occurs with one server, the failure is masked and the computation continues if there is one server running. No rollbacks are necessary on either the client or server.

The process replication technique allows for faster recovery than transactional processing for non-transactional applications, such as switching systems and PBX's. Clients may experience some delay during a recovery, but no rollback is involved. Thus, it is ideal for telecommunication applications that constantly manage or monitor some resource. It is also suitable for applications that incur long transactions or do not satisfy transaction property, such as atomicity or isolation. Our experience shows that checkpointing and message logging is most suitable to achieve high availability in those applications (Huang and Wang, 1995).

[2] Hot replication management is not supported in *SwiFT* for Windows beta release.

We have been working on a set of reusable modules for building reliable and fault-tolerant applications for over six years. The set of modules is called software implemented fault tolerance (*SwiFT*) (Huang and Kintala, 1993). *SwiFT* has been embedded in many telecommunication systems to improve system availability. It contains a collection of daemon processes and libraries. *SwiFT* can be used to handle both client-side and server-side error recoveries. The design philosophy of *SwiFT* is to make the client error recovery as transparent as possible and provide a set of fault tolerance APIs to be embedded into server programs. This philosophy has proven to be a key to the success of the *SwiFT* since developers in Lucent often have access to the source code of server programs but have no control of client programs developed by other companies.

NT-SwiFT components were originally designed to run on UNIX platforms. The Unix version contains *watchd*, *libft*, and *Winckp* components (Huang and Kintala, 1993; Huang and Wang, 1995; Wang et al., 1995), which have been widely used by AT&T and Lucent. Because of the needs to make applications running on Windows NT more reliable, we started the *SwiFT* for Windows NT (called *NT-SwiFT*) work in 1996. Since Windows NT and Unix are very different in process and kernel architectures, we could not port the *Unix-SwiFT* code directly onto Windows NT, even with a Unix emulator tool for Windows such as UWIN (Korn, 1997). As a result, all the software has to be developed from scratch.

This paper describes the *NT-SwiFT* components and compares the differences in the UNIX and NT implementations. *NT-SwiFT* employs the process replication technique using checkpointing and message logging for fault tolerance. It provides application monitoring and failure recovery, checkpoint and message logging, file replication, windows event logging and replay in case of a failure. We also describe some applications using these components and discuss how to leverage NT system services and cope with some missing features.

The usefulness of these facilities is evaluated by a series of fault injection experiments to examine the failure detection and recovery capability of *NT-SwiFT*. Three commercial server applications are used as benchmark applications in these experiments: Apache web server version 1.1.3 for Win32, Microsoft IIS web server version 2.0, and Microsoft SQL version 7. Our experiments demonstrate that *NT-SwiFT* performs better than *MSCS* in failure coverage, the number of faults tolerated for three server applications.

2. Building fault-tolerant applications with NT-SwiFT components: the examples

NT-SwiFT is a set of re-usable software modules for building reliable, fault-tolerant applications on Win-

dows NT. These components can either stand-alone or be integrated into existing commercial products to enhance their fault tolerance and scalability. *NT-SwiFT* is geared especially for process and application replications using warm and cold replication schemes. *NT-SwiFT* detects hang and abend (abnormal end) failures in addition to crash failures. In this section, we introduce the software architecture of *NT-SwiFT* and the functions of its components. We use two examples, a server example and a client example, to demonstrate how applications become fault-tolerant using *NT-SwiFT* components in Section 2.1 and 2.2. These two examples also illustrate the interaction among those components in the complete fault-tolerance life cycles of the client and server applications.

As shown in Fig. 1, *NT-SwiFT*'s components include:

1. *Watchd*: for process failure detection, recovery, replication management, and distributed system services. *Watchd* contains a GUI for system configuration as shown in Fig. 2.
2. *Libft*: for data checkpointing, communication messages logging and recovery,
3. *REPL*: for on-line file replication and disaster recovery, and

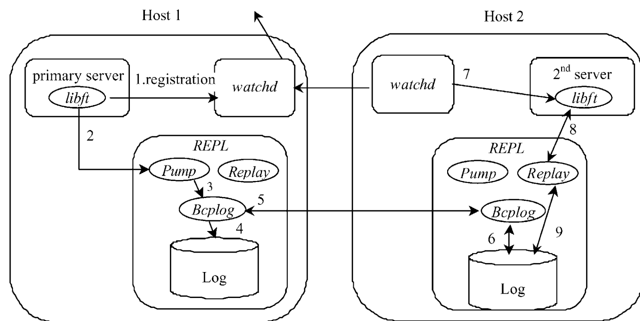

Fig. 1. The *NT-SwiFT* components and the software architecture.

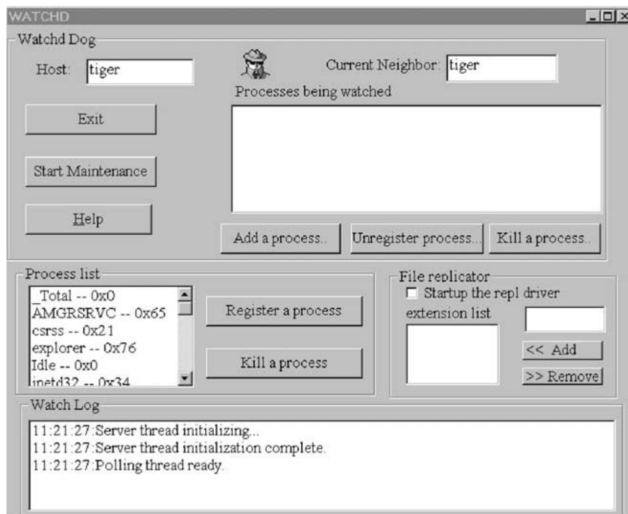

Fig. 2. The watchd GUI.

4. *Winckp*: for transparent process checkpointing and mouse/keyboard events logging and replaying.

2.1. Using NT-SwiFT to build a reliable server application

Suppose an application server wishes to achieve the following fault-tolerance goals:

1. It is able to tolerate failures such as server crash, server hang, and host crash.
2. It applies the cold backup policy.
3. It is able to checkpoint (automatically) its critical volatile data in the main memory periodically to local hard disk, so that it can continue its service from the last checkpoint should a failure occurs.
4. It keeps its persistent data in local files, thus the file content shall be replicated to the backup host so that this server can migrate it should the current host crashes.

An application typically experience several phases in its fault-tolerance life cycle with *NT-SwiFT* components. These phases are fault-tolerance configuration, server execution, failure detection and recovery. Fig. 1 illustrates how *NT-SwiFT* components interact with each other during these phases; whereas Fig. 3 gives examples to use *libft* APIs to achieve the fault-tolerance goals. Interested readers can find detail information of *libft* API in (Lucent, 1999).

In the configuration phase, a fault-tolerant application process can register itself to *watchd* via either *watchd* GUI (see Fig. 2) or *libft* APIs such as *FtRegWatch()* (see Fig. 3 for an example). *Watchd* pings the server process periodically in order to detect a server crash after the registration. To detect machine failures, *watchd* runs on every machine in a network and uses an adaptive diagnosis protocol (Huang and Kintala, 1993), i.e., each *watchd* pings its neighbor *watchd*; if its neighbor fails, *watchd* pings its next neighbor and so on. To detect a server hang, the thread of a server process that intends to

```
Main(){

  HANDLE fd;

  FtRegWatch();            // Registration to watchd that monitors the process

  FtHbeat(t_id, 300);      // libft APIs to register to watchd for process hang detection
   SomeWorks(){…};  // The application is considered hung if it doesn't come out of
                   //  this critical region within 300 seconds.

  Char *cm= t_ftmalloc(1024);  // libft API to create critical data for checkpoints
  t_checkpoint(t_id);           // libft API to checkpoint critical data to persistent
storage

  FtCreateFile(fd, …);          // libft APIs for file operations that are later logged and
replicated
  FtWriteFile(fd, …);           // by REPL

  FtDelWatch();                 // libft API to un-register itself
}
```

Fig. 3. Sample codes that illustrate the use of *libft* APIs.

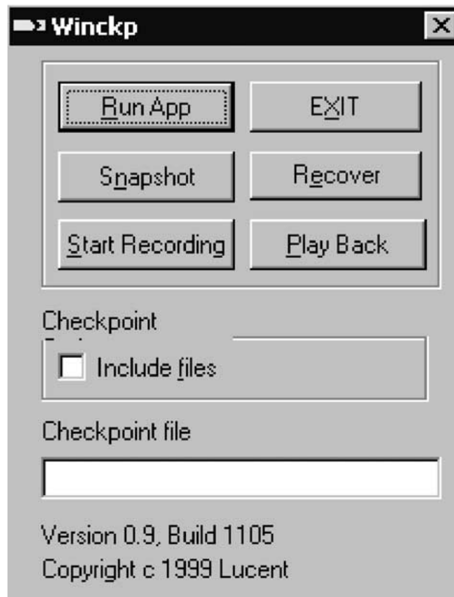

Fig. 4. The GUI of *Winckp*.

enter a critical section needs to notify the *watchd* with a timeout value. An application can call the *libft* API *FtHbeat()* that takes a thread *id* and a timeout value as arguments. A server process is considered hung if *watchd* does not receive a heartbeat from the server within a given interval. Fig. 3 illustrates the use of these *libft* APIs (Fig. 4).

In the execution phase, a fault-tolerant application can save its volatile data (in memory) and persistent data (in file) via *NT-SwiFT* components. For a stateful server, a consistent state needs to be recovered after *watchd* detects and restarts the server after failures. *Libft* allows an application to select critical data from data segment (e.g. global or static variables) and heap (e.g. data allocated via *t_ftmalloc()*). The critical data can be saved to a file or to another process on a local or a remote machine, or a protected segment of its own virtual address space via API *t_checkpoint()*.

A server program may also create and update files in order to manage its persistent data during execution. To make a *fail-over* possible in a share-nothing environment, component *REPL* can be used to do file replication. As shown in Fig. 1, *REPL* implementation includes one module to intercept file system calls (*libft*), and three daemon processes to send messages and to replay file system calls. *Libft* provides *FtCreateFile()*, *FtReadFile()*, and *FtWriteFile()* APIs to automatically log the file operations messages with the three daemon processes. *Pump* server on the primary host is used for pumping the file data to the *Bcplog* server that saves the received data into a log file on the backup host. *Replay* server replays or replicates the files from the log file generated by *Bcplog*. The steps 2 to 6 shown in Fig. 1 illustrate this scenario.

Next, we describe the recovery in cases of server hang, server crash, and host crash. To recover a hung or a crashed server, *watchd* kills and restarts the hung server process. The server resumes its service after *watchd* restores its critical data from latest checkpoint into its main memory via *libft* API *t_recover()*. For a machine crash recovery, *watchd* does a fail-over of the server application by bringing up a cold copy of the server application on its backup machine. The server resumes its service after *watchd* restores its critical data from latest checkpoint into its main memory (via *t_recover()*) and *REPL* replays its message logs (see the steps 7, 8 and 9 in Fig. 1).

2.2. Using NT-SwiFT to build a reliable client application

The design philosophy of client-side recovery components is to make clients/users transparent to client program failures. A client program may accept a user's keyboard and mouse inputs and, at the same time, talk to one or more server programs running on server machines via communication channels. When a client application fails (either due to a program failure, an OS failure or a machine failure), all input data is lost and all communication channels are broken. Without any fault tolerance facility, the user has to restart the client program, re-establish communication channels and redo all the inputs. This could result in a long recovery time and a frustration of the user. *NT-SwiFT* provides fault tolerance utilities which (1) detect failure of a client program; (2) automatically restart a client program at failure recovery; (3) re-establish communication channels to server programs; (4) replay all the user inputs and bring the client program back to the state just before the failure occurred. One can embed these *NT-SwiFT* components into client programs without modifying the client source code.

The first component in *NT-SwiFT* for the client-side error recovery is *watchd*. Once *watchd* detects a failure, it restarts the application program automatically. If the client application fails too often (more than a threshold given to *watchd*), *watchd* reboots the machine and then restarts the application. The second component is *Winckp* that can be used to transparently checkpoint an application program state into a file or another process. In recovery, the checkpointed state is restored back to the client application memory. In addition, *Winckp* can be used to log input events from the mouse and keyboard of a client machine. In recovery, the logged input events are replayed to recover the client input data. The last component is *libft* library. *Libft* is used to intercept *winsock* function calls in client applications for checkpointing communication endpoints and logging outgoing messages. During the recovery process, *libft* reestablishes communication endpoints using the

checkpointed information. If necessary, *libft* replays the logged messages.

3. Implementation issues

The mechanisms of *NT-SwiFT* derive from those of *UNIX-SwiFT*. However, due to the differences between UNIX and NT, their implementations are very different. As mentioned in Korn (1997), there are many ways to port UNIX applications to Windows NT. In fact, the first porting effort we tried was to use the UWIN developed by D. Korn in AT&T Labs. However, we later decided to re-implement the *NT-SwiFT* components from scratch due to the following considerations:

1. Some of the *UNIX-SwiFT* components such as *REPL*, *Winckp* and *libft* depend on UNIX internals. They cannot be ported directly by using a library mechanism such as UWIN (Korn, 1997) or a subsystem such as OpenNT (Walli, 1997).
2. We did not want to depend on any third-party software.
3. We wanted to enhance *watchd* with a Windows GUI and threads.
4. To understand how NT applications fail, we need to have intimate knowledge of the NT architecture. Re-implementing *SwiFT* on NT using native NT system services helped us to understand the NT internals better.

In this section, we describe how *NT-SwiFT* components are implemented and the differences in implementations between the *UNIX-SwiFT* and the *NT-SwiFT*.

3.1. Watchd

Watchd is capable of detecting three kinds of failures; process crash failures, machine failures and process hang failures. Process crashes can be detected by the facilities provided by an operating system. *Watchd* also receives the heartbeats sent from an application process in order to inform *watchd* that the process has not hung. In our implementation, *watchd* uses *OpenProcess()* and *WaitForMultipleObjects()* to detect an application crash (*vs. kill(pid, 0)* and SIGCHLD in UNIX). It uses non-blocking socket calls and time-outs to detect machine failures. *Watchd* detects a process hang by listening to its heartbeats. An application can send its heart beats to *watchd* by calling *FtHbeat()* functions in *libft*.

The major differences between the UNIX version and the NT version are the process/thread models and the inter-process communications. The UNIX version uses three processes to implement *watchd*. The three processes communicate using socket messages and UNIX signals (*SIGUSR1* and *SIGUSR2*). In NT, since there are no corresponding *SIGUSR1* and *SIGUSR2* signals, all functions of *NT-watchd* are implemented in one process with four NT threads. The first thread is the polling thread to detect failures; the second one is the GUI thread for system configuration and display; the third one is the service thread that accepts requests from applications and from other *watchds*; the last thread is the heart-beat thread that accepts application heart beats for hang detection. Threads are synchronized using semaphores and critical sections. In addition, as shows, *watchd* also refreshes a process list periodically, which is similar to what NT task manager provides in order to manage a process on the machine. The main advantage of using threads is its low performance overhead—most of the interprocess communication overhead in *UNIX-watchd* modules is removed. However, the main disadvantage of using threads is that self-recovery and fault containment are difficult, if not impossible, to achieve. For example, in *UNIX-watchd* a crash of any module (a process) can be recovered automatically and the failure is transparent to *watchd* clients. However, in *NT-watchd*, any crash of a *watchd* module (a thread) causes the entire *watchd* process to crash.

3.2. Libft

Libft contains three sets of functions—the first set is for dynamic memory allocation and recovery, the second set of functions is for system configuration and the last set of functions is to intercept *winsock* calls and kernel calls. The implementations of the first two sets of functions are almost identical on both UNIX and NT. More information on *libft* APIs and their implementation can be found in Huang and Kintala (1993). However, implementations of the last set of functions (intercepting calls) between UNIX and NT are very different. In UNIX, the interception of system and socket calls is done by using the dynamic shared library mechanism (i.e. *dlopen()* and *dlsym()*). On Windows NT, interception of system calls is achieved by modification of import address tables and by the library injection mechanisms (Richter, 1997a,b,c). To checkpoint and recover kernel states, *NT-SwiFT* has to intercept all NT calls which create file handles, process handles, thread handles, socket handles and windows handles. It also has to intercept socket calls for messages logging and replay and file system calls which change files contents and attributes. A complete list of kernel and *winsock* calls intercepted by *libft* is illustrated in Table 1.

3.3. REPL

REPL implementation includes one module to intercept file system calls (*libft*), and three daemon processes

Table 1
A summary of NT system calls that are intercepted in *NT-SwiFT*

WindowsSocket (Wsock32.dll)	Accept, bind, closesocket, connect, ioctlsocket, listen, setsockopt, shutdown, socket, send, recv, recvfrom, sendto
FileOperations (Kernel32.dll)	CopyFileA, CopyFileExA, CopyFileExW, CopyFileW, CreateFileA, CreateFileW, CreateFileMapping, DeleteFileA, DeleteFileW, MoveFileA, MoveFileW, MoveFileExA, MoveFileExW, OpenFile, ReadFile, ReadFileEx, ReadFileScatter, SetFilePointer, UnlockFile, UnlockFileEx, WriteFile, WriteFileEx, WriteFileGather
Directory (Kernel32.dll)	CreateDirectoryA, CreateDirectoryExA, CreateDirectoryExW, CreateDirectoryW, RemoveDirectoryA, RemoveDirectoryW, SetCurrentDirectoryA, SetCurrentDirectoryW
Process and Thread (Kernel32.dll)	CreateRemoteThread, CreateThread, CreateProcessA, CreateProcessW, ExitProcess, ExitThread, OpenProcess, TerminateProcess, TerminateThread
Event (Kernel32.dll)	CreateEventA, CreateEventW, OpenEventA, OpenEventW, ResetEvent, SetEvent
NamedPipe (Kernel32.dll)	ConnectNamedPipe, CreateNamedPipeA, CreateNamedPipeW, DisconnectNamedPipe, SetNamedPipeHandleState, WaitNamedPipeA, WaitNamedPipeW
MailSlot (Kernel32.dll)	CreateMailslotA, CreateMailslotW, SetMailslotInfo
Mutex (Kernel32.dll)	CreateMutexA, CreateMutexW, OpenMutexA, OpenMutexW, ReleaseMutex
Semaphore (Kernel32.dll)	CreateSemaphoreA, CreateSemaphoreW, OpenSemaphoreA, OpenSemaphoreW, ReleaseSemaphore
CriticalSection (Kernel32.dll)	EnterCriticalSection, InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, LeaveCriticalSection
DLL Library (Kernel32.dll)	FreeLibrary, FreeLibraryAndExitThread, LoadLibraryA, LoadLibraryExA, LoadLibraryExW, LoadLibraryW
Other handles (Kernel32.dll)	CloseHandle, DuplicateHandle, SetHandleCount, SetHandleInformation

to send messages and to replay file system calls. The implementations of the daemon processes are very similar to the UNIX ones. However, the facilities for intercepting file system calls are very different.

For Unix version, dynamic shared library mechanism (*dlopen()* and *dlsym()*) is used to intercept and replay file system calls whereas the mechanism of library interception is used in NT version. The file interception library is a wrapper of Win32 File API that records all file system calls and data to the local *Pump* server. A remote thread in the application process is created to load the file interception library when the process is started. The file interception library supports two different replication approaches: *replay-mode* and *copy-mode*. When a file operation is invoked, the library intercepts the changes and sends messages to remote backup machines immediately. The *copy-mode* replication is achieved by copying the files operated by the application; the *Pump* daemon replicates the working files until the application stops any file operation for a configurable time interval.

On remote backup machine, the behavior is very similar for both versions. *REPL* daemons on remote backup machines replay the changes to update the files. *REPL* daemons are all user-level processes, which send I/O messages, log I/O messages and replay I/O messages between the primary host and the backup host. These daemon processes handle link failures, machine failures, I/O failures on the backup machine, messages lost, etc. so that the replicated files are consistent as long as they can be accessed. *Libft* also uses *REPL* modules to make checkpoint files replicated on all backup machines.

3.4. Winckp

The main purpose of *Winckp* is to transparently take a snapshot of the memory space of an application process so that the data in volatile memory can be recovered after failure. *Winckp* leverages a set of Win32 functions such as *SetThreadContext()*, *WriteProcessMemory()* that allow a process to alter another process. Microsoft created many of these functions for use by debuggers (Richter, 1997a,b,c). In addition, *Winckp* uses *libft* to intercept file system calls and window object calls if necessary. When an application takes a checkpoint, it has to not only save its memory contents but also its file contents and attributes. The window handles are also saved if the protected application has graphical user interface. When the application rolls back to its previous checkpointed state, it has to undo all file updates after the last checkpoint as well as restore its memory content and construct a mapping between new and old window handles if necessary. The file rollback mechanism uses the *libft* interception routines as described in Wang et al. (1995). *Winckp* also records all keyboard/mouse messages logging and playback these messages if needed.

In this section, we discuss how *Winckp* deals with threads, memory, files, window objects for check-

pointing and keyboard or mouse events for message logging and playback in an application process. Though *Winckp* provides powerful ability in the aspects above, there are limitations. We will discuss the limitation later.

Winckp starts the target application by calling *CreateProcess( )*. *Winckp* and the target application run as two separate processes. NT allows a process to obtain and set the thread context for threads in a different process. *Winckp* obtains and restores the thread context using *GetThreadContext( )* and *SetThreadContext( )*. To take a snapshot, *Winckp* suspends all threads and stores state information into files using system call *SuspendThread( )*. During a rollback operation, all threads are also suspended. The state information is restored from files. *Winckp* resets all thread contexts then resumes all threads. If the application process has died, *Winckp* starts a new process, suspends its main thread, restores all state information, and then resumes all threads. *Winckp* avoids checkpointing a thread in the *waiting* state since the kernel states cannot be recovered in such a condition. Instead, *Winckp* defers the checkpoint until the thread returns to *ready* state. To detect a thread in the *waiting* state, *Winckp* examines the return attribute, *Suspension Count,* from the system call *SuspendThread( )*. The thread is in the *waiting* state if this count is greater than one.

Winckp saves an application's memory including data, heap and stack. The memory image is obtained and restored using *ReadProcessMemory( )* and *WriteProcessMemory( )*. *Winckp* determines the address and the amount of memory to save as follows. An NT process has 2 GB of private address space, ranging from 0×00010000 to $0 \times 7FFEFFFF$ (Richter, 1997b), but not every region in this space needs to be saved. The *VitualQueryEx( )* system call allows us to examine the space region by region. It is necessary to save a memory region only if both its write access is enabled and its physical storage is committed (Richter, 1997c). *Winckp* also stores the MEMORY_BASIC_INFORMATION structure along with each memory region. During a rollback operation, all threads are suspended. *Winckp* calls *WriteProcessMemory( )* to restore the memory content.

To checkpoint and recover files, *Winckp* uses a system call interception mechanism to intercept file-related system calls, such as *CreateFile( )*, *WriteFile( )*, etc. *Winckp* records each handle value and the parameters used to create the handle. In recovery, *Winckp* recreates all the handles by replaying the calls with the recorded parameters. An issue is that the values of the recovered handles may be different from their checkpointed values. *Winckp* creates a mapping of old handle values to the new handle values. The system call interception routines replace the old handle values with the new values before making the real call.

To make sure file content is consistent, *Winckp* generates idempotent undo logs for all file updates. The undo logs are played at the recovery time.

Winckp intercepts a subset of functions in USER32.dll and GDI32.dll, such as *CreateWindow( )* and *CreateDC( )*. The recovery of window handles and other GDI objects has been a difficult task due to the dependency among these objects and the fact that they are not isolated from other processes or kernel.

To recover a window handle, *Winckp* first starts a new process and let the new process create all related windows. Afterwards, *Winckp* creates a mapping between the old window handle value and the new window handle value. We replace the old value by new value in the checkpoint file before restoring it back to memory. It works for a small set of applications, such as winmine.exe and solitaire.exe. However, the recovery procedure becomes tedious and error-prone if a GUI involves hundred of sub-windows. This issue might be addressed if window handles are managed in a hierarchical fashion, i.e., each sub-window handle is private to its parent window. A window handle can only be accessed through the window hierarchy starting at the root window. As a result, only one window handle, i.e., the root window, needs to be logged and recovered should a failure occur.

Also, we are primarily interested in system events related to keyboard strokes and mouse inputs. Win32 subsystem provides a hook that allows a user application to monitor system events such as keyboard strokes, window messages, debugging information, etc., and to react to these events through a user-defined callback procedure. User application may specify those system events of interest and install the corresponding callback procedures via the Win32 API. *Winckp* captures those events by calling *SetWindowsHookEx( )* with flag WH_JOURNALRECORD, and all keyboard events and mouse events are copied from the Win32 system's message queue to our callback procedure. These events are kept in a temporary file. To replay, we insert these events one after the other in their timestamp order back to Win32 system message queue by installing the WH_JOURNALPLAYBACK callback procedure. The Win32 system temporarily disables the inputs from keyboard and mouse when the WH_JOURNALPLAYBACK callback procedure is installed. It executes only the events fed from the callback procedure until our event log is up and the WH_JOURNALPLAYBACK callback procedure is un-installed.

Winckp currently has the following limitations:

1. Thread ID and thread handle value: Current *Winckp* rolls back the number of threads to the number in the last snapshot state. A dead thread is recreated. The recreated thread usually has a different thread handle and thread ID. Applications depend on thread handle and thread ID may not work.

2. Synchronization objects: All kernel objects used for synchronization are not recovered if they no longer exist, such as events and mutexes.
3. Interprocess communication: The current implementation does not support recovery of interprocess communications, such as COM calls or socket calls.
4. Playback non-determinism: Since playback involves with the entire display area, it inevitably affects GUI windows of other processes. There is no guarantee from *Winckp* that the state of all windows at the beginning of playback is the same as the beginning of recording. In addition, changing the playback speed may also affect the outcome. For instance, the Windows may interpret two single clicks as one double click or vice versa.

4. Fault injection and overhead experiments

The ability of *NT-SwiFT* to increase the availability of application servers has been demonstrated with many commercial products. Examples are

1. Microsoft IIS web server: Suppose it is executing on a machine that crashes. *NT-SwiFT* restarts IIS on another machine, and all incoming web requests are automatically redirected to the new machine.
2. Microsoft Word: Suppose it is being used to create a document that is being saved to a disk that crashes. As the document is being created, *NT-SwiFT* duplicates the changes to a disk on a backup machine.
3. A long-running simulation program: Suppose it contains a bug that results in a memory leak that will cause the program to crash. *NT-SwiFT* detects the symptoms before the crash and rejuvenates the program from the most recent checkpoint of critical internal state. The program is eventually able to finish. The checkpointing overhead is minimal because only critical data is saved.

In order to quantify the capability of *NT-SwiFT* in terms of availability enhancement, experiments were designed to compare it with the capability of *MSCS* (Microsoft, 1999). The results are included in Section 4.1. *NT-SwiFT* incurs overheads while helping application servers to increase their availability. A series of experiments have been conducted to measure the overheads due to process monitoring (*watchd*), checkpointing (*libft* and *Winckp*), and file replication (*libft* and *REPL*). Those overhead results are in Section 4.2.

4.1. Fault injection experiment

In order to quantify the ability of *NT-SwiFT* to increase system availability, we conducted fault injection experiments using the DTS fault injection tool (Tsai et al., 2000). The design of the DTS tool is geared toward testing of client–server systems, which often have high availability requirements. The DTS tool injects faults into the server application, and the response observed by the client determines the server availability.

In our experiments, we tested the following server applications:

1. Apache web server version 1.1.3 for Win32.
2. Microsoft IIS web server version 3.0.
3. Microsoft SQL Server version 7.0, Enterprise Edition.

We chose these three applications to test a mix of Microsoft and non-Microsoft servers. Apache and IIS are web servers from two different vendors and thus provide an interesting point of comparison. The Enterprise Edition of Microsoft SQL Server 7.0 is cluster-aware, meaning that it uses the Cluster API (Libertone, 1999) to explicitly support *MSCS*.

Each of these applications was subjected to injected faults in the following configurations:

1. With no external fault tolerance software.
2. With *MSCS*.
3. With *NT-SwiFT*.

All experiments were performed on a 100 MHz Pentium system with 48 MB of RAM and Windows NT Enterprise Server 4.0 with Service Pack 4.

Faults were injected by intercepting the first call to each of the 681 functions in Kernel32.dll. Then, the parameter values of those function calls were corrupted. This fault injection technique was chosen for its ability to cause a variety of failure scenarios, including not only process terminations but also hangs and data corruption. Fault tolerance solutions such as *MSCS* and

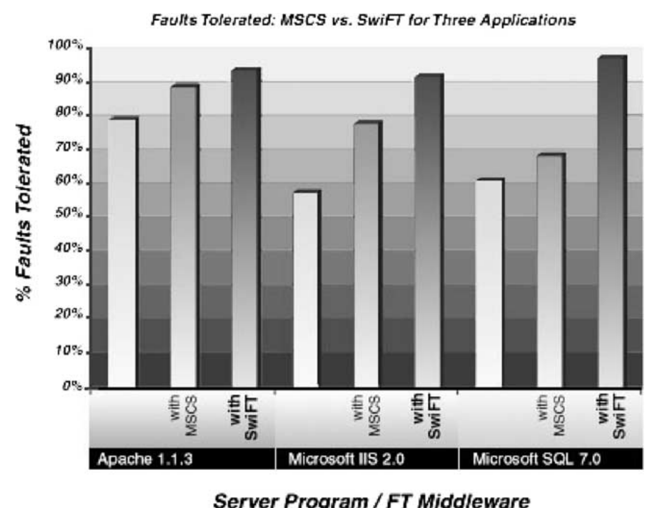

Fig. 5. Empirical reliability of *NT-SwiFT* vs. *MSCS*.

Table 2
List of faults tolerated by *NT-SwiFT* but not *MSCS*

Function name	Parameter:Fault type, ...
CreateEventA	3:1, 3:2, 4:2
CreateEventW	1:1, 1:2, 2:0, 4:1, 4:2
CreateFileMappingA	2:1, 2:2, 6:1, 6:2
CreateFileMappingW	2:1, 6:1
CreateFileW	1:0, 1:1, 1:2, 3:2, 4:1, 4:2, 6:1, 6:2
CreateIoCompletionPort	2:2
CreateMutexA	1:1, 1:2, 3:2
CreateMutexW	1:1, 1:2, 2:0, 3:1, 3:2
CreatePipe	1:0, 1:1, 1:2, 3:1, 3:2
CreateSemaphoreA	2:1
CreateThread	1:1, 1:2, 3:1, 3:2, 6:1
ExpandEnvironmentStringsA	1:0, 2:2, 3:1, 3:2
FileTimeToSystemTime	1:0, 1:1, 1:2, 2:0
FindFirstFileA	2:1, 2:2
GetComputerNameW	1:0, 1:1, 1:2, 2:0, 2:1, 2:2
GetDateFormatA	4:2, 5:1, 5:2
GetDiskFreeSpaceW	1:2, 2:0, 2:1, 3:0, 3:1, 3:2, 4:0, 4:1, 4:2, 5:0, 5:2
GetDriveTypeW	1:2
GetFileInformationByHandle	2:0, 2:1, 2:2
GetFileSize	2:1, 2:2
GetFullPathNameA	2:0, 3:0, 4:2
GetLocalTime	1:1, 1:2
GetLocaleInfoA	1:0, 3:1, 3:2
GetLocaleInfoW	1:0, 3:1, 3:2
GetModuleHandleA	1:1, 1:2
GetOverlappedResult	2:0, 2:1, 2:2, 3:0, 3:1, 3:2
GetSystemInfo	1:1, 1:2
GetSystemTime	1:0, 1:1, 1:2
GetSystemTimeAdjustment	1:0, 1:1, 1:2
GetThreadTimes	2:0, 2:1, 2:2, 3:0, 3:1, 3:2, 4:0, 4:1, 5:0, 5:1, 5:2
GetTimeFormatA	4:2, 5:1, 5:2
GetTimeZoneInformation	1:0, 1:1
HeapAlloc	2:0, 3:0
HeapCreate	3:0
HeapFree	1:0, 1:1, 1:2, 3:1, 3:2
InitializeCriticalSection	1:1, 1:2
InterlockedDecrement	1:0, 1:1, 1:2
InterlockedExchange	1:0, 1:1, 1:2
InterlockedExchangeAdd	1:0, 1:1, 1:2
InterlockedIncrement	1:0, 1:1, 1:2
LCMapStringW	5:2
LeaveCriticalSection	1:0, 1:1
LoadLibraryA	1:0, 1:1, 1:2
LoadLibraryExA	1:1, 1:2
LoadLibraryW	1:1, 1:2
LocalReAlloc	1:0, 2:1, 2:2, 3:0
MapViewOfFile	2:0, 2:2
MapViewOfFileEx	2:0, 2:2
MultiByteToWideChar	2:2, 3:0, 3:1, 5:0, 5:1, 5:2
ReadFile	2:0, 2:1, 2:2, 4:0, 4:1, 4:2, 5:1, 5:2
ReadFileEx	1:0, 1:1, 2:2, 3:0, 3:1, 4:0, 4:1, 4:2, 5:1, 5:2
ReadFileScatter	1:0, 1:1, 2:2, 3:0, 3:1, 4:0, 4:1, 4:2, 5:1, 5:2
SetFilePointer	1:1, 1:2, 2:0, 2:2, 3:1, 3:2, 4:2
WaitForMultipleObjects	2:0, 2:1, 2:2
WaitForMultipleObjectsEx	1:1, 1:2, 2:0, 2:1
WaitForSingleObject	1:1
WaitForSingleObjectEx	1:1
WideCharToMultiByte	2:2, 3:0, 3:1, 5:0, 5:1, 5:2, 7:2
WriteFile	4:0, 4:1, 4:2, 5:0, 5:1, 5:2
WriteFileEx	1:0, 4:1, 4:2, 5:0, 5:1, 5:2
WriteFileGather	4:0, 4:1, 4:2, 5:0, 5:1, 5:2
lstrcmpW	2:2
lstrcmpiW	1:1

SwiFT often focus on detecting crashes at the machine, network, or process level. Such crashes are usually tolerated easily. In contrast, corruption of standard library function parameters produces more interesting failures.

We applied the same set of faults to all applications and fault tolerance configurations. Fig. 5 shows the percentage of these injected faults that was properly tolerated such that the server application was able to deliver the correct response. The value of *NT-SwiFT* and *MSCS* is reflected by the increased availability of the application servers. As expected, both *MSCS* and *NT-SwiFT* increase the failure coverage for all three applications. For example, the ability of Microsoft IIS server 3.0 to tolerate the injected faults improves by 21% and 35% with *MSCS* and with *NT-SwiFT*, respectively. Other experiments also indicate similar fault tolerance capabilities for *MSCS* and *NT-SwiFT*. Table 2 shows a list of faults that *MSCS* was not able to tolerate but which were tolerated by *NT-SwiFT*. Faults that are not listed in Table 2 were either tolerated by both *MSCS* and *NT-SwiFT* or not tolerated by either. Each fault is specified by three parameters:

(1) Function name
(2) Parameter
(3) Fault type

Fault type 0 resets all bits of the parameter to zero. Fault type 1 sets all bits of the parameter to one. Fault type 2 flips all bits of the parameter (i.e., one's complement).

In Table 2 all faults for the same function are listed in the same row. The function name is shown in the first column, and the parameter number and fault type are listed in the second column. For example, in the row containing CreateEventA, three faults are listed. The first fault involved the corruption of the 3rd parameter of CreateEventA with fault type 1. The second fault involved the corruption the third parameter of CreateEventA with fault type 2, and so forth. Each fault that is listed in Table 2 was injected while Apache, IIS, or SQL Server was executing. However, the application information was omitted from the table for conciseness.

More detailed results and descriptions of the experiments are given in Tsai et al. (2000). The DTS tool is available for download at http://www.bell-labs.com/projects/swift/ntdts.

4.2. Overhead

NT-SwiFT incurs overheads when its components are invoked to serve fault-tolerant applications. The overheads are due to process monitoring (*watchd*), checkpointing (*libft* and *Winckp*), and file replication (*libft* and *REPL*). *Watchd* is responsible for failure detection; this includes machine crashes, process crashes and process hangs. It also maintains information for management purpose, such as lists of running processes and threads, etc. This process also causes overhead. The process of file replication involves both *REPL* and *libft*. As described in Section 3.2 and 3.3, large number of messages that contains file operations and data need to be saved to the backup server using *REPL*. Therefore, the system load rises when *REPL* is operating. Checkpointing using *Winckp* is another source of overhead. *Winckp* takes a memory snapshot of an application when checkpointing is enabled. The overhead of checkpointing depends on the amount of critical data in the memory as well as the frequency of checkpointing.

A series of experiments were designed to measure the *NT-SwiFT* overhead. Our experimental environment consists of two server machines, a primary node and a backup node, and several client machines. The server machines are Pentium II 333 PCs each with 128 MB RAM and 6 GB disk space. All these machines are connected with a 100 Mbps fast Ethernet directly. Each experiment is repeatedly performed numerous times to collect a large enough set of experimental results so that a measure with 95% of confidence interval with interval half-widths of less than 3% can be obtained using statistics methods. [3]

As described earlier, *watchd* is primarily responsible of failure detection. To detect process crashes, *watchd* takes advantage of NT notification mechanism so that *watchd* is notified immediately each time when a process crashes. As a result, we suppose that the overhead of detecting a process crash is negligible. To detect machine crashes or process hangs, polling method is used. An application administrator can configure the polling interval using the *watchd* GUI (see Fig. 2). We design an experiment to compare the total execution time of a computational intensive application with and without *watchd* running. We set the polling interval of *watchd* to 10 s in this experiment, as many telecommunication applications demand the same requirement in our experience. We select a pattern recognition application that recognizes a pattern from a 2D image based on the "self-similarity" technology. A user first selects a few points on the 2D image as input parameters to this application. This application constructs a contour on the input 2D image based on the selected points, called contour points. The execution time of this application is linearly proportional to the number of selected contour points. Table 3 lists the total execution time of this application with or without *watchd* running for various contour points. Fig. 6 illustrates that the *watchd* overhead decreases from 0.72% to 0.48% as the execution

[3] The reason that we apply the statistics method is to enhance the validity of our measured data. It is usually the case that we observe two different measures when we perform the same experiment twice. This is due to various measurement errors, such as clock resolution, etc.

Table 3
Watchd overhead

	Points				
	6000	12000	18000	24000	30000
Execution time (s)					
With *watchd*	312.4863	622.1547	933.2653	1244.246	1561.703
Without *watchd*	310.2827	617.899	928.6183	1238.294	1548.734
Overhead (%)	0.7179	0.7102	0.6887	0.5004	0.4807

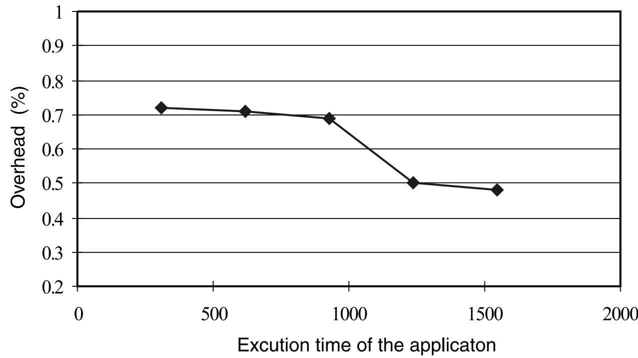

Fig. 6. The overhead of *watchd*.

time increases. This low level of overhead is primarily due to the implementation of *watchd*. Our experiments show that one polling roughly takes about 10 ms on a Pentium II 333 MHz machine, and the time period does not vary much when the number of fault-tolerant applications increases. Furthermore, *watchd* increases the CPU utilization by about 3% at the moment when polling takes place. In MSCS however, this administrative overhead increases linearly to the number of fault-tolerant applications. Recent reports indicate that excessive overhead could occur if there are more than a hundred of "shared resource" (or the fault-tolerant applications) running (Microsoft, 2002).

File replication is useful when the state of an application depends on the file data. The operations of file replication are achieved via *REPL* and *libft*, as described in Section 3.3. We select Microsoft Word as the benchmark application in our experiments to analyze the *REPL* overhead, since Word is a typical application involving with file operations. We measure the response time as the time period from clicking the *Save* button until the saving completes. We estimate the overhead caused by *REPL* by measuring the response delay of file saving with and without *REPL*. *REPL* supports two operating modes, copy mode and replay mode. In copy mode, the associated files are replicated only when there are no additional file operations for a predetermined interval (see Section 3.3 for details). We expect that a user "feel" no response time delay with the copy mode unless the user saves a file at the time when *REPL* is copying the file to its remote backup. We first analyze the overhead under such condition, and then we argue that the probability of this condition to happen is small in an NT environment. Fig. 7 compares the response time of file saving using Word with and without the continuous file copying by *REPL*. Fig. 8 illustrates the overhead under the condition that both local file saving (by the user) and remote copying (by *REPL*) are working at the same time. The overhead increases from 18% to 53% as the files sizes increases from 2 to 12 Mbytes. We claim however that a user rarely feels such impact from *REPL*. We have done a survey over five commonly used file types on an NT file server shared by 20 users in our laboratory. These types include files edited with Notepad (.txt), Word (.doc), PowerPoint (.ppt), Excel (.xls) and Acrobat (.pdf). We found that, among 20000-plus files, 98% of them are less than 2 Mbytes, 99% of them are less than 3 Mbytes, and 99.9% of them are less 10 Mbytes. It takes *REPL* only a few seconds to copy a 10 Mbytes file over a 100 MB LAN. Therefore the probability that a user saves a file while *REPL* is in action is less than 1% if the copy interval is set to 10 min. We conclude that the user rarely feels impact from *REPL* in the copy mode, however, the overhead could be significant when it occurs.

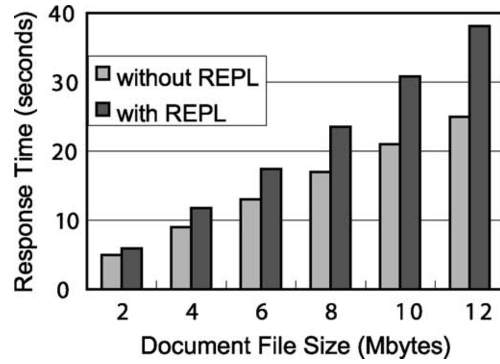

Fig. 7. The response time with copy mode.

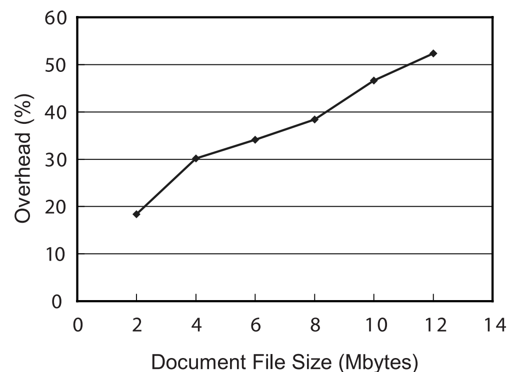

Fig. 8. The overhead of copy mode.

In replay mode, all file operations are intercepted by *libft's* interception library. The overhead of replay mode comes from three different sources. First, each intercepted file operation is wrapped into a data packet forwarded to *Pump* daemon, then *Pump* daemon forward the message to remote *Bcplog* daemon asynchronously. Second, the wrapped packet consists of header and data. The header identifies each file operation in replay log file for recovery purpose. The data contains information such as data of a write operation, the file pointer information, filename, etc. Third, if the application performs numerous I/O operations such as *ReadFile( )* and *SetFile Pointer( )*, *libft* needs to keep track of all file pointers and related information. In summary, the overhead of file replication can be significant in certain cases.

We have done similar experiments as in the copy mode of *REPL* to measure the overhead of replay mode. These experiments are done for the Word documents from 2 to 12 Mbytes. As Fig. 9 shows, the response time to save a Word document with 2 Mbytes size are 5.327 and 4.597 s with and without file replication. Fig. 10 indicates that the overhead of replay mode decreases from 16% to 7% as the file size increases. Based on our overhead analysis on replay mode, it suggests that the overhead is linearly proportional to the number of file operations, not the file size. Therefore, the relative overhead, i.e., overhead divided by file size, decreases with the file size provided that the same sequences of file operations are applied.

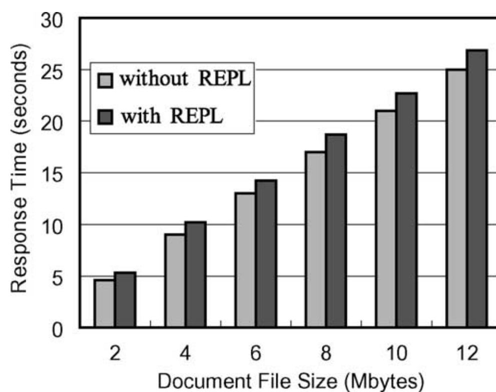

Fig. 9. Response time in replay mode.

Table 4
The checkpoint sizes of the hypercube simulator for different dimensions

	n						
	3	4	5	6	7	8	9
Size (MB)	1.2	3.2	8	20	45	103	211

To measure the overhead of checkpointing, we select a computation-intensive simulation program as our benchmark application. This simulation program simulates an efficient unicast algorithm implemented on a hypercube. It takes 2–130 h to complete a simulation run depending on the dimension n. The simulator takes longer time to reach its equilibrium state as the dimension n grows. The memory requirement of this simulator is roughly of the size $c \times n \times 2^n$ bytes, where n is the dimension of the hypercube and the constant c is equivalent to 5×10^4. Table 4 illustrates the average memory sizes for n from 3 to 9. The checkpoint time depends on the memory allocated for the simulator at the time the checkpoint takes place. Fig. 11 shows that it takes in average 134 ms to 70.99 s to complete one checkpoint as the dimension n grows. Fig. 12 illustrates another aspect of checkpoint overhead of the same hypercube simulator, where *NT-SwiFT* carries periodic checkpointing. In this scenario, *Winckp* takes one checkpoint automatically at the beginning of each period. We compare the total execution time of the simulator with and without periodic checkpoint. The relative overhead of the periodic checkpoint is defined as the checkpoint time divided by the checkpoint interval. Fig. 12 illustrates the relative overhead of periodic

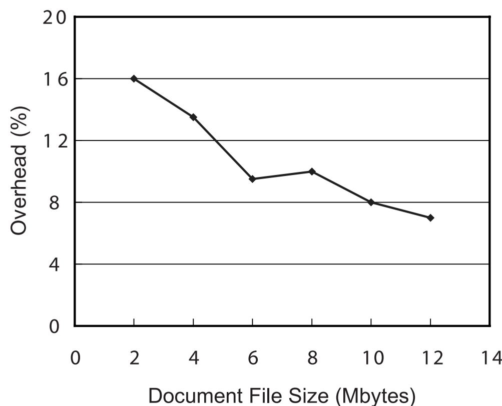

Fig. 10. Overhead of replay mode.

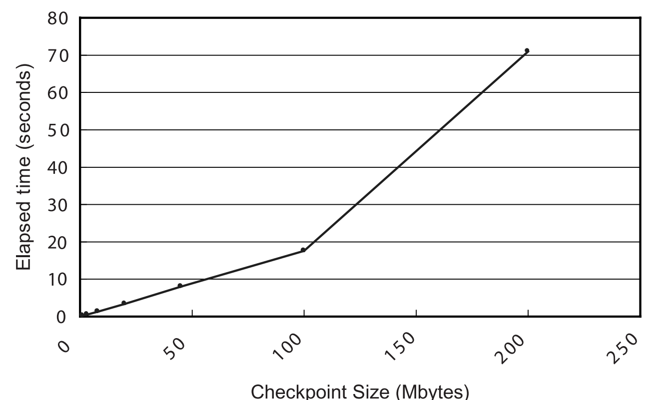

Fig. 11. The performance of *Winckp*.

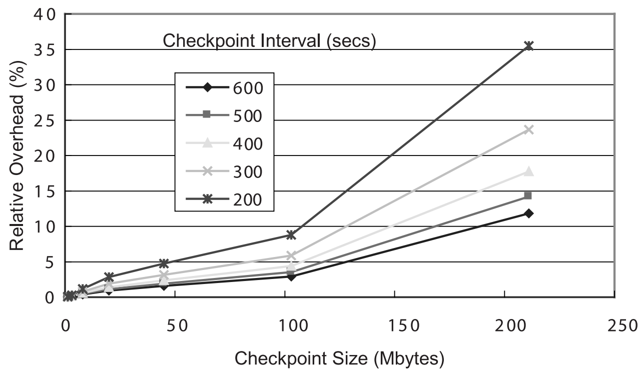

Fig. 12. The overhead of *Winckp*.

checkpoint as the checkpoint interval varies from 200 to 600 s.

5. Related work

A few commercial NT cluster products also provide some of the *NT-SwiFT* functions. A survey on NT clustering solutions can be found in Microsoft (1999). Examples of NT clustering solutions are Microsoft *MSCS*, Tandem CAS, Marathon Endurance, Apcon PowerSwitch, NCR LifeKeeper, Veritas FirstWatch, Octopus HA+, etc. These commercial NT cluster products provide basic *fail-over* and detection capabilities. Some of them also provide file replication or disk-mirroring facilities for persistent data recovery. However, there are at least three major differences between *NT-SwiFT* and these clustering tools:

1. The fundamental design philosophy is different between *NT-SwiFT* and these commercial tools. Most of these tools assume that application programs cannot be changed and therefore all the recovery mechanisms have to be completely transparent to application programs. Consequently, these clustering tools provide either no application APIs or a very small set of APIs to be embedded into application programs. Our design philosophy considers the recovery mechanisms into two categories: client recovery and server recovery. We also think that the client error recovery mechanisms have to be transparent to the client application programs. However, we believe that a truly fault-tolerant server application has to be enhanced with fault tolerance APIs. Therefore, a large part of our effort is to design and implement a set of fault tolerance APIs for the server application developers. [4] As a result, the APIs provided by *libft* are more powerful and complete than those provided by these commercial clustering tools. These fault tolerance APIs also have to interact with other components in *NT-SwiFT* such as *watchd*, *REPL*, and *Winckp*. Therefore, an integration of all fault tolerance components is essential but none of the commercial clustering tools provides such integration.

Most of the commercial clustering tools assume transaction model for error recovery while our focus is on the checkpoint and rollback recovery. As a result, none of these tools integrate rollback recovery mechanisms such as process checkpoint, events/messages logging and replay, etc. into their recovery mechanisms. Without an integrated solution, application developers may have to design and implement a lot of recovery routines into their programs.

A similar checkpointing facility was developed at Intel (Srouji et al., 1998). The main differences are (1) it requires a different build process to replace an application's startup function with their checkpoint DLL startup code; (2) it utilizes the *QueueUserAPC()* mechanism on NT to replay all system calls at recovery by application threads, while our system calls are replayed by the injected thread; (3) it does not roll back any persistent storage; (4) it does not handle GUI programs; (5) it does not log or replay window messages. There is also a checkpointing facility on the Brazos parallel programming environment (Abdel-Shafi et al., 1999). To support thread migration, Brazos currently implements a subset of *Winckp* functions including saving the thread context, stack and memory.

2. *NT-SwiFT* provides facilities to achieve application rejuvenation [5] (Garg et al., 1996), IP requests dispatching, process migration and load balancing. As a result, *NT-SwiFT* cannot only increase application availability but also improve application robustness, performance and scalability.

6. Concluding remarks

The goal of *NT-SwiFT* research is to study the fault-tolerance and high availability requirements of applications running on NT and to create generic and reusable components that can enhance the availability of these applications. We have described the following components: *watchd* for process failure detection and recovery, *libft* for critical data checkpointing, communication messages logging and recovery, *REPL* for on-line incremental file replication and disaster recovery, *Winckp* for transparent process checkpointing, and mouse and keyboard events logging and replaying. We have demonstrated that leveraging specific facilities on Windows NT such as filter drivers, intermediate drivers, library injection and memory management routines

[4] Note that except some functions in *libft*, all other components in *NT-SwiFT* can be used transparently with application programs.

[5] Application rejuvenation is a mechanism that monitors applications behaviors, predicts applications failures and rejuvenates unhealthy applications even before they actually fail.

makes the implementation of some fault tolerance mechanisms easier on Windows NT than on UNIX.

NT-SwiFT has been used in some Lucent telecommunication products to detect and recover from process or node failures. One of those products, now in Avaya Inc., is a small office, dual node private branch exchange (PBX) system running on Windows NT. Performance of *NT-SwiFT* in such real-time telecommunication systems is satisfactory.

Acknowledgements

The authors would also like to thank Gaurav Suri and Yi-Min Wang who implemented the first prototype of *watchd* and *libft* in *NT-SwiFT* and are grateful to Dave Korn for his help in using UWIN and comments on this paper.

References

Abdel-Shafi, H., et al., 1999. Efficient user-level thread migration and checkpointing on Windows NT clusters. In: Proceedings of the 3rd USENIX Windows NT Symposium, July, pp. 1–10.

Birman, K., 1996. Building Secure and Reliable Network Applications. Manning Publication Co.

Garg, S., Huang, Y., Trivedi, K., Kintala, C., 1996. Minimizing completion time of a program by checkpointing and rejuvenation, ACM SIGMETRICS 96, Philadelphia, PA, May, pp. 252–261.

Gray, J., Reuter, A., 1993. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers.

Huang, Y., Kintala, C., 1993. Software implemented fault tolerance. In: Proceedings of the 23rd IEEE Fault-tolerant Computing Symposium (FTCS23), Toulouse, France, June, pp. 2–10.

Huang, Y., Wang, Y., 1995. Why optimistic message logging has not been used in telecommunication. In: Proceedings of the 25th IEEE Fault-tolerant Computing Symposium (FTCS25), Pasadena, CA, pp. 459–463.

Korn, D., 1997. UWIN—UNIX for Windows. In: Proceedings of the Usenix Windows NT Workshop, Seattle, WA, August, pp. 133–145.

Liang, D., Fang, C.-L., Yuan, S.-M., Chen, C., Jan, G., 1999. A fault-tolerant object service. Journal of Systems and Software 48, 197–211.

Libertone, D., 1999. Windows NT Cluster Server Guidebook. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Lucnet Technologies, 1999. Software Implemented Fault Tolerance: SwiFT for Windows NT, White Paper, Lucent Technologies: Bell Labs. August 1999, or http://www.research.avayalabs.com/project/swift/detail/.

Microsoft Inc., 1999. White Paper: Windows NT Clustering Architecture.

Microsoft Inc., 2002. http://support.microsoft.com/support/kb/articles/Q257/9/82.ASP.

OMG, 1999. Fault Tolerant CORBA, OMG TC Document, orbos/99-12-08.

Richter, J., 1997a. Breaking through process boundary wall. In: Advanced Windows, 3rd ed. Microsoft Press, pp. 899–970 (Chapter 18).

Richter, J., 1997b. Processes. In: Advanced Windows, 3rd ed. Microsoft Press, pp. 33–72 (Chapter 3).

Richter, J., 1997c. Win32 memory architecture. In: Advanced Windows, 3rd ed. Microsoft Press, pp. 115–144 (Chapter 5).

Srouji, J., et al., 1998. A transparent checkpoint facility on NT. In: Proceedings of the 2nd USENIX NT Symposium, Seattle, WA, pp. 77–85.

Tsai, T.K., Singh, N., 2000. Reliability testing of applications on Windows NT. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), June 25–28, 2000, New York, USA, pp. 427–436.

Walli, S.R., 1997. OpenNT: UNIX application portability to Windows NT via an alternative environment subsystem. In: Proceedings of the Usenix Windows NT Workshop, Seattle, WA, pp. 123–132.

Wang, Y.-M., Huang, Y., Vo, K., Chung, E., Kintala, C., 1995. Checkpoint and its applications. In: Proceedings of the 25th IEEE Fault-tolerant Computing Symposium, Pasadena, CA, pp. 22–31.

Deron Liang received a BS degree in electrical engineering from National Taiwan University in 1983, and an MS and a Ph.D. in computer science from the University of Maryland at College Park in 1991 and 1992 respectively. He is a faculty of Computer Science Department, National Taiwan Ocean University, Taiwan since 2001. He also holds joint appointment with the Institute of Information Science (IIS), Academia Sinica, Taipei, Taiwan, Republic of China. He was with IIS from 1993 till 2001.

His current research interests are in the areas of software fault-tolerance, system security, and system reliability analysis. Dr. Liang is a member of ACM and IEEE.

Pi-Yu Emerald Chung is a member of the Core Engineering at Siebel Systems. She designs and develops data synchronization services for handheld and mobile devices. Her research interests include mobile network architecture, security, fault-tolerance computing and distributed transaction processing. She received a BS in electrical engineering from National Taiwan University and an MS and a Ph.D. in electrical engineering from the University of Illinois at Urbana-Champaign.

Chandra M.R. Kintala is Vice President of Research Realization Center in Avaya Labs in Basking Ridge, NJ. He received Ph.D. in Computer Science; and has five patents and over 40 research publications in software fault tolerance, programming environments and theoretical computer science and a Smithsonian medal sponsored by Computer-World for his pioneering contributions to software-implemented fault tolerance (SwiFT).

Previously, Chandra was Head of Distributed Software Research in Bell Labs in Murray Hill, NJ. He was also an Adjunct Professor of Computer Science at Stevens Institute of Technology. He edited two special issues of Bell Labs Technical Journal on Software.

Yenmm Huang is currently the VP of Engineering for PreCache Inc. in charge of R&D of the company. Before he joined the PreCache in 2001, he was the Division Manager of Dependable Distributed Computing Research Department in AT&T Labs. He received his B.S. degree from National Taiwan University in 1982, a M.S. degree and Ph.D. from University of Maryland at College Park in 1986 and 1989, respectively.

He has worked on many software tools and techniques to improve reliability and availability of software systems. He also helped many projects in reliable system designs and implementations. He worked for Bell Labs from 1989 to 1999. He was the only recipient of the Lucent Commemorating Stock Certificate in recognition for work in inventing and pioneering the development of software fault tolerance from Bell Labs Research in 1996. His SwiFT work won the Computerworld Smithsonian Awards in 1998.

He has 11 US patents and more than 30 publications. He has served in many program committees such as FTCS, DSN, ICDCS, SRDS, WWW, etc. His main research interests are fault-tolerant computing, distributed event processing, distributed object technologies, performance evaluation, distributed systems and cluster computing, etc.

Woei-Jyh Lee received his BS degree from the Department of Computer Science and Information Engineering at the National Taiwan

University in 1993, and his M.S. degree from the Department of Computer Science at the New York University in 1998. He worked at Bell Laboratories Research, Lucent Technologies, from 1998 till 2000. He is currently pursuing a Ph.D. degree in the Department of Computer Science at the University of Maryland at College Park. His research interests include systems simulation and performance evaluation, network policies and management, distributed systems, and Internet protocols.

Timothy K. Tsai is a member of the technical staff at Avaya Labs, Avaya Inc. His research interests include computer security, fault-tolerant system design and validation, software engineering, and distributed systems. He received a B.S. in electrical engineering from Brigham Young University and an M.S. and a Ph.D. in electrical engineering from the University of Illinois at Urbana-Champaign. He is a member of IEEE and IEEE Computer Society.

Chung-Yih Wang is currently the Senior Research Engineer for Pre-Cache Inc. in charge of distributed information service. He received his B.S. degree from National Taiwan University in 1993, a M.S. degree from National Tsing-Hwa University 1995. His research interests include distributed systems, software reliability, and data networking.