Ubermicro Phase 3

CS4210

Version 1.0

April 24, 2006

Jose Caban & Puyan Lotfi

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Jose Caban	Document Complete	04/24/2006
0.8	Jose Caban	Document complete minus test results	04/23/2006
0.0	Jose Caban	Layout created	04/23/2006

Contents

Figures

1 Introduction

1.1 Overview

The server will work perfectly assuming the RPC Servers do not crash: error recovery exceptions are not handled with regard to RPC requests.

1.2 Constructs used

The system was built using Sun RPC as the communication protocol and handles JPEG resizing as part of the fancy stuff it does.

2 Installation and Execution

2.1 Installation

- Extract the provided ZIP file into whatever directory you desire and execute the following commands (comments with a #):

```
% cd Project\ 3/

# Building the JPEG Library
% cd jpeg_resizer/jpeg-6b
% ./configure
% make

# Building the RPC Server
% cd ../rpc_server
% make -f Makefile.rpc

#Building the Proxy
% cd ../../proxy_server/
% make
```

Figure 1: Building the Proxy and RPC Server

2.2 Execution

All tests were run on Samwise and should work on any enterprise lab x86 Linux box.

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 2: 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 3: 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.

The proxy server does not support Shared Memory in this implementation, so there is no point in passing anything other than a 0 to the l33t_server.

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).

The RPC Servers can be specified on the command line (shown in the following page). The RPC server itself can be run with the simple command in its directory:

```
# The RPC Server can be run on whatever system you want:
% cd Project\ 3/jpeg_resizer/rpc_server
% ./jpeg_server
```

Figure 4: Executing the RPC Server

```
# Running the proxy
## Without the RPC Server, straight up hosting:
% cd Project\ 3/proxy_server
% ../bin/proxy/l33t_proxy 0

## With the RPC Server, assuming an RPC on localhost
% cd Project\ 3/proxy_server
% ../bin/proxy/l33t_proxy 1 127.0.0.1

## Assuming many RPC Servers
% cd Project\ 3/proxy_server
% ../bin/proxy/l33t_proxy 1 127.0.0.1 128.61.42.147 10.10.10.1 192.168.0.2
```

Figure 5: Executing the Proxy

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:

Update: The current client program had given us problems with this stage of the project (Project 3), and since it was ok to do testing with any other http client, we decided to write some scripts with bash and wget. Wget supports proxies, and works perfectly with our own proxy. Since it is a widely used console based http client (just fetches files and saves them to the current working directory) it seems ok to use as a testing tool.

```
#!/bin/bash
export http_proxy="http://127.0.0.1:1338"
var0=0
LIMIT=100
echo "Starting latency test: 4 jpegs fetched $LIMIT times."
date > start_date

while [ "$var0" -lt "$LIMIT" ]
  do
      var0=$(($var0 + 1))
      time wget \
       http://asskoala.servebeer.com/~nayupuyan/small_one.jpg
      time wget \
       http://asskoala.servebeer.com/~nayupuyan/small_two.jpg
      time wget \
       http://asskoala.servebeer.com/~nayupuyan/small_three.jpg
      time wget \
       http://asskoala.servebeer.com/~nayupuyan/small_four.jpg
done

echo "Start time: "
cat start_date
echo "End time: "
date
rm -f *.jpg* start_date
echo "End of latency test!"

exit 0
```

Figure 6: Sample Bash Script

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

Update: These are 4 scripts, 2 do throughput tests (a few large images) and 2 do latency tests (many small images). For each of the 2 types of tests, there was a verbose version and a quiet version. Each wget call is run in the background to try to force some concurrency in the tests, but the script is also run in 2 instances in 2 separate consoles. This truly tests the concurrency and latency of our proxy when using multiple RPC servers. In our tests, we ran 4 instances of the RPC server on 4 different machines.

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.

Update: When running the Proxy in JPEG Resize RPC mode, the heavy memory usage causes memory corruption if running a high number of threads. Because of this, the heavy load tests were run using 3 proxy threads to avoid memory corruption. In web browsing mode, there have never been any issues using 16 threads. The problem likely arises due to the heavy hitting on the server and rapid malloc/dealloc of somewhat large segments of memory. The memory limitations of the pthreads then result in corruption.

3.1.3 RPC Server

The RPC communication protocol used is the Sun RPC system. The RPC Server, located in the jpeg_resizer/rpc_server directory, executes and waits. When the proxy receives a request for a file ending in ".jpg" or ".jpeg", it downloads the file into memory (while stripping the HTTP OK header). The proxy then sends a variable size opaque file type to the RPC server, which simply contains the JPEG file itself.

The RPC server then writes this jpeg to a temporary file, opens a file descriptor to the file, passes it to the JPEG resize function provided by the JPEG library, destroys the temporary file, and returns a variable size opaque value back to the proxy. **Note that the temporary file is the same. This is important if multiple machines are running an RPC server (or even the same machine) in the same, shared NFS directory.** The storage space taken up by the malloc'd file from the JPEG library is released before a new JPEG is handled. This was done in this manner to allow for future work in caching, but there is no leaked memory only lazy freeing.

Finally, the proxy receives the opaque file from the RPC Server, returns it to the client, releases the opaque memory using xdr_free and continues its work. Because this is time consuming, it is not recommended that the proxy be run with fewer than 16 threads to sustain low latency web browsing.

This design was chosen to allow the proxy to run on a system connected to the internet while the RPC server itself can run on a system that has no internet access and required very little modification to the JPEG library code itself – acting very much like a wrapper.

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 requests are divided among the threads, who each request the same file a certain number of times.

Update: There really are not any parameters needed to run the test scripts. There is one environmental variable used by wget (http_proxy) to set it to use a proxy server. In the scripts it automatically is set to http_proxy="127.0.0.1:1338" because that is the configuration used in testing.

4 Testing and Benchmarks

4.1 Overview

Here is a list of machines used for testing:

Helsinki: Dual Hyperthreaded 3.2Ghz Intel Xeons 1MB L2 Cache

Frodo: Hyperthreaded 3.06Ghz Intel Xeon 512KB L2 Cache

Samwise & Boromir: Same specs as Frodo (both are the same model IBM ThinkCenter)

Stmartin: Hyperthreaded Intel Xeon 256KB L2 Cache

The proxy and test scripts were both run on *Boromir*. The four different RPC servers will run on each of the listed machines. There are four large and four small images at: http://asskoala.servebeer.com/~nayupuyan. The four small files are each about 5KB, and the four large files are each about 1 to 1.5MB. There are issues when browsing to sites that do redirects, which is why we did not use any CoC Apache servers.

The latency test fetches each of the four small images 100 times; the throughput test fetches each of the four large files 10 times. For the latency tests, we felt the need to run the script in 2 separate consoles at the same time because the images were so small that the requests would generally finish in order (because they were done quickly).

4.2 Latency Tests

It should be noted that the requests are not generated at the same time, but they achieve the same result. The test was run with and without the RPC feature (shown in Fig. 7). The Average runtime was lower without the RPC:

	w/o RPC	w/ RPC
Average (sec)	29.66666667	34

Table 4.2.1: Average Latency Test Result

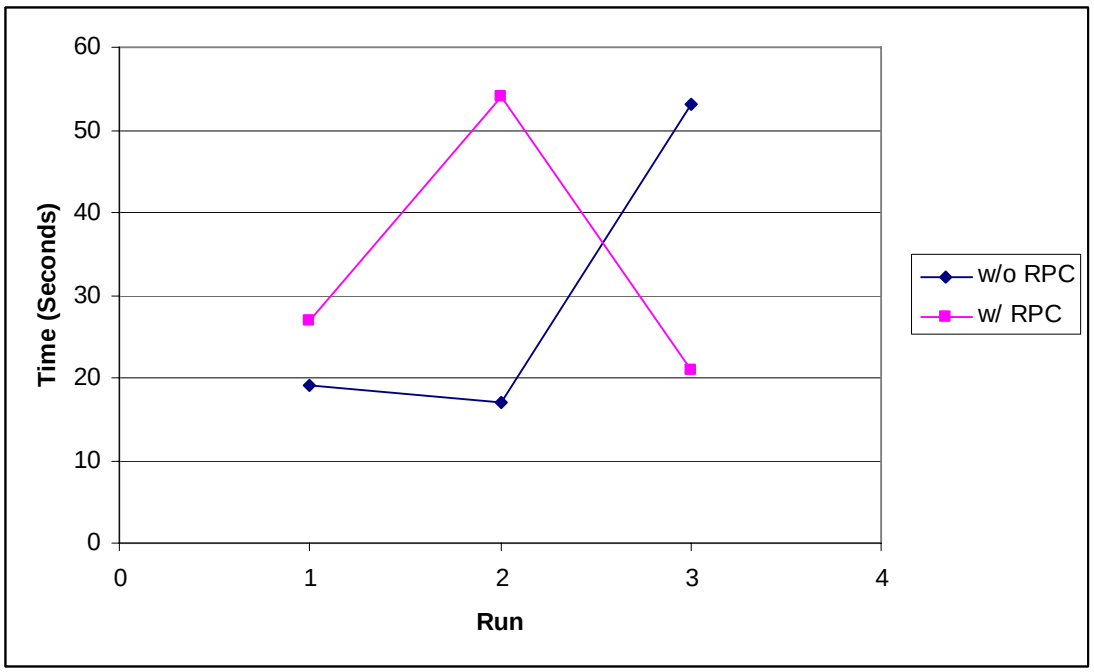

Figure 7: Latency Test Results

4.3 Throughput Tests

The throughput test, described above, was also run on both the RPC and non-RPC enhanced proxy server. The average times were significantly lower on the non-RPC server:

	w/o RPC	w/ RPC
Average	19.5	35.5

Table 4.3.1: Average Throughput Test Result

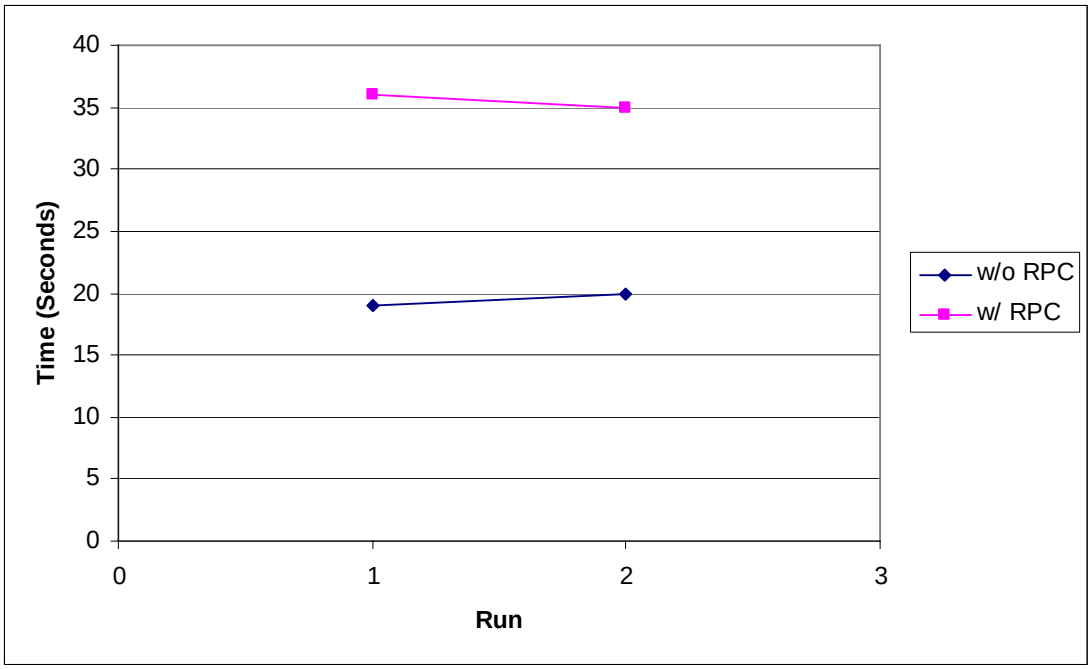

Figure 8: Throughput Test Results

4.4 Conclusion

The addition of the JPEG resizing code does not significantly affect the performance of the proxy server: it would, however, greatly aid users of dial-up or slower internet connections.