

# tm: A Stress Tester

Andrew Sutton    Todd Acheson    Dr. Shawn Ostermann

June 9, 1999

### **Abstract**

*tm* is a program designed to test systems by simulating over-exaggerated use. The program can be used to check the accuracy of shared resources of a system or determine if a system will fail under intensive use. The simulation of users is done by threads running scripts. The configuration of *tm* and a description of the scripting language is included. A real test using *tm* is described and the results are analyzed. The program can be expanded to test many different types of services, and is useful uncovering bugs and bottlenecks in current systems.

# 1 Introduction

Ohio University, Computer Services supports centralized mail services for the main campus and regional campuses. Computer Services supports, on average, around forty thousand mailboxes. Through the summer of 1998, the mailbox access was performed through IMAPv2 [1] (Internet Message Access Protocol version 2) and POP3 [2] (Post Office Protocol version 3). Since the IMAP standard had evolved to IMAPv4 [3], it was necessary to convert the mailboxes to the new standard and begin using an IMAPv4 compliant server. We decided on the Cyrus IMAP server from Carnegie Mellon University and began a conversion process on the mailboxes. No problems had been noticed during the testing of the server in our testing environment, but in the production environment inconsistencies in shared IMAP resources were observed shortly after switching to the Cyrus Server.

The observed inconsistencies were not easily reproduced activities that we could easily observe. The continued use of the Cyrus IMAP server demanded that we resolve the problems.

It was suspected that the Digital UNIX implementation of the `mmap()` functionality was broken in some way that allowed two different IMAP connections to have inconsistent views of what should have been consistent shared data. Cyrus IMAP provides Access Control Lists [4] (ACLs) on mailboxes that are maintained in a file that is mapped into every connection's address space using `mmap()` calls. Inconsistencies in ACLs had been observed periodically through standard IMAP clients, and we decided that the ACLs would need to be analyzed.

A design for a test environment was conceived that would meet a number of criteria:

- could run unattended and determine the correctness of an operation and log errors.
- could sustain a predetermined amount of activity against the Cyrus server.
- could be expanded to arbitrary IMAP functionality.

A model was developed that used a “writer”, a “reader” and a “monitor” to check correctness of operations. The reader and writer would need to maintain separate IMAP connections while the monitor would compare the results of the read and write operations. The resultant program took the name “tm” or “test monitor”.

After some initial tests, it was suggested that the program could be expanded to test more than just the ACLs on IMAP mailboxes. After Cyrus expressed interest in including the program for their distribution, *tm* was cleaned up, overhauled, and modified to communicate with multiple services and given the ability to test different aspects of each service.

## 2 The Basics

Since the goal of the project was to locate possible bugs and bottlenecks in various systems using shared resources, *tm* needed to be able to repeatedly perform and verify multiple operations asynchronously. The asynchronous behavior is required to simulate the effect of multiple users accessing the shared resources. We decided the best way to accomplish this was by making *tm* multi-threaded.

Although *tm* simulates asynchronous user behavior, the actual operations performed must be carefully synchronized. There are three steps to performing and verifying an operation. First, *tm* must alter a shared resource (a write operation). Second, the resource must be examined by a read operation. Finally, the results of both the read and the write must be examined for correctness. The operations in the procedure are accomplished by different threads, creating a triplet that “simulates” each user. The threads are called the writer, the reader and the monitor.

Each thread in the thread set (reader, writer and monitor) has one task to perform. The task being performed is dependent on the task that was performed before it. After a task is completed, each thread signals that it is ready to proceed to the next task. When all threads have reported, they are signalled to begin working on the next task.

The first time through, the reader, writer and monitor all report that they are ready to begin. This is the first task and is done to ensure that all threads in the triplet are synchronized. For the next task, the writer performs its write operation and then stores the result in a “deposit”. The reader and the monitor simply report that they are ready to proceed. After all the threads have checked in, they proceed to the next task. This time the reader must perform its read operation on the resource that was altered and store the result. The writer and the monitor report readiness. Finally, the monitor compares the results of the writer and the reader and logs the success of the comparison while the reader and the writer check in. There is one more synchronization in which no member of the set performs an operation, and the process begins again.

Although the entire process may be hard-coded fairly easily, we wanted some flexibility to run several different tests on different systems. This meant that we needed to be able to configure *tm* to start several different thread sets and have each set perform different operations. We also wanted the ability to create background “noise”, or operations used to simulate more user activity. To accomplish this, *tm* requires a configuration file that determines the number and type of threads (reader/writer pairs or noise), and script files that determine the operations performed by each thread.

## 3 Configuration

The configuration file determines the number and types of threads. There are two constructs that *tm* recognizes. The first is a reader/writer pair and the other is noise. The following is an example configuration script:

```

# This is an example configuration file.
{acl_writer1 acl_reader1}
{mbox_writer1 mbox_writer2}
5 {mbox_list}
5 {mail_check}

```

The first line is a comment. Any line beginning with “#” is ignored. The next two lines are declarations of reader/writer pairs. The writer must always come first in the pair, otherwise the output of the program will be skewed. The last two lines are noise declarations. The number preceding the name specifies the number of scripts of that type to create. The names inside the braces determine the file that is used as the thread’s script.

For every reader/writer pair, *tm* generates three threads to complete the triplet, the reader, writer, and monitor. Noise threads are not monitored and do not require synchronization with any other threads. Users should note that *tm* does not place restrictions on the number of threads created, but the system does. Generally, tests should not exceed more than 500 threads. The number may vary from machine to machine, but starting too many threads can make any computer start thrashing.

## 4 The Scripting Language

In order to accomplish our requirement of testing different resources, we developed a limited scripting language that would enable the scripts to complete their tasks. The language is not extensive and only offers the ability to do what is required. Scripts may open services, issue commands, loop, wait, and close services.

The scripting language relies on service modules to facilitate the server transactions. Users may define more service modules to enable *tm* to communicate with different servers. Currently the only module written is the one that communicates with IMAPv4.

The following is a description of each command in the scripting language.

### **connect** *servhostport*

This command sets up the thread’s data stub based on the service supplied by the script. It then allows the service module to try to connect to the server given by the host and the port. If the port is not given, the default port is used for the well known service. However, the service and the host must be given.

### **command** *cmd\_str*

Pass the command string to the service module. Since we rely on the service module to actually transact the operation, *cmd\_str* is given to it to be translated. If the script is a reader or a writer, the service module must synchronize the threads. The service module must also determine if the result of the write/read/compare needs to be checked or logged.

**loop/end** *num*

Perform the operations between loop and end *num* times. This command is handled by the script module.

**wait** *sec|rand*

This causes the thread to sleep for either *sec* seconds or a random amount of time if *rand* is specified. The maximum random wait is defined in *global.h*. This command is also handled by the script module.

The following is a list of suggested additions to the scripting language that have not yet been implemented.

**login** *user passwd*

This was proposed a generic login for all services.

**transmit** *trans\_str*

Transmit data specified by *trans\_str* to a service. Much like command, the service module must translate *trans\_str*. However, unlike the command statement, transmit statements are not checked for correctness.

**loop/end** *num|forever*

This was not really a new addition. It was proposed we give the scripts the ability to continue looping infinitely.

## 5 A Real Test

After it had all been put together, tested and debugged, we began the process of using *tm* to test the new email server. We wanted to separate and record all of our tests, so we modified *tm* to change into a test directory that was specified on the command line. This allows us to keep track of the different tests by keeping the files in different directories.

The following is a description of a real test that was run against Ohio University's primary account server, and a description of some of the problems that arose.

```
# oak_test_1 - configuration file
# 2 reader/writer pairs operating on ACLs
# lots of background noise
{writer1 reader1}
{writer2 reader2}
150 {noise1}
150 {noise2}
150 {noise3}
```

We only configured 2 pairs of readers and writers, and 450 noise threads doing arbitrary, repetitive tasks for a total of 456 threads.

```

# oak_test_1 - writer1
# change ACLs on a mailbox several times
open imap oak.cats.ohiou.edu
command LOGIN user pass
loop 5000
command SETACL MAILBOX inbox sutton lprew
command SETACL MAILBOX inbox sutton lri
command SETACL MAILBOX inbox sutton cda
end

```

The writer's task is to login and change the ACLs on user's inbox fifteen thousand times, rotating between the three ACL settings. After this is completed, close the connection and exit.

```

# oak_test_1 - reader1
# read ACLs on a mailbox for every change
open imap oak.cats.ohiou.edu
command LOGIN user pass
loop 5000
command GETACL MAILBOX inbox sutton
command GETACL MAILBOX inbox sutton
command GETACL MAILBOX inbox sutton
end

```

The reader performs the inverse operation of the writer. For every place the writer alters an ACL, the reader looks at the ACL. The reader, writer and monitor synchronize on every "command" operation. For any SETACL/GETACL pair, the order of operations is SETACL, GETACL, and log the comparison of the results.

```

# oak_test_1 - noise1
# open a connection, and do something lame
open imap oak.cats.ohiou.edu
loop 5000
command CAPABILITIES
command NOOP
end

```

The noise scripts make the server perform mundane tasks. This script could be modified to read email, but there are 150 copies of it hitting the server at the same time, and that would probably be more background noise than we want for this particular test.

The other scripts used (writer2, reader2, noise2, and noise3) are similar in appearance and function to the previous scripts. The test was run from a DEC Alpha 2400 against the oak server. Oak is a DEC Alpha with 3 400MHz processors, 3 GB of RAM and 250GB of disk space. It is running Digital Unix 4, and is by no means a trivial machine. During the test, *tm* created and ran

456 threads against the Cyrus IMAPv4 email server on oak for some interesting results.

There are three files created by *tm*. The first, *tm.pid*, is a pid file that simply stores the process id. The second, *out.log*, is a dump file that contains some synchronization information and every interaction with the remote server. It is used primarily for debugging, and *tm* can be ran without creating it (the file becomes enormous on long tests). The third file, *tm.log*, is the interesting one. It contains the results of every write/read/compare operation. If there are inconsistencies any time during the test, they are noted here.

The first time the test was run, many of the ACLs did not match after they were changed. From this we concluded that there was a synchronization problem with the Cyrus server. It was also noted that during the test, real IMAP users were experiencing severe lag times when logging in to check their email or open a new mailbox. In addition to those problems, several sendmail processes were stuck in a “U” state, meaning “unchangeable”. This meant that the processes were never scheduled to execute and had to be killed by hand. The abuse taken by the system was enough to warrant a reboot.

After some digging, we traced most of the problems to a lock in the email server. Cyrus keeps a file containing a list of all of the mailboxes. Every time one of these is accessed, the file is locked, searched, modified (depending on the operation) and unlocked. With four threads repeatedly requiring the server to perform that sequence of operations, the server did not have time to service anybody else. Obviously, the server was taking too much time during a critical section effectively locking users out of the system.

The repetitive nature of *tm* also uncovered a Digital Unix operating system bug. The stalled sendmail processes and the inconsistent ACLs could be traced back to a failure in the operating system to control locks under stress, and therefore lose track of processes. After some phone calls and some email, new versions of both systems were released. Recent tests with similar configurations show that the problems have been fixed.

## 6 Problems and Warnings

Despite the fact that *tm* has proven itself to be a useful system testing tool, it can have disastrous effects on the computers it is targeting. The repetitive nature of the program lends itself very well to being a stress tester, and breaking systems. Using it requires some knowledge of the products being tested and the system the products are running on.

Misuse of *tm* as a testing tool can easily lead to down time. If the test is too strenuous for the service or the system, it can cause the targeted machine to start thrashing. Also, if the operating system has bugs in it, *tm* can cause the entire computer to hang. For this reason, we introduced the “wait” command to the scripting language. This allows the threads to take a short break from their usually ferocious repetition. It is recommended that tests on production servers include the wait statement between commands. This allows other users

an opportunity to use the system, filling the time gap between script commands.

There are also some security issues of using *tm* as well. First, most services require a login, and *tm* does not offer anything in the way of encryption or security. One solution to this problem is to use ssh port-forwarding, a feature of secure shell that allows connections to be made securely through it's own connection. This assumes that ssh can forward large amounts of data correctly.

Secondly, *tm* can be used very efficiently for denial of service attacks. An example of this would be to overload a web server. Most web servers support some option that allows a maximum number of connections (the default for the Apache web server is 150), and most UNIX computers that *tm* can be compiled on will allow users to run over 150 threads. A single line configuration file, and a three line noise script can force almost any web server to refuse connections from real users.

## 7 Expansions

Since *tm* was designed to be flexible, we left out everything that made it specific to the IMAPv4 protocol and the Cyrus implementation. The only thing that *tm* does is spawn threads, read scripts, and log results if the service module decides that is necessary. The rest of the implementation is left for users to create. This includes implementation of protocols, parsing of command strings, and comparison of results. The facility for doing all of that is provided by *tm*.

The services that would best be tested by *tm* are systems that use shared resources. IMAP, according to the Cyrus implementation, uses ACLs for accessing mailboxes. The access control lists allow users to share mailboxes, and therefore share data between processes. Some examples of services that *tm* can test effectively are LDAP (Lightweight Directory Access Protocol) servers, databases, and other information servers. however, protocols like HTTP does not share information and is not easily tested for consistency. However, *tm* can be used on systems like web servers to determine a breaking point.

## 8 Current Implementation

Currently, *tm* is in version 1.1. The only service module that has been written is the IMAP module, and that only works for the Cyrus implementation (IMAPv4 did not originally contain specification for ACLs). Ports are available for Digital Unix, SunOS, and IRIX. A Linux port should also be available soon. The usage of the program is as follows.

```
usage: tm [-n] [-o ofile] [-l lfile] [-p pfile] [-d dir]  
-n          Turn off debugging. This prevents tm from producing out.log.  
-o ofile   Make tm dump debugging output in ofile instead of out.log.  
-l lfile   Make tm log comparison results in lfile instead of tm.log.  
-p pfile   Record the pid file in pfile instead of tm.pid.  
-d dir     Force tm to execute in the directory dir.
```

## References

- [1] M. Crispin. Interactive Mail Access Protocol - Version 2, jul 1988. RFC 1064.
- [2] J. Meyers and M. Rose. Post Office Protocol - Version 3, may 1996. RFC 1939.
- [3] M. Crispin. Internet Message Access Protocol - Version 4rev1, dec 1996. RFC 2060.
- [4] J. Meyers. IMAP4 ACL Extension, jan 1997. RFC 2086.