

Feature article submission for Windows NT Magazine

Title: How Reliable Is Your NT Application? : Using Fault Injection to Test Critical Applications

Authors:

Timothy K. Tsai
600 Mountain Ave., Rm. 2B-442
Murray Hill, NJ 07974
Tel: (908) 582-4763

Navjot Singh
600 Mountain Ave., Rm. 2B-413
Murray Hill, NJ 07974
Tel: (908) 582-4106

Author biographies:

Timothy Tsai is a member of the technical staff of Bell Labs, Lucent Technologies. You can reach him at ttsai@research.bell-labs.com.

Navjot Singh is a member of the technical staff of Bell Labs, Lucent Technologies. You can reach him at singh@research.bell-labs.com.

Executive Summary:

Microsoft Windows NT is being used increasingly as a platform for server applications that demand high reliability and availability. However, although the ability to accurately characterize the reliability of these server systems is important, few tools exist to test these systems. Consequently, reliability and availability are described in non-precise terms: with gross estimates (e.g., a system has three "nines" of availability) or in term of specific scenarios that can be handled. Not only are these descriptions non-precise, they also do not take into account the specific applications that particular customers are interested in running. The ntDTS fault injection tool enables testing of specific NT applications along with any desired fault tolerance software and produces results that quantify the failure coverage of the complete system. This article describes the ntDTS tool and presents experiments that demonstrate the utility of the tool in testing and comparing the reliability of several well-known applications and fault tolerance software.

Ask a company that sells NT server systems just how reliable their system is, and you will probably hear a sales pitch about redundant fans, RAID, hot-swappable components, and other reliable design features. There might even be a mention of the number of nines of availability or a recounting of the many satisfied customers with continuous uptime. Still, the nagging question is how well a system will operate with your particular applications and workloads. The knowledge that the system design is sound and that other users have had good experiences only partially addresses the question. The most direct answer lies in explicit testing of your system, with the operating system, application, and fault-tolerant support that you will be using. Suppose you will be running a web server using Microsoft Internet Information Server (IIS) version 3 on Windows NT4.0 with SP4 along with Microsoft Cluster Server. That's what you should be testing to gain confidence that your system is reliable.

But, how do you go about performing reliability testing? Functional testing (does the application produce the right answer) and performance testing (how fast is my application) are easy enough. Just run the program and see how fast the correct answer appears. In contrast, reliability testing requires *fault injection*. Fault injection is the insertion of faults or at least the effects of faults into the system under test. Models of hardware or software problems that might occur are the basis of such faults. An example of a hardware fault is an electrical short in a memory chip that causes a "stuck-at-0" memory bit. A software fault might be a coding mistake that inadvertently omits the initialization statement for a variable. In either case, the fault causes corruption of the system state. This means that the operating system or application starts to behave improperly and may eventually crash or hang the machine or application (like a blue screen). Perhaps the application may silently misbehave and produce the wrong answer (like a database that commits the wrong value).

A reliable system contains measures to tolerate faults. Reliability testing of such a system involves execution of the system in the presence of faults to examine the ability of the system to produce correct results despite the influence of faults. This is basically what fault injection is. Run your program, inject a fault, and see if the program still spits out the right answer. However, there are a few complications. First, the faults of interest are usually not trivial to inject. Imagine trying to cause a short in a memory chip just to test its effect on your application. Not only would that permanently damage the chip, but also just how would you even go about causing such a short? In addition, fault injection must be coordinated with the program execution. It doesn't make much sense to inject a fault after the program has already finished producing its results.

Software-implemented fault injection greatly simplifies the use of fault injection. Software-implemented fault injection is the use of software to inject fault effects that emulate the underlying fault. For example, instead of directly causing a short in a memory chip, the software-implemented approach uses an injection program to zero out the corresponding bit in memory. The fault effect is the same, but no damage to the chip hardware occurs. In addition, reconfiguration of the injection program to inject different memory locations with different fault effects is simple. Another benefit is the ability to synchronize the injection program with the original tested application. This allows the injection program to inject the fault at a particular point in the application.

The ntDTS Tool

The NT Dependability Test Suite (ntDTS) fault injection tool is a freely downloadable package from Lucent Bell Labs that facilitates the fault injection testing process for Windows NT systems. See <http://www.bell-labs.com/projects/swift> for instructions on downloading the ntDTS package. By using the ntDTS tool, you will be able to test the reliability of your critical applications yourself.

There are two main reasons to use ntDTS:

1. Improve the reliability of your application. The tool will discover specific failure scenarios that are repeatable. This allows you to go back and improve your code to handle those scenarios. Note that you can make improvements to the application itself or in the operating system or fault tolerance software.
2. Compare the reliability of different systems. Suppose you are deliberating the merits of two different fault tolerance packages. The ntDTS tool produces quantitative reliability results to aid in the decision-making process.

The ntDTS package consists of two main parts:

1. A set of programs and scripts to instrument the test application for fault injection
2. The actual fault injection tool (Figure 1) that runs the application, injects faults, and collects the results.

We chose to develop the ntDTS tool in Java. The use of Java increases the portability of the tool (the subsequent port to Linux required substantially less time). The object-oriented nature of Java allows extensible objects to encapsulate the basic fault injection tool functionality. Consequently, configuration of the tool for different applications only requires the writing of a minimal amount of Java code.

Actually, in many cases the user needs to write no additional Java code because parameter files contain most of the needed configuration information. These parameter files describe the types of desired faults and applications.

So, what types of results does the tool produce? To understand the output, we need to understand the fault injection mechanism.

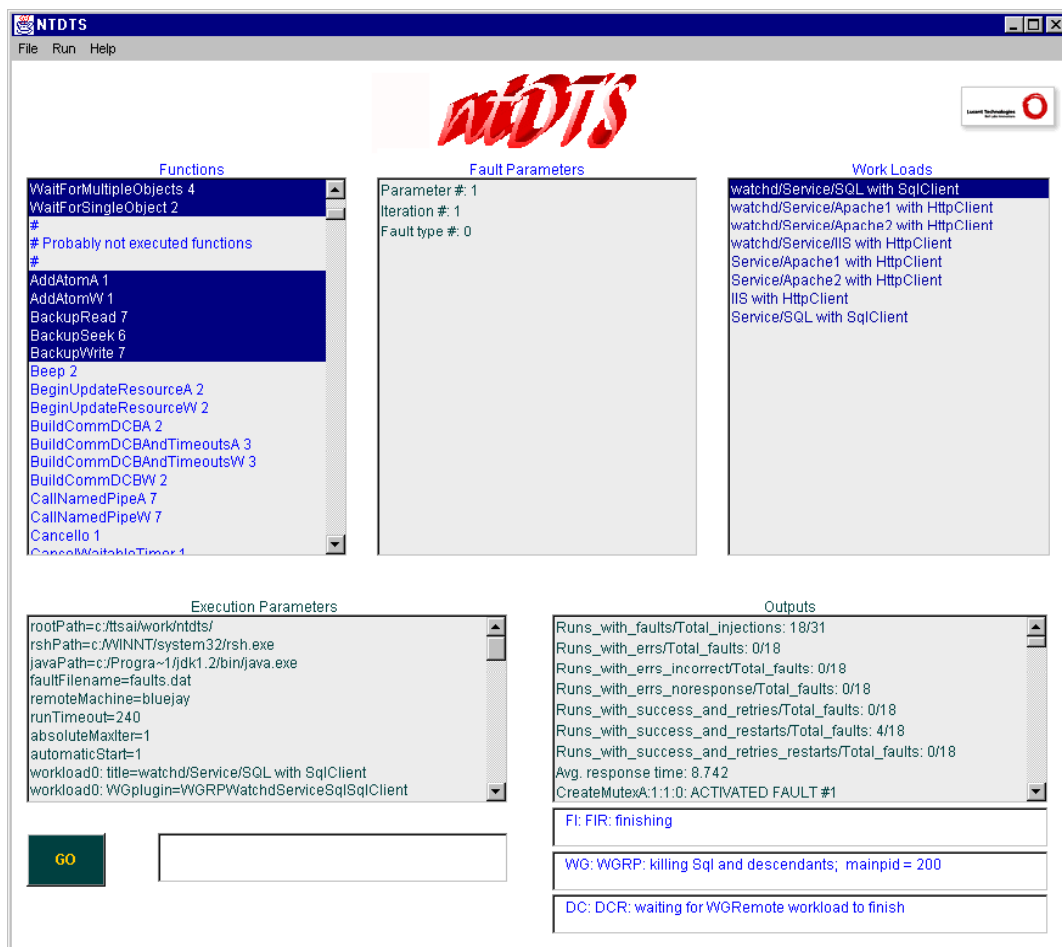


Figure 1 ntDTS Graphical User Interface

Fault Injection

The ntDTS tool injects faults by corrupting system call parameters. Suppose we want to inject the CreatePipe() function. The function prototype for CreatePipe() is

```

BOOL CreatePipe(
    PHANDLE hReadPipe           // pointer to read handle
    PHANDLE hWritePipe        // pointer to write handle
    LPSECURITY_ATTRIBUTES lpPipeAttributes // pointer to security attributes
    DWORD nSize               // pipe size
);

```

The tool corrupts one parameter at a time. For example, to corrupt the *hReadPipe* parameter, the tool does one of the following:

1. Set *hReadPipe* to 0x0.
2. Set *hReadPipe* to 0xffffffff.
3. Flip all the bits in *hReadPipe*. If the original value is 0x3, then the corrupted value is 0xffffffffc.

The tool then calls the original function with the corrupted parameter. This type of fault injection causes fault effects such as corrupting kernel data or passing back a bad return value. For Windows NT systems, functions in system DLLs (Dynamically Linked Libraries) invoke most system calls. The tool injects faults into the DLL functions, which then affect the associated system calls.

To automate the fault injection process, we need to do the following:

1. Create a wrapper DLL. The wrapper DLL contains masquerade functions for each of the functions in the original DLL.
2. Alter the target application Import Address Table (IAT) entries to point to the wrapper DLL. The IAT is a table that lists the DLL where each function is located.
3. Create configuration files to activate the wrapper DLL.

Figure 2 illustrates the roles of the wrapper DLL and the IAT in the fault injection mechanism.

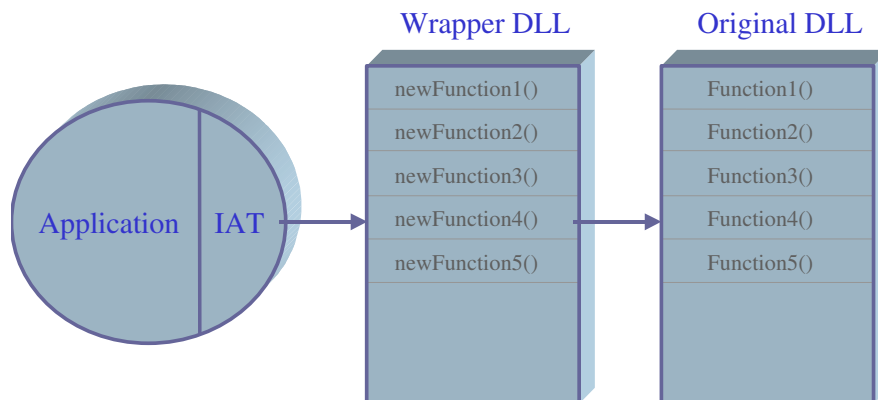


Figure 2 Fault Injection Mechanism

The above steps prepare the application for fault injection. Figure 3 shows what happens during the actual fault injection. The rectangular boxes in Figure 3 represent executable code. *IIS* is the IIS application code, which links at run-time to the *Wrapper DLL*, *KERNEL32.dll*, and *NTDLL.dll*. *HttpClient* is a small program that sends HTTP requests to the IIS web server. *Kernel Mode Components* include the

NT executive, kernel, and device drivers. *Parameter File* and *Output File* contain the contents for an actual fault injection. The following describes the sequence of events for a sample fault injection. The step numbers correspond to the numbers in Figure 3:

- 1) *IIS* starts by linking with the *Wrapper DLL* and other DLLs. The linking causes the initialization routine in the *Wrapper DLL* to read the fault injection *Parameter File*.
- 2) The *HttpClient* program starts and sends a request to *IIS*.
- 3) *IIS* processes the HTTP request and calls the *CreatePipe()* function.
- 4) The modified *IIS* IAT calls the *Wrapper DLL* function corresponding to *CreatePipe()*. The *Wrapper DLL* corrupts one of the parameters for *CreatePipe()*.
- 5) The *Wrapper DLL* creates an output file containing the details of the injected fault.
- 6) The actual *CreatePipe()* function in *KERNEL32.dll* is called using the corrupted parameter.
- 7) The *CreatePipe()* function eventually calls functions in *NTDLL.dll*, which access the resources in the *Kernel Mode Components*.

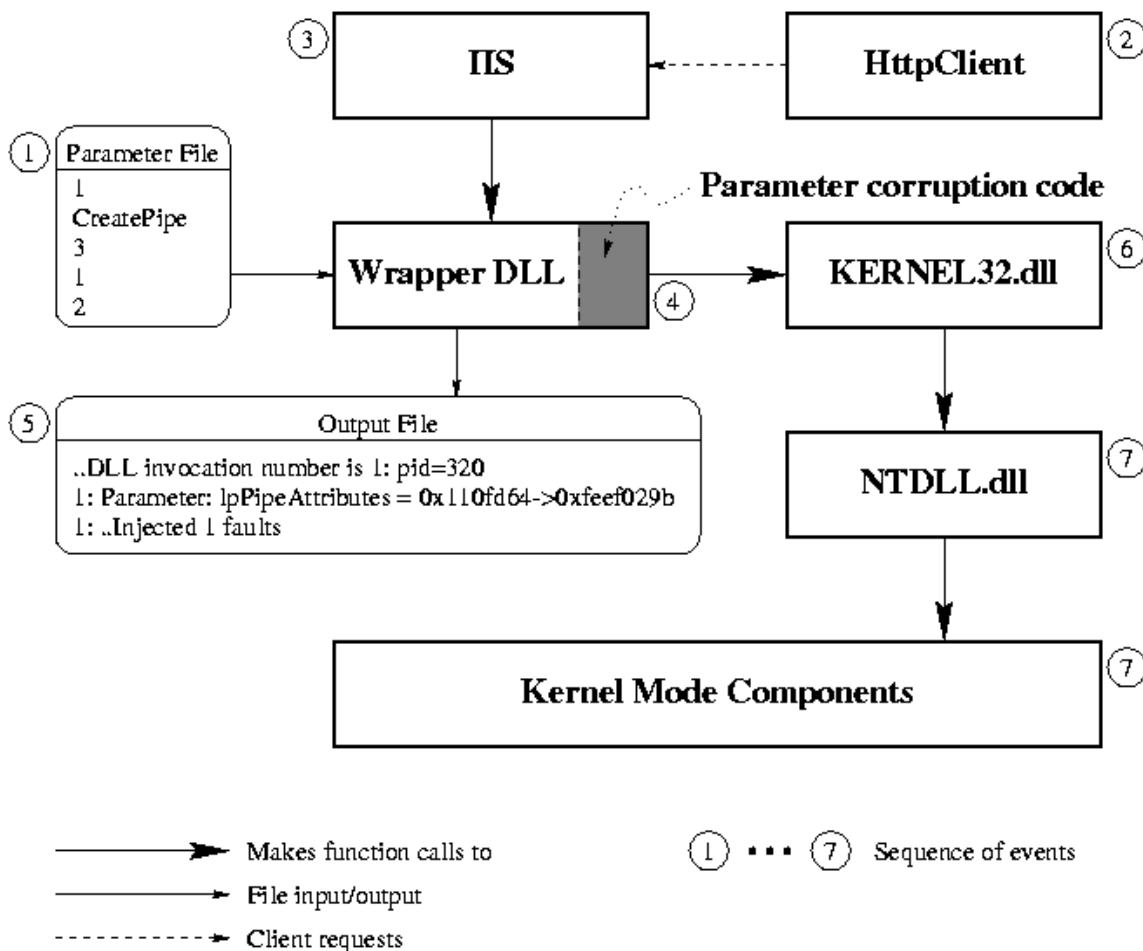


Figure 3 A Single Fault Injection run

Some Experimental Results

We tested the following applications using ntDTS:

1. Microsoft Internet Information Server (IIS) version 3.0.
2. Apache web server version 1.3.3 for win32.
3. Microsoft SQL server version 7.

We executed these applications in three different configurations: (1) as stand-alone NT services, (2) with Microsoft Cluster Server (MSCS), and (3) with the watchd component of NT-SwiFT. NT-SwiFT is a fault tolerance package from Lucent Bell Labs (see <http://www.bell-labs.com/projects/swift>). All experiments were conducted on the same machine: a 100 MHz Pentium PC with 48 MB of memory running Windows NT Server 4.0 with Service Pack 4.

For each application, a simple client program sends requests to the application. For the IIS and Apache web servers, the HttpClient program sends two types of requests:

1. An HTTP request for a 115 KB static HTML file.
2. An HTTP request for a 1 KB static HTML file via the Common Gateway Interface (CGI).

For the SQL server, the SqlClient program sends a SQL select request based on a single table. Both HttpClient and SqlClient check the correctness of the server reply. If the reply is incorrect or if the client program doesn't receive any response within a timeout period (default of 15 seconds), ntDTS retries the request, with a second retry if needed. After the client program receives a correct reply or the third attempt fails, the client program outputs information about the success or failure of the requests and the number of retries attempted.

The following results show how the ntDTS tool is useful. First, the tool enables quantitative comparisons of different fault tolerance packages. Second, the tool explicitly identifies situations not adequately handled by the fault tolerance software. Thus, the tool can serve as a testing tool to improve the failure coverage of the fault tolerance software and the reliability of the entire system.

Figure 4 shows results for experiments with IIS as a stand-alone service, with MSCS, and with watchd. Experiments with Apache and SQL server were also performed, but the IIS results are representative of the other results. The possible outcomes are the five outcomes described in Table 1. The chart shows each outcome as a percentage of the total number of activated faults (the calling of a function activates the corresponding fault) for that particular application.

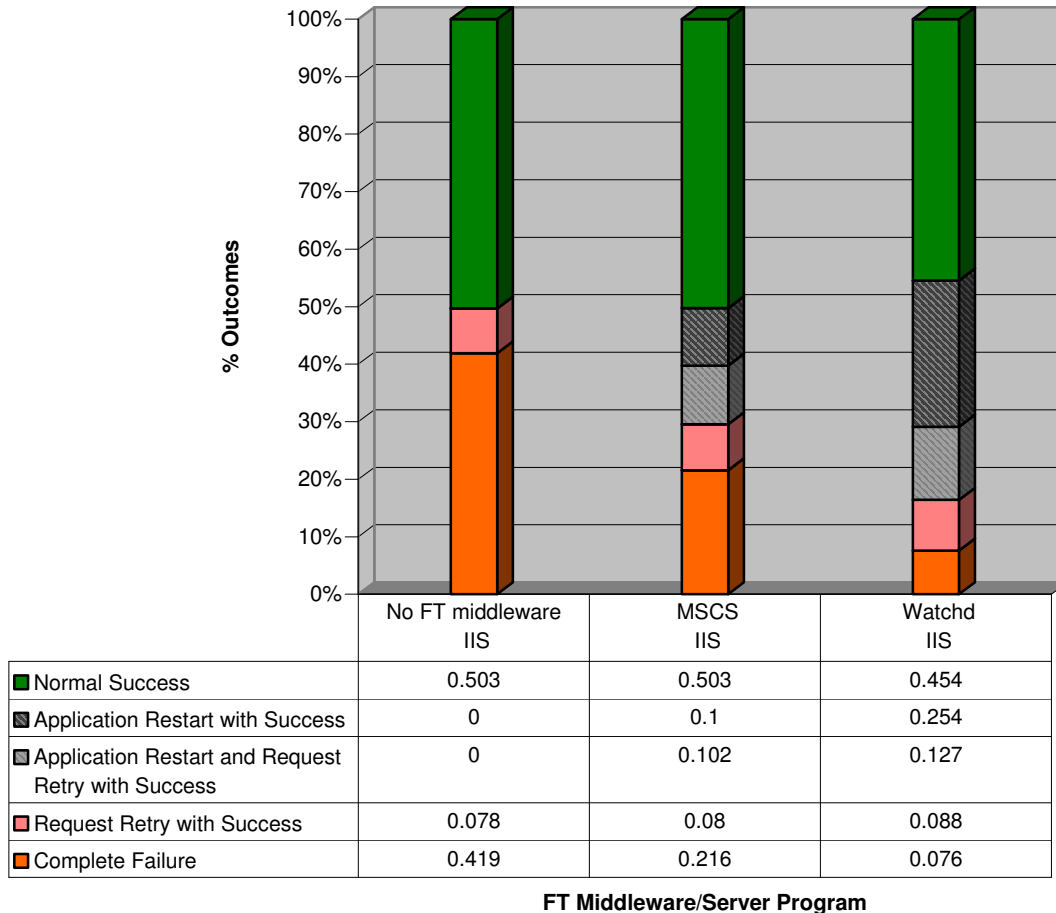


Figure 4 Standalone/MSCS/watchd Comparison for IIS

Table 1 Possible Fault Injection Outcomes

Outcome	Description
Normal Success	The application provided correct responses to all requests with no application restarts or request retries.
Application Restart with Success	After a restart of the application, the application provided a correct response.
Application Restart and Request Retry with Success	After a restart of the application and the retry of at least one client request, the application provided a correct response.
Request Retry with Success	After the retry of at least one client request, the application provided a correct response.
Complete Failure	At least one of the client requests did not succeed. The application provided no response or an incorrect response.

Perhaps the most important and obvious observation from Figure 4 is that both MSCS and watchd are effective in increasing the reliability of IIS. The dark red portions of the figure represent the fault injections that resulted in *Complete Failure* outcomes, i.e., cases where the application was not able to produce the correct response even after repeated client request retries. The complete failure percentages for all applications decreased markedly with the addition of MSCS or watchd.

MSCS and watchd are able to reduce the number of complete failures due to their ability to detect situations in which the monitored application is malfunctioning and then to restart the application. The number of *Normal Success* outcomes remains essentially the same for each application. With the addition of MSCS or watchd, some *Complete Failure* outcomes became *Application Restart with Success* outcomes.

Figure 4 shows that while both MSCS and watchd decrease the number of *Complete Failure* outcomes, watchd does a much better job for the fault set used. In fairness to MSCS, we only used the default failure detection mechanisms in our experiments. It is possible to configure MSCS with custom failure detection mechanisms that interact with and monitor all aspects of the IIS and SQL server programs. However, the creation of these detection mechanisms requires additional knowledge and effort. Thus, the comparison between MSCS and watchd only includes the default MSCS and watchd packages.

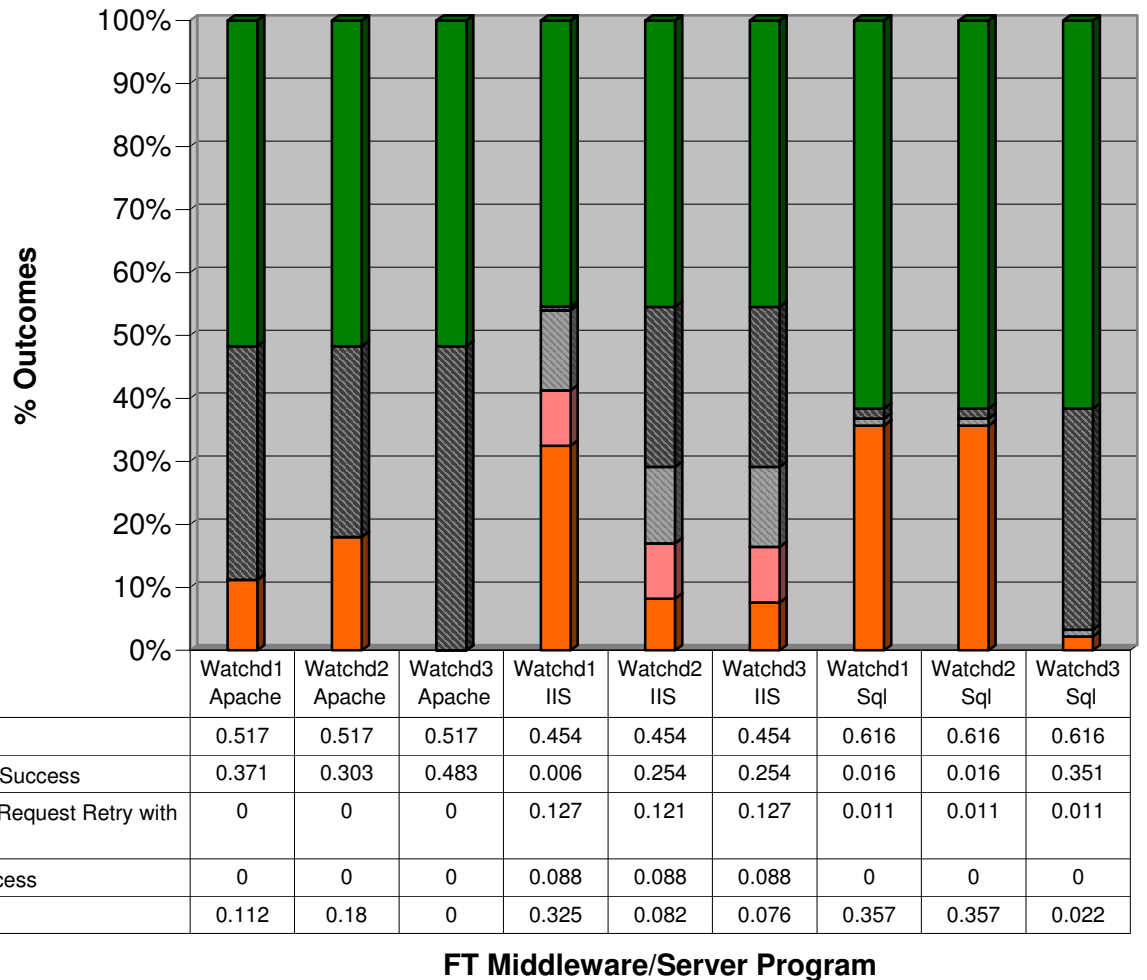


Figure 5 watchd Reliability Improvements

Another important use of the ntDTS tool is the identification of reliability problems. The ntDTS results list the specific faults that caused the application to fail. The failures may be the result of problems in the application, operating system, or fault tolerance software. By replaying the fault scenarios that led to failures, the specific deficiencies can be isolated and corrected. For instance, we were able to pinpoint the existence of small timing problems in watchd involving the startup of services. Fixes to watchd resulted in a version with improved results for IIS. We repeated the process a second time and produced a third version with improved results for all three applications. After each round of improvements to

watchd, the ntDTS tool was used to determine the actual improvement in reliability. We stopped after the third version, since our goal was to show how the ntDTS tool helped us to improve watchd. Figure 5 shows the effect of the improvements in the watchd results on Apache, IIS, and SQL Server. Watchd1 is the initial watchd version, while Watchd2 is the intermediate version, and Watchd3 is the final version. We focused on watchd improvements, since we had access to the watchd source code. Certainly, similar improvements are possible with MSCS or the applications themselves.

What Lies Ahead

Our hope for the ntDTS tool is that it will help increase the popularity of fault injection as a means of reliability testing. We are making the ntDTS tool available free for downloading at <http://www.bell-labs.com/projects/swift> so that people will be able to test the reliability of their applications directly. As a more ambitious goal, we also hope ntDTS will serve as the basis for further development of reliability benchmarks and methods for testing reliable systems.