Ubermicro Phase 2

CS4210

Version 1.0

March 27, 2006

Jose Caban & Puyan Lotfi

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Puyan Lotfi	Proofreading complete	03/27/2006
0.9	Jose Caban	Completed text and graphs	03/27/2006
0.0	Jose Caban	Layout created	03/27/2006

Contents

Figures

1 Introduction

1.1 Overview

The server has been built to be robust in the socket mode. In shared memory mode, the server does not properly handle 404's, directory listings, etc, as these were part of microhttpd. Since the purpose is to benchmark the different implementations, it was decided that there was no reason to ensure robustness at this stage: this will be handled during refactoring for the next phase.

1.2 Constructs used

The system was built using SysV Semaphores and Shared Memory. The shared memory segments are handled by a thread safe queue controlled by the proxy server in shared memory mode. Signals are properly handled to free the shared memory.

2 Installation and Execution

2.1 Installation

- Extract the provided ZIP file into whatever directory you desire
- Run "gmake" from the "server/" directory to build the server executable.
- Run "gmake" from the "proxy_server/" directory to build the proxy.
- Run "gmake" from the "client/" directory to build the client.

2.2 Execution

All tests were run on Faye (asskoala.servebeer.com, login can be supplied if desired) and Titanium. Development was also done on SuSe Linux.

Tests were run on Samwise (enterprise lab) to ensure that the application ran correctly on the CoC machines.

2.2.1 Server

The server must be executed first in shared memory mode, the configuration file is found in the server/ directory. To run it, simple "cd server/" and execute "../bin/server/l33t_server". The shared memory mode must be specified on the command line. Executing the above will display the runtime information.

```
#COMMENT
VariableName:Value
```

Figure 1: Config File format

Proper execution of the server should look similar to the following:

```
asskoala@Faye Project 2 % bin/server/l33t_server 1 server/l33t_server.conf
* Beginning l33t_server Initialization *


** Using 16 threads
** Using home_dir: .
** Using Shared Memory.
* l33t_server started *
** Initializing Thread Pool
*** Listening on Port: 1337
```

Figure 2: Executing the Ubermicro server

Please look at the provided configuration file for example settings.

Finally, to quit the server, simply type q and then enter. All other input will be ignored. All signals and typing in 'q' will cause the shared memory segments to be marked for deletion. If the server is killed before the proxy, the segments will stay in memory until the proxy server is killed.

2.2.2 Proxy

The proxy must be started after the server if in shared memory mode. Execution is the same as with the server. The default port is 1338 and is changeable in the configuration file (the configuration parameters are the same as in the server, though only MAX_THREADS and PORT_NUMBER parameters are used).

2.2.3 Client

The client can be executed much like the server, simply run "bin/l33t_client ADDRESS PORT_NUMBER TESTFILE". Note that the client does not support DNS resolution and, as such, the ADDRESS must be specified as an ip address (usually 127.0.0.1). Some example test files are included in the /client/tests directory. These files use standard XML formatting. **Update:** For the proxy tests, the filenames must have http://localhost/ specified before the actual filename. No other changes are necessary in the server. In addition, the client now waits for all threads to spawn before executing. A simple test file is shown in Figure 3:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- XML File for test execution-->

<Tests>
      <simpleTest>
      <!-- The number of threads to spawn to make requests -->
      <numthreads>32</numthreads>

      <!-- The number of times the request will be made per threads -->
      <iterations>10</iterations>

      <!-- The files to retrieve -->
      <filesToRetrieve>

         <!-- The file names, use / for directories -->
         <file>/index.html</file>
         <file>/pwner.jpg</file>
         <file>/makefile</file>
         <file>/special.zip</file>

      </filesToRetrieve>
   </simpleTest>
</Tests>
```

Figure 3: Sample Test file

Currently, the client only executes the first test in the file. Explanation of how it works is in section 3.2.

3 Implementation

3.1 Server & Proxy

3.1.1 Networking

Our use of sockets was done in a way that allowed for cleaner and more reusable code. As time goes on, we will refractor more and more pieces of code so that we can have more code reuse. The idea was that we could isolate the TCP connection process into three major components: setting up socket data structures and binding to a port, accepting connections, and handling accepted connections. These are separated into various functions. We also loop our recv() calls to ensure all the data we are expecting arrives properly from the TCP bytestream.

3.1.2 Threading

The threading model uses a simple producer/consumer model with a linked list queue of (relatively) infinite length to store sockets. The producer establishes the connection and places the socket in the queue. The workers wait until the queue has a socket and handle the connections as soon as they can. The queue is atomically operated upon and is the only place in which mutexes are locked within the source code.

3.1.3 Shared Memory

The shared memory implementation used is the SysV system. SysV semaphores are used for synchronization. The shared memory segments are initialized by the server. The proxy attaches to the segments and uses an internal mutex locked queue to handle the shared memory segments built by the server in a "memory pool" fashion similar to that used by the thread pool implementation. To change the number of shared memory segments, the header file must be modified and both servers rebuilt. The parameter is the "NBUFF" parameter. The default is 4 buffers and this is what was used in all tests.

3.2 Client

The client takes in three parameters, one being the server's IP, port number, and an xml file. The xml file specified how many threads the client should spawn, what files to request, and how many times the file should be requested. Therefore, the file requesting is divided among the threads, who each request the same file a certain number of times.

4 Testing and Benchmarks

4.1 Test Beds

Samwise (used for CoC runability):
Dual 3.06GHz Xeons
1GB RAM
SCSI disks
RedHat Enterprise Linux AS 4

Titanium:
1.6GHz Sempron 64
512MB RAM
SATA RAID 5 disk array
FreeBSD 6.1-PRERELEASE (i386)

Faye:
Dual AthlonMP 2100+
1024MB RAM
Dual 300GB 7200rpm disks
FreeBSD 6.0-RELEASE (i386)

4.2 Titanium tests

4.2.1 Simple File Test

The simple file tests request build 4 threads and request 4 different files of the same size. The small files are 5K, medium 500K, and large 5MB. The discussion is lumped before the graphs. Both the server and proxy run with 16 threads and, in shared memory mode, use 4 shared memory buffers.

As can be seen in Figure 4, the overhead of the semaphores in the shared memory mode cause a loss in performance compared to using sockets. However, as the file size increases, the performance increase versus sockets is huge with a 60-second time difference per thread in the large file test in favor of the shared memory model.

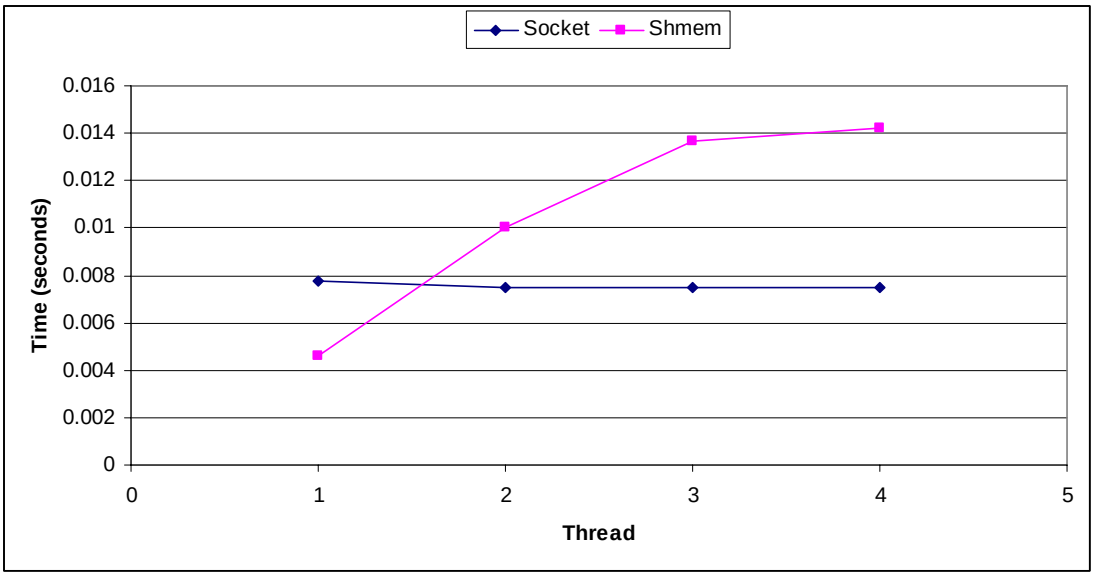

Figure 4: Titanium Simple Small File Test

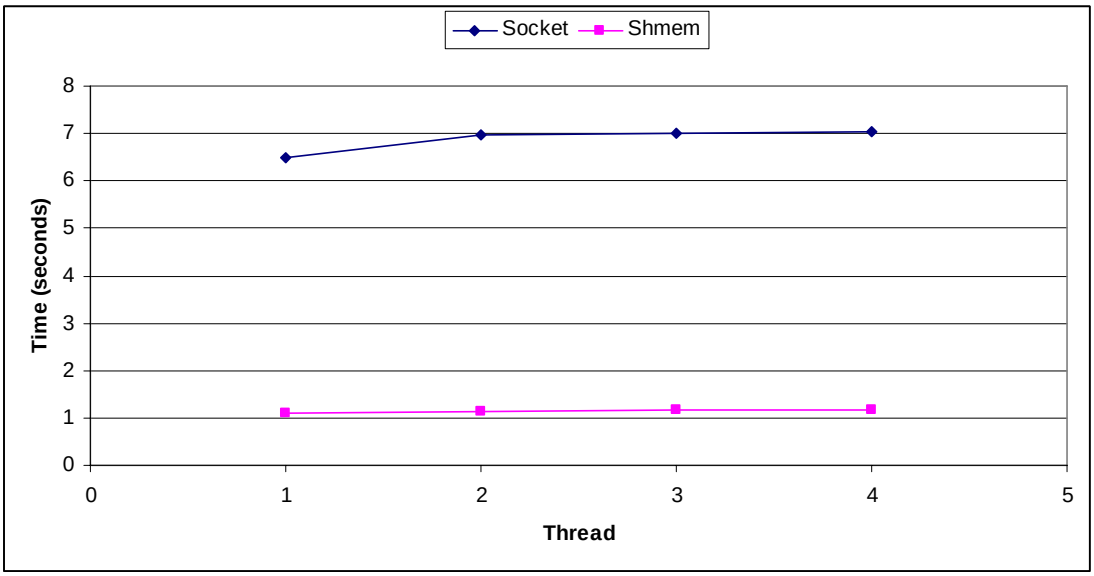

Figure 5: Titanium Simple Medium File Test

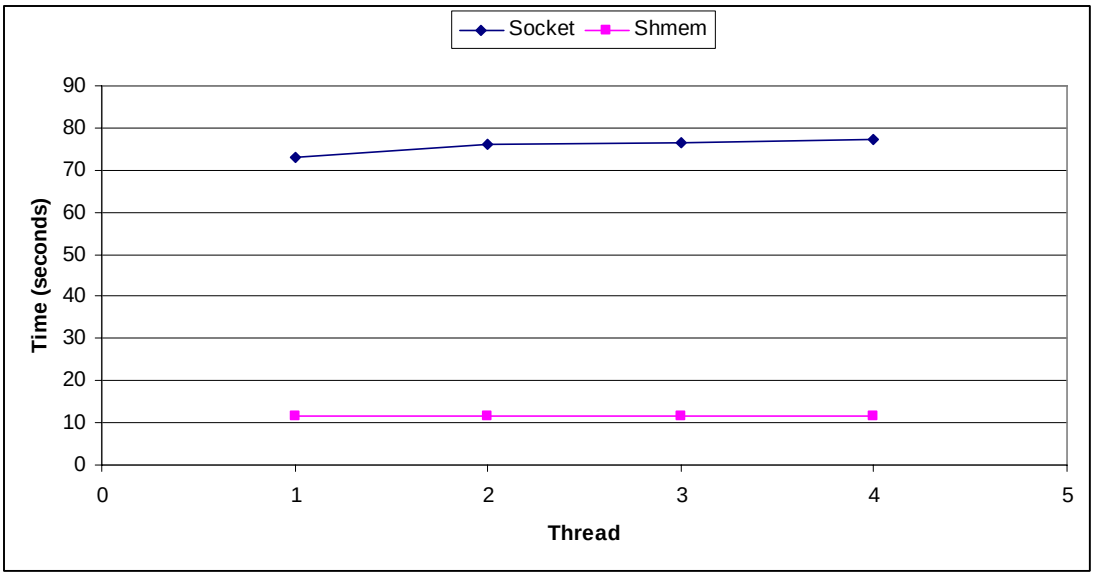

Figure 6: Titanium Simple Large File Test

4.3 Faye tests

4.3.1 Simple File Test

The simple file test is described in section 4.2.1. In the multiprocessor environment, a few things are noted. The Shared Memory test always performs better than in the socket model. However, the socket performance on the multiprocessor environment is significantly better than in the Titanium performance. This can be due to a few issues, however. Faye uses the ULE Operating System Scheduler, which gives is fully preemptable and is O(1) with regard to scheduling a process. Titanium uses the BSD4.4 scheduler, which is not kernel pre-emptable and is O(n). This difference is enough to result in such a drastic hit due to context switching on Titanium.

Regardless, the performance of the shared memory relative to the socket server on Faye results in a victory for the shared memory model.

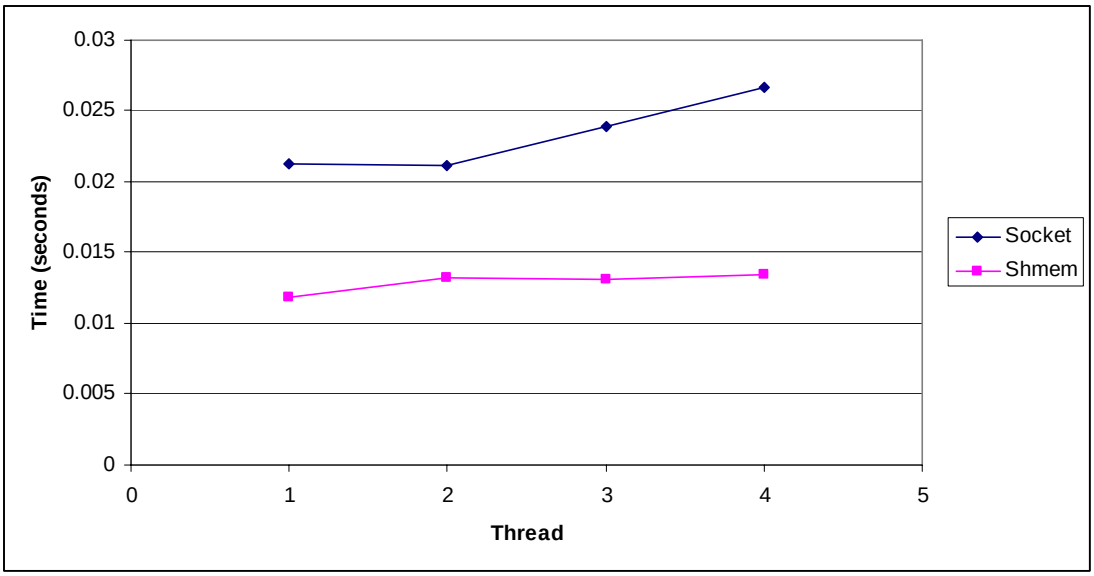

Figure 7: Faye Simple Small File Test

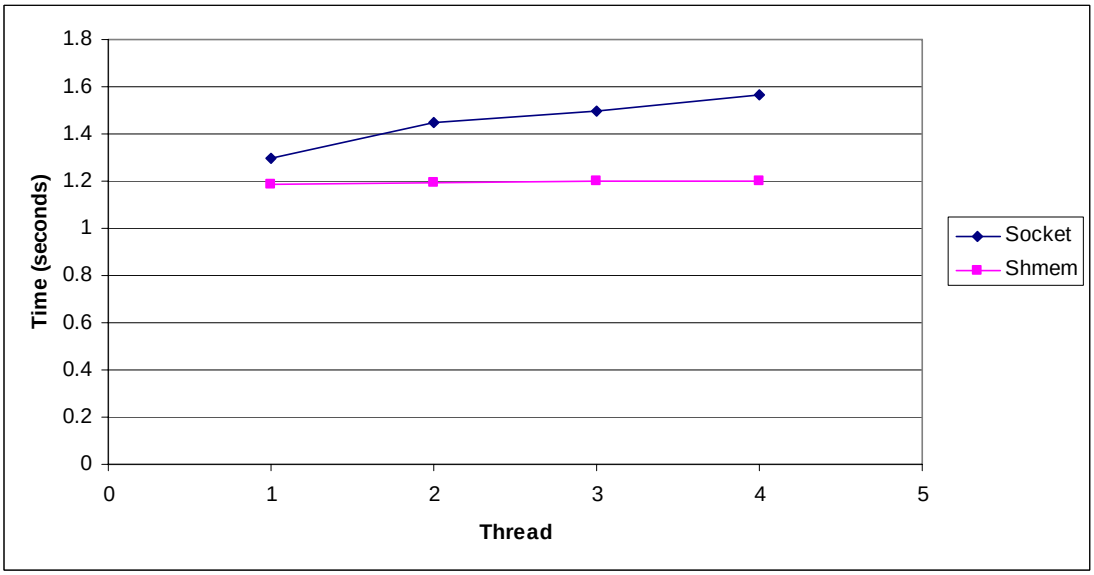

Figure 8: Faye Simple Medium File Test

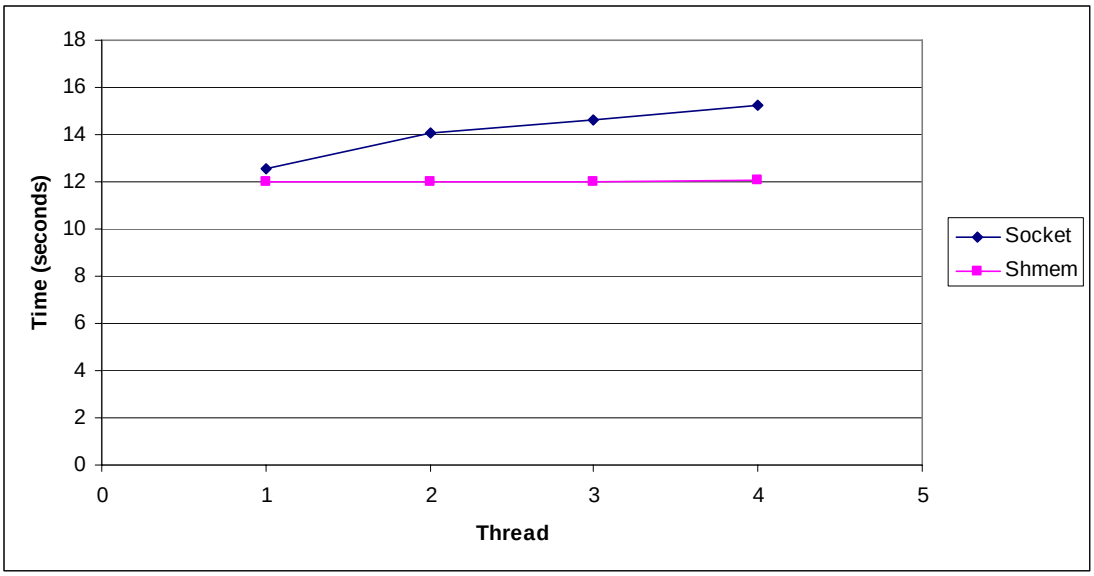

Figure 9: Faye Simple Large File Test

4.3.2 Advanced File Test

The advanced file test spawns 16 threads and makes requests on the proxy. However, the proxy and server still have the same settings: 16 threads each and 4 shared memory segment buffers. The small file test accesses 5K files and the medium file test accesses 500K files.

In the small file test, the times are relatively the same except for a large spike in completion time for the socket mode. This could be due to some background process switching or simply the overhead on the sockets themselves.

Despite only 4 shared memory buffers, the shared memory mode greatly outperforms the socket mode in the medium file test.

The Advanced small file test can be used to test the latency of the two modes. The latency differences are nominal between the two.

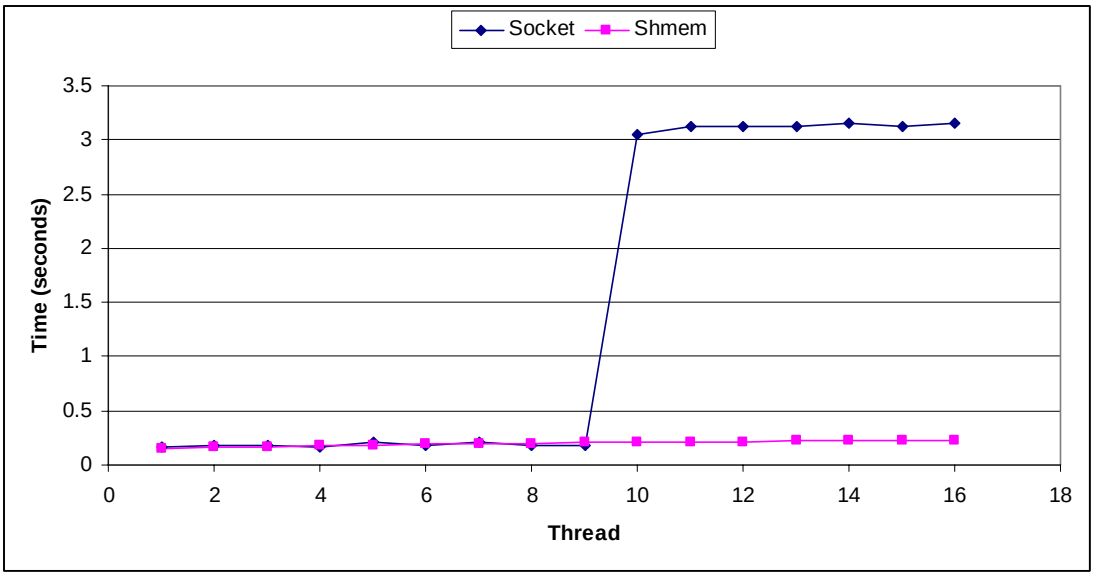

Figure 10: Faye Advanced Small File Test

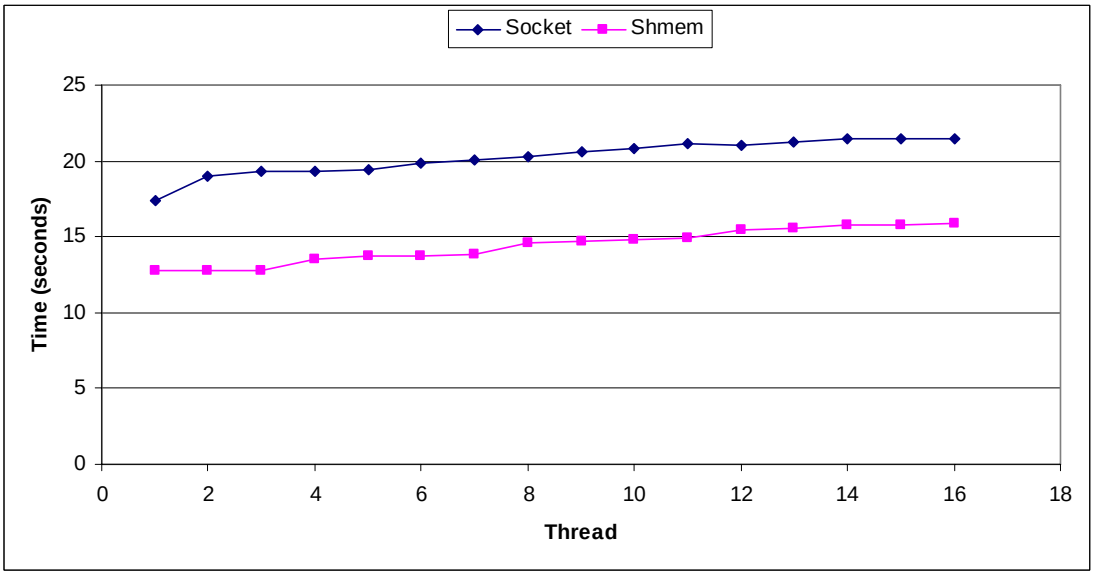

Figure 11: Faye Advanced Medium File Test

4.3.3 Throughput

The throughput test spawns 12 threads and requests 10MB files. In this test, the shared memory mode once again outperforms the socket mode again despite only 4 shared memory

buffers. The actual file copying from process to process completes quickly, the majority of the time is used up in sending the file over a socket from the proxy to the client.

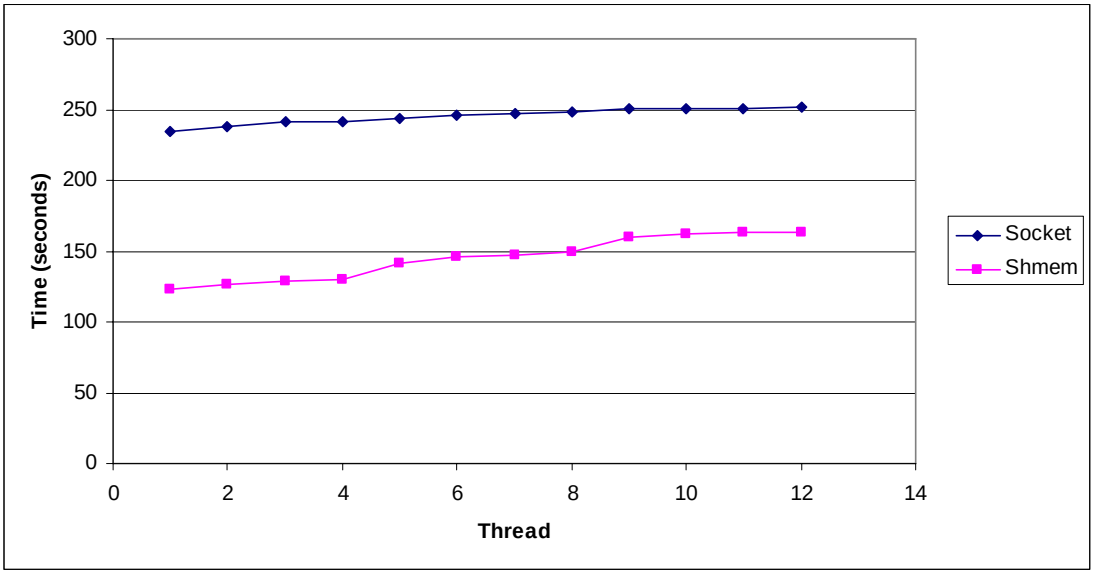

Figure 12: Faye Throughput Test

5 Conclusion

The project served well to highlight the advantages and disadvantages in using shared memory. It is significantly more complicated to use shared memory over sockets. However, the performance increases are well worth the changes in a performance important application.