The following paper was originally published in the
Proceedings of the 2nd USENIX Windows NT Symposium
Seattle, Washington, August 3–4, 1998

NT-SwiFT: Software Implemented Fault Tolerance on Windows NT

Yennun Huang, P. Emerald Chung, and Chandra Kintala
Bell Labs, Lucent Technologies
Chung-Yih Wang and De-Ron Liang
Institute of Information Science, Academia Sinica

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

NT-SwiFT: Software Implemented Fault Tolerance on Windows NT

Yennun Huang
P. Emerald Chung
Chandra Kintala

Bell Laboratories,
Lucent Technologies, Inc.
600 Mountain Avenue
Murray Hill, NJ 07974

Chung-Yih Wang[1]
De-Ron Liang[1]

Institute of Information Science
Academia Sinica
Taipei, Taiwan
R.O.C.

Abstract

More and more high available applications are implemented on Windows NT. However, the current version of Windows NT (NT4) does not provide some facilities that are needed to implement these fault tolerant applications. In this paper, we describe a set of components collectively named NT-SwiFT (Software Implemented Fault Tolerance) which facilitates building fault-tolerant and highly available applications on Windows NT. NT-SwiFT provides components for automatic error detection and recovery, checkpointing, event logging and replay, communication error recovery, incremental data replications, IP packets re-routing, etc. SwiFT components were originally designed on UNIX. The UNIX version was first ported to NT to run on UWIN [Korn97]. Gradually a large portion of the software has been re-implemented to take advantage of native NT system services. This paper describes these components and compares the differences in the UNIX and NT implementations. We also describe some applications using these components and discuss how to leverage NT system services and cope with some missing features.

1. Introduction

Windows NT has become a popular and viable computing platform for critical applications due to its many useful features and low hardware costs. The telecommunication industry has also started to build fault-tolerant and highly available applications on NT. To achieve high reliability and availability in a distributed environment, three types of techniques have been deployed, namely, transaction processing [Gray93], active replication [Birman96], and checkpointing/message logging [Huang93]. Transaction processing is popular in the financial industry. In a transactional system, applications usually have a well-defined transaction boundary, such as updating a record. When a fault occurs, both client and server abort the on-going transaction and rollback to a clean state. Active replication usually involves several identical servers running synchronously. It often assumes a deterministic behavior on these servers and requires an atomic broadcast mechanism to synchronize messages. When a failure occurs in one server, the failure is masked and the computation continues as long as there is one server running. No rollbacks are necessary on either client or server.

Checkpointing and message logging is another way to provide fault tolerant services. 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 and its state is recovered by loading its last checkpoint and replaying its logged messages. Client may notice some delay during a recovery, but no rollback is involved. Many telecommunication applications constantly manage or monitor some physical devices. Our experience shows that checkpointing and message logging is most suitable for this type of applications [Huang95]. To implement checkpoint and messages logging, we need a number of facilities not provided by Windows NT 4.0. They are application monitoring and failure recovery, application checkpoint and message logging, file replication, Windows events log-

[1]This work is sponsored by Lucent Technologies, Inc.

ging and replay, IP packets dispatching, and IP packets re-routing in case of a machine failure. As a result, each application has to implement its own recovery mechanisms. These recovery mechanisms are usually very complex and hence may not be easy to design and implement by application developers. Therefore, it is desirable to provide them as reusable software components.

In Bell laboratories, we have been working on a set of reusable modules for building reliable and fault tolerant applications for the last 6 years. The set of modules is called SwiFT (Software Implemented Fault Tolerance) [Huang93]. SwiFT has been embedded into tens of telecommunication systems to improve system availability and has been licensed to companies such as Tandem Co., etc. 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 but 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 Bell Labs often have access to the source code of server programs but have no control of client programs developed by other companies.

SwiFT was first implemented and applied on UNIX systems (UNIX-SwiFT). More than two years ago, we started porting SwiFT fault tolerance mechanisms to Windows NT (NT-SwiFT). At the beginning, we were not sure if NT provides enough mechanisms and utilities for us to implement all fault tolerance utilities we need. However, after more than two years of NT-SwiFT effort, we concluded that Windows NT does have all the facilities that are needed to implement SwiFT on Windows NT although some of the NT-SwiFT implementations are quite complex. In this paper, we describe the NT-SwiFT components, their implementation issues and some examples of using NT-SwiFT to enhance applications' reliability and availability. The paper is organized as follows. Section 2 describes NT-SwiFT components. Section 3 discusses some implementation details and issues. Section 4 shows some examples of using NT-SwiFT and performance measurements. Section 5 compares NT-SwiFT with related work. Section 6 concludes the paper.

2. NT-SwiFT Components

As described earlier, NT-SwiFT components can be used in both client and server error recovery. Therefore, we describe NT-SwiFT components in two categories - client components and server components. Please note that since a program could be both a client and a server, all these components can be applied in a program.

2.1 Components for client error recovery

The design philosophy of client-side recovery components is to make them transparent to client programs. That is, one can embed NT-SwiFT components into client programs without modifying the client source code. 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 are 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 brings the client program back to the state just before the failure occurred.

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* which 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. The third component is *winrecord,* which 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. In recovery, *libft* re-establishes communication endpoints using the checkpointed information and, if necessary, replays the logged messages.

2.2 Components for server error recovery

On the server side, *watchd* can also be used to detect and recover a server program from a failure. A fault tolerant application process can register its replication

strategy to *watchd*. There are two replication strategies that *watchd* supports: hot, and cold. In the hot replication case, *watchd* monitors all replicas of a fault tolerant process; if any replica failure is detected, *watchd* recovers the failed replica on another machine so that the number of replicas (degree of fault tolerance) remains constant. In the cold replication scheme, *watchd* assumes that there is only one active copy of a fault tolerant process; if the active copy fails, *watchd* will first try to recover the failed process on its local machine; if the local recovery fails, *watchd* then migrates the process onto another machine (a fail-over). *Watchd* also provides a few distributed system services such as remote execution, remote file copy, remote status query, etc. Many of these services can be invoked by an application using *libft* APIs. *Watchd* detects two kinds of server failures - hang or crash. To detect a server hang, the server process needs to periodically sends its heartbeats to *watchd*. A server process is considered hung if *watchd* does not receive a heartbeat from the server within a given interval. To send heartbeats to *watchd*, an application can call the *hbeat()* function in *libft* which takes a thread *id* and a timeout value as arguments. To detect a server crash, *watchd* pings the server process periodically. For a hang recovery, *watchd* kills and restarts the hung server process. For a crash recovery, *watchd* first determines the cause of the crash. It can be a machine (including OS failure) or an application program failure. To handle a machine, *watchd* does a fail-over for the server application by either bringing up a cold copy of the server application on another machine or making a warm copy of the server application active. To handle an application program crash failure, *watchd* simply restarts the application. *Watchd* contains a GUI for system configuration as shown in figure [watchd].

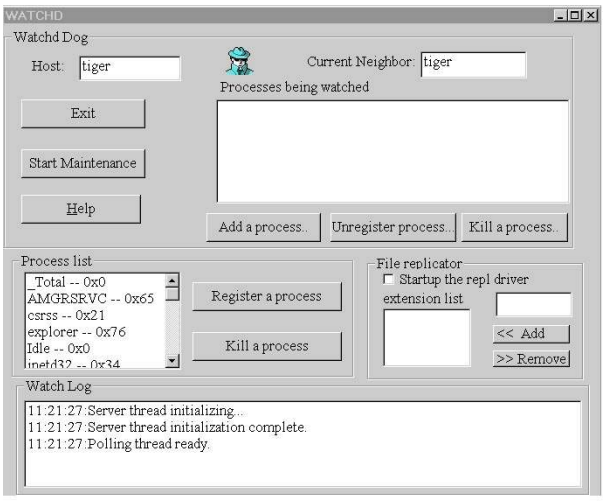

Figure [watchd]: *watchd* GUI

Libft has four major sets of functions for servers:

1 Critical data checkpointing: *Libft* allows an application to select critical data from data segment (e.g. global or static variables) and heap (e.g. data allocated via *malloc()*). 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. In case of a process crash, the data can be restored into the memory of a newly created process.

2 Communication channel recovery: *Libft* can intercept *winsock* system calls, such as *accept(), listen(), send(), recv()* so that *winsock* communication endpoints and messages can be logged. *Libft* recreates communication endpoints and replays logged messages during a recovery.

3 Requesting services from *Watchd*: *Libft* provides functions for system configurations such as registering a host or a process to be monitored by w*atchd*. In addition, it allows an application to send heart-beats to *watchd* and to invoke distributed system services such as remote file copy and status query from w*atchd*.

4 Intercepting kernel calls for system handles and file updates: *Libft* can be used to intercept calls that create system handles and that change file contents or attributes. This interception is needed in *winckp* for transparent checkpointing and roll-back recovery [Wang95].

A server program may also create and update files during execution. To make a fail-over possible in a share-nothing environment, component *REPL* can be used to do selective and incremental file replication. By using the *watchd* GUI, a user can specify the types of files that he/she wants to be replicated (see figure [watchd]). For example, a user enters *ppt* in the "*File Replicator*" sub-window in the *watchd* GUI and *REPL* replicates all *powerpoint* files of a machine onto one or more backup machines. *REPL* is typically used in a fail-over environment where files could become unavailable when a machine crashes. *REPL* has also been used for disaster recovery for UNIX applications where a backup machine/disk is located in very far away site.

In a cluster environment, NT-SwiFT provides *ONE-IP* driver (*oneip.sys*) to dispatch and fail-over IP packets. The *ONE-IP* driver provides a single IP image for a cluster of machines. This *ONE-IP* mechanism transpar-

ently distributes TCP/IP requests to a set of server machines in a cluster for load balancing and failure recovery [Damani97]. The *ONE-IP* driver can be installed on a set of server machines. A distributed election protocol is used to select one machine as the dispatcher. All server machines in the cluster share the same cluster IP address. In a typical configuration, they (including the dispatcher) run the same applications such as a *web* server, database servers and internet service daemons to provide services. Client applications use the cluster IP address to access a server for services. To achieve load balancing and fault tolerance, the dispatcher picks up the client requests and forwards them to one of the server machines for a service. If the dispatcher fails, *watchd* detects the failure and promotes another machine to become a dispatcher.

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 [Korn97], 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*, *ONE-IP*, *libckp* and *libft* depend on the UNIX internals. They can not be ported directly by using a library mechanism such as UWIN [Korn97] or a subsystem such as OpenNT [Walli97].
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 application fails, we need to have intimate knowledge of the NT architecture. Re-implementing SwiFT on NT using native NT system services help 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 runs on every machine in a network and uses an adaptive diagnosis protocol [HUANG93] to detect machine failures, i.e., each *watchd* pings its neighbor *watchd*; if its neighbor fails, *watchd* pings its next neighbor and so on. 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 a hang detection. Threads are synchronized using semaphores and critical sections. 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.

Watchd uses *OpenProcess()* and *WaitForMultipleObjects()* to detect a 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 *hbeat()* functions in *libft*.

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 [Huang93]. 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 [Richter97-18]. 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 mes-

sages 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 and three daemon processes for sending messages and replaying 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. On UNIX, we use the dynamic shared library mechanism (*dlopen()* and *dlsym()*) to intercept and replay file system calls. On NT, we implement a filter driver, named *REPL.sys*, to intercept file system calls. When a specified type of file is changed, REPL driver intercepts the changes and sends messages to remote backup machines. REPL daemons on remote backup machines then 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

Winckp is a utility program that provides snapshot and rollback functions to an application in a transparent way. Winckp deals with executable files and no source code is needed. *Winckp* starts the application by *CreateProcess()* and obtains its process handle and the thread handle of its main thread. The GUI interface of *Winckp* allows a user to take snapshot of an application or roll back the memory of an application. To take a snapshot, *Winckp* suspends the main thread and stores the thread context and memory content into a checkpoint file. The thread context is obtained and restored using *GetThreadContext()* and *SetThreadContext()*. The memory image is obtained and restored using *ReadProcessMemory()* and *WriteProcessMemory()*. *Winckp* determines the address and the amount of memory needed to be saved. An NT process has about 2GB of private address space, ranging from 0x00010000 through 0x7FFEFFFF [Richter97-3]. Note that not every region in this space needs to be saved. The *VitualQueryEx()* system call allow us to examine the space region by region. A memory region is necessary to be saved if its write access is enabled and if its physical storage is committed [Richter97-5]. *Winckp* also stores the MEMORY_BASIC_INFORMATION structure along with each memory region. During a rollback operation, the application main thread is suspended. *Winckp* reads the thread context from the checkpoint file and calls *SetThreadContext()*. It then calls *WriteProcessMemory()* to restore the memory content.

To recover an application process from a failure, *winckp* not only has to restore the process memory content but also has to recreate all the handles that were owned by the process before the recovery. To checkpoint and recover kernel states, *winckp* uses the *libft* interception facilities to intercept all NT calls which create file handles, process handles, thread handles, socket handles, such as *CreateProcess()*, *CreateFile()*, *CreateThread()*, etc. *Winckp* records each handle value and the parameters that are used to create the handle. In recovery, *winckp* recreates all the handles by replaying the calls with the recorded parameters. To make the values of the recovered handles equal to their recorded values, *winckp* uses a different mechanism from the UNIX version (namely *libckp* [Wang95]). In *libckp*, each newly created handle is duplicated to its old value by using the *dup2()* call. In NT, since there is no function that can duplicate a handle to a given handle value, *winckp* uses a loop that keeps duplicating a handle using D*uplicateHandle()* call till the returned handle value is equal to the recorded value. Then, all other handles are closed. This process is repeated till all the handles are created.[2]

Winckp also uses *libft* to intercept file system calls. When an application takes a checkpoint, it has not only to save its memory contents but also its file contents and attributes. 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. The file roll-back mechanism uses the *libft* interception routines as described in [Wang95].

3.5 Winrecord

In *winrecord*, 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. *Winrecord* captures those events by calling *SetWindowsHookEx()* with flag WH_JOURNALRECORD, and all keyboard

[2] This mechanism does not work for Windows handles.

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 uninstalled.

3.6 *ONE-IP* driver

The *ONE-IP* driver is an NDIS (Network Driver Interface Specification) intermediate driver. It is sitting between transport drivers and NDIS NIC (Network Interface Card) mini-ports. The driver is installed on every machine in the cluster. Our design works in the following way: all the client request packets are first sent to the dispatcher machine and the dispatcher machine selects a server from the cluster and forwards the packet to that server. A problem is that all machines share the cluster IP address. In order for a packet to reach the dispatcher, only the dispatcher should reply ARP requests for the cluster IP. In our implementations, when the immediate driver on a server machine receives an ARP request packet for the cluster IP address, if it is not the dispatcher, it discards the ARP packet.

On the dispatcher machine, when the NIC driver receives a packet, it calls the *ReceiveHandler()* in the transport interface of the *ONE-IP* intermediate driver. The *ReceiveHandler()* examines the packet. If the packet is from a client request, it contains an Ethernet packet header and an IP packet in the lookahead buffer. If the destination address of the IP packet matches the cluster IP address, a server is selected based on a hash value of the client IP address (source address in the IP packet). The Ethernet packet header is then modified in the following way: the source MAC address is changed to the dispatcher's MAC address; the destination MAC address is changed to the selected server's MAC address. The packet is then sent to the NIC driver by *NdisSend()* call and reaches the selected server. Since a dispatcher can also be servicing requests, if the dispatcher itself is selected, then the packet is passed up to the protocol driver without modifications.

One desired feature for the *ONE-IP* driver is the capability to dynamically reconfigure the dispatching hash function, the cluster IP address or the cluster size. To achieve this, we create a logical device in the *ONE-IP* driver by *IoCreateDevice()* and expose a device name in the NT object namespace, *\\device\oneip*. A user-level program can change parameters in the *ONE-IP* driver by first obtaining a handle to the logical device by *CreateFile()* with the device name and then issuing a *DeviceIoControl()* via the handle.

To make the *ONE-IP* dispatching mechanism fault tolerant, we integrate *ONE-IP* driver with the *watchd* daemon. As mentioned earlier, *watchd* runs on every machine in a SwiFT domain. When the first *watchd* comes up in the domain, it makes its own *ONE-IP* driver the primary dispatcher by calling the *DeviceIoControl (sys_handle, SET_PRIMARY,…)*. When the dispatcher machine fails, the neighboring *watchd* detects the failure and set its *ONE-IP* driver the new primary dispatcher.

The UNIX implementation of *ONE-IP* is done in the NetBSD kernel [Damani97] [Wang 97]. The dispatcher runs our modified kernel and is configured to run in the routing mode. The main kernel modifications are in the IP forwarding layer ([Wright 95] p.222). We modified the `ip_forward()` routine so that the selected server's IP address is used as the next hop for the packet.

Since the UNIX implementation involves changing kernel code, it is difficult to port to a system where the kernel source code is unavailable. On the other hand, the NDIS driver approach on NT is much easier to be adopted into a product.

4. Applications and Overhead

We are currently working with a few projects in Lucent Technologies to embed NT-SwiFT in their systems to improve their fault tolerance and availability. In one project, the system uses NT-SwiFT to detect application failures such as process crashes and hangs. Once a failure is detected, *watchd* stops the process and restart the process. If a process fails too many times in a given interval, *watchd* then automatically reboots the NT machine and restarts the application. In another project, we are using *watchd* and *libft* to provide a warm backup scheme for a switch prototype implemented on Windows NT where processes on the primary board checkpoint their critical states to the backup processes on a backup board whenever necessary. When a failure is detected, *watchd* makes the backup board the primary by changing a flag in the shared memory of the board.

In a normal situation, *watchd* polls applications and machines every 10 seconds and one polling takes about 10 milliseconds on a Pentium 180MHz machine. By polling, *watchd* increases the CPU utilization by about 4%. *Libft* overhead depends on the frequency of checkpointing and message logging. In one study, it showed 5 to 10% increases for the service time of a server program when checkpoint and message logging were used. REPL overhead also depends on the intensity of I/O write operations. One study showed 14% decrease of I/O throughputs when using REPL in replicating files for a disaster recovery.

5. Comparison with Related Work

Some of the NT-SwiFT functions are also provided by a few commercial NT cluster mechanisms. A survey on NT clustering solutions can be found in [NT-CLUSTER]. 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 application programs can not 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[3]. 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 SwiFT such as *watchd*, *REPL*, *winrecord*, *winckp* and *ONE-IP*. Therefore, an integration of all fault tolerance components is a must but none of the commercial clustering tools provides such integration.
2. Most of these clustering tools assume transaction model for error recovery while our focus is on the checkpoint and roll-back recovery. As a result, none of these tools integrate roll-back 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.
3. NT-SwiFT provides facilities to do application rejuvenation [Garg96][4], IP requests dispatching, process migration and load balancing. As a result, NT-SwiFT can not only increase application availability but also improve application robustness, performance and scalability.

6. Concluding Remarks and Future work

The goal of NT-SwiFT research is to understand the fault-tolerance and high availability requirements of applications running on NT and to create generic and reusable components that can facilitate the development of these applications. We have described components including *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, *winrecord* for mouse and keyboard events logging and replaying, and *ONE-IP* for IP packets dispatching, fail-over and re-routing. We have demonstrated that leveraging specific facilities on Windows NT such as filter drivers, intermediate drivers, library injection and memory management routines makes the implementation of some fault tolerance mechanisms easier on Windows NT than on UNIX.

Currently, we are working on enhancing the NT-SwiFT to deal with process thread failure detection and recovery, incremental state checkpoint to remote processes, integration of the NT-SwiFT with some middle-ware

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

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

tools such as CORBA and DCOM, dynamic process migration for load balancing, intercepting calls in other DLLs such as *advapi32.dll, user32.dll, GDI32.dll*, etc., and the compatibility of NT-SwiFT with other popular commercial clustering tools such as MSCS.

Acknowledgements: Gaurav Suri and Yi-Min Wang implemented the first prototype of *watchd* and *libft* in NT-SwiFT. Recently, Woei-Jyh Lee joined our NT-SwiFT team and contributed in the porting of *ONE-IP* driver and *watchd*. The authors would also like to thank the users of the NT-SwiFT who constantly provide ideas for improvements and Dave Korn for his help in using UWIN and comments on this paper.

References:

[Birman96] Kenneth P. Birman, "Building Secure and Reliable Network Applications", Manning Publication Co. 1996.

[Damani97] Damani, O. P., Chung, P.-Y., Huang, Y., Kintala, C. M., and Wang, Y-M., "*ONE-IP*: Techniques for Hosting a Service on a Cluster of Machines", *Sixth International World Wide Web Conference (WWW6)*, Santa Clara, pp. 735-743, Apr. 1997.

[DDK-NDIS] "Network Drivers", Windows NT 4.0 DDK, Microsoft MSDN Library.

[Huang93] Huang, Y. and Kintala, C. "Software Implemented Fault Tolerance", *Proceedings of the 23rd IEEE Fault Tolerant Computing Symposium (FTCS23),* Toulous, France, June 1993, Pages 2-10.

[Huang95] Y. Huang and Y. Wang, ‘‘Why optimistic message logging has not been used in telecommunication", *Proceedings of the 25th IEEE Fault Tolerant Computing Symposium (FTCS25), Pasadena*, California, page 459-463, 1995.

[Garg96] S. Garg and Y. Huang and K. Trivedi and C. Kintala, ‘‘Minimizing Completion Time of a Program by Checkpointing and Rejuvenation",ACM SIGMETRICS 96, Philadelphia, PA, pages 252-261, May, 1996.

[Gray93] J. Gray and A. Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann Publishers, 1993.

[Richter96-3] J. Richter, "Processes", Chapter 3, in *Advanced Windows,* Ed. 3, pp.33-72, Microsoft Press, 1996.

[Korn97] D. Korn, “UWIN – UNIX for Windows”, *Proceedings of Usenix Windows NT Workshop*, Seattle, Washington, pp. 133-145, 1997.

[NTCLUSTER] "Lab Reports: Clustering Solutions for Windows NT", Windows NT Magazine, pp.54-95, June 1997.

[Richter97-5] J. Richter, "Win32 Memory Architecture", Chapter 5, in *Advanced Windows*, Ed. 3, pp.115 - 144, Microsoft Press, 1997.

[Richter97-18] J. Richter, "Breaking Through Process Boundary Wall", Chapter 18, in *Advanced Windows*, Ed. 3, pp.899-970, Microsoft Press, 1997.

[Walli97] S. R. Walli, “OpenNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem”, *Proceedings of Usenix Windows NT Workshop*, Seattle, Washington, pp. 123-132, 1997.

[Wang95] Y. Wang and Y. Huang and K. Vo and E. Chung and C. Kintala, “Checkpoint and its applications”, *Proceedings of the 25th IEEE Fault Tolerant Computing Symposium,* Pasadena, California, pp. 22-31, 1995.

[Wang97] Y.-M. Wang, O. P. Damani, P. E. Chung, Y. Huang and C. M. Kintala, Web Server Clustering with Single-IP Image: Design and Implementation", in *Proc. Int. Symp. on Multimedia Information Processing*, Dec. 1997, also in http://www.research.att.com/~ymwang/papers/newONE-IP.htm

WINDOWS SOCKET (Wsock32.dll)	accept, bind, closesocket, connect, ioctlsocket, listen, setsockopt, shutdown, socket, send, recv, recvfrom, sendto
File Operations (Kernel32.dll)	CopyFileA, CopyFileExA, CopyFileExW, CopyFileW, CreateFileA, CreateFileW, 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

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