Project 2

Report

March 27, 2006

Vladimir Urazov

Omar Zakaria

Contents

1 Building The Sources

The source code for the project is broken up into the following directories:

common – this directory contains the code common to the executables that can be built in the other directories, such as some of the networking functionality, the multi-thread-safe list data structure, cross-platform semaphore and shared memory implementations, etc. This directory contains its own make file, which builds a static library named libcommon.a from all of the source files. This library can then be statically linked in the build of the server, the proxy and the client to provide the necessary functionality to those executables.

server – this directory contains the HTTP server code. The make file contained in this directory produces an executable in the bin directory named server, which can be used to start up the HTTP server on the local machine.

proxy – this directory contains the proxy server code, which can be used to route requests from the client to a remote or local HTTP server and back. The make file will produce an executable named proxy in the bin directory.

client – this directory contains the code for the testing client, used to do performance and robustness tests on the server. It also builds into an executable with the make file, but assumes that the common library has already been built. The program produced by the make file in the bin directory will be named client.

tools – this directory contains some handy tools that can be utilized if the proxy or the server don't behave. For instance, the clean_shmem.sh script will use the output of the ipcs command to find and kill all memory segments allocated by the current user on the machine. This can be useful if the proxy was started with the use shared memory option and for some reason crashed or otherwise terminated without performing proper cleanup.

The simplest way to go about building the entire project is by typing make in the root directory of the project. That will create a bin folder there and will build all the components of the project into it. Hence, once the make process completes, the directory will contain the following files: client, libcommon.a, proxy, server. The client, the proxy and the server can then be immediately run.

2 Running The Executables

The project build process creates three executables: server, proxy and client.

server is the HTTP server executable. It supports the following command-line parameters:

port – the port that the server will listen for connections on. The default value for port is 1337.

pool-size – the number of threads to create to handle client requests. The default value is 16. Note that due to per-user constraints Linux imposes on the number of available threads, the server will fail creating enough worker threads if this parameter is set too high.

home – the path to the root document directory, relative to the directory from which we are starting the server. All the documents will be served relative to the home directory. The default value for this parameter is the current directory: ‘.’.

So, for example, if we are in the root directory of the project, after running make we can start up the HTTP the server as follows:

```
bin/server port=80 pool-size=32 home=./home
```

(Note that the way parameter parsing works, there should be no spaces between the names of the parameters, the equal signs and the parameter values. Each space will be treated as the start of a new parameter.)

The server has certain security restrictions: for instance, it will not allow clients to go up the directory hierarchy, and also it will respond with forbidden HTTP status, if the user running the server has no permissions reading the requested file.

Additionally, note that the server will allocate a piece of shared memory when started, and so it should not be killed with a termination signal – instead, one should hit ‘q’ in the console, which will trigger a clean termination of the server. For more information on the use of shared memory by the server, see the design section of this document.

proxy is the HTTP proxy server executable. It supports the following parameters:

port – the port that the proxy server will listen for connections on. The default value is 8080.

pool-size – the number of threads to create to handle client requests. The default value is 16.

use-shm – this argument takes in a number, such that zero means we are not using shared memory, and non-zero means we are using shared memory. The default value is 0. For more on the use of shared memory by the proxy, check the design section of this document. Note that if this argument is not provided or explicitly set to zero, none of the parameters whose names start with shm- have any effect on the execution of the proxy server.

shm-count – the number of shared memory chunks to allocate for use by the proxy server. The default value is 4. This is subject to shared memory constraints imposed by the system.

shm-size – the size of each of the allocated shared memory segments. The default value is 256. This is also subject to local system constraints.

shm-key – for the purposes of inter-process communication, each shared memory segment must have a unique name. Further, since we are using System V shared memory, the name of each shared memory segment must be a unique *number*. This parameter is here to provide flexibility in the naming of shared memory segments. The first shared memory segment allocated by the proxy will be named with this parameter value. The second segment will increment the number and use that as the name, etc. This should help to avoid shared memory segment name clashes with other programs in the system. The default value of this parameter is 31337.

So, if we built the executable from the root directory of the project, one possible way to run the proxy would be as follows:

```
bin/proxy use-shm=1 shm-size=512
```

This will start the proxy on the default port, with use of four shared memory segments of half a kilobyte each.

client is the testing client executable used for exercising the HTTP server and the proxy server. In order to run, it requires a URL file, which is basically a plain text file with URLs that the client will request from the server. These URLs can have either host names or dotted IP addresses with them - the client can handle both kinds of URLs. The client supports the following options:

> *job-count* - the number of requests to make to the server. The default value is 1. Note that this is the total number of requests to be made, and this will cycle through the URLs in the URL file. So, for example, suppose this parameter is set to 100, and the URL file contains two URLs, then each URL will be requested 50 times.
>
> *thread-count* - the number of threads to spawn to make requests to the server at the same time. Note that these threads will handle as many requests as the job-count specifies, so, for example, if the job count is 100, and thread-count is 10, then likely each thread will make 10 requests to the server (though that's not necessarily the case, since for example if one of the workers is taking a long time to receive the response from the server, then the others will pick up the slack).
>
> *url-file* - the name of the file that contains the URLs to request. If no value is provided for this parameter, then the URLs will be read from the console.

So, if we have built the project from the root directory of the project, we can run the client as follows:

```
bin/client job-count=1000 thread-count=32 < urlfile
```

Note that there are multiple types of URLs that the client understands. First of all, the client can make requests directly to the HTTP server. To do that, we give it a regular URL, like these:

```
http://salo.cc.gatech.edu/file.html
http://130.207.114.229/file.html
```

Additionally, the client can make requests through a proxy. This is done simply through a different URL format, so that the client can make both direct and proxied requests in a single testing session. In order to issue a request through a proxy server, one simply needs to concatenate the proxy server URL with the requested URL, like so:

```
http://localhost:8080/http://salo.cc.gatech.edu:1337/file.html
```

In this case, the client will contact the proxy running on port 8080 on the local machine, and through it issue a request for file.html to the server running on port 1337 on Salo.

3 System Design

3.1 Semaphores

The project makes use of semaphores to synchronize shared memory access. The semaphores are wrapped in our own interface, in case there is ever the need to change underlying implementations (as there, indeed, was during the project). Currently, the project makes use of POSIX semaphores to achieve inter-process synchronization. The fact that this implementation worked actually came as something of a surprise. Indeed, online documentation indicates that one should use System V semaphores for IPC, as POSIX semaphores only work within a single process. However, the team's tests showed that to be simply not the case. At least Red Hat Linux GCC libraries, on which the project was primarily tested (Helsinki, Salo, and the Enterprise and States Labs), are perfectly happy synchronizing different processes (server and proxy) with POSIX semaphores. Whether this is due to faulty standard conformance of the semaphore libraries, or to outdated online documentation, the team is not entirely sure.

3.2 Shared Memory

Like semaphores, shared memory is wrapped into its own standard interface for the project, so that if there is need to change underlying implementations, it can easily be done. Additionally, the interface provides synchronized access to the data stored in shared memory through use of semaphores. Currently, the project uses a System V implementation of shared memory for Linux. The original implementation utilized POSIX shared memory interfaces, but that turned out to not be as portable as System V, and so was changed.

When shared memory is created through the project's own interface, defined in src/common/shared_memory.h, the user specifies the name of the shared memory segment and the desired data length. The library then allocates that much space plus a little more for a custom shared memory descriptor. The descriptor structure is put just before the data segment in shared memory and contains two pieces of data: the size of the data segment following it, which is useful when another process opens the same shared memory segment, so it can know how much it can write into the memory; the descriptor also contains the number of bytes written into shared memory on the last writing, which is useful for synchronized read/write access to the shared memory.

In addition to allocating the shared memory, the library also automatically creates two semaphores – can_read and can_write. Their names are derived from the name of the shared memory segment by appending the suffix _sr and _sw respectively.

When shared memory is opened, the library only needs the name of the shared memory segment. It will then automatically open the shared memory, and the associated semaphores for synchronized access.

In order to achieve synchronized memory access, the library makes use of three pieces of data – the number of bytes written to memory on the last write access, and two semaphores: can_read and can_write. When shared memory is created, can_read is blocked and can_write is signaled, so that the memory is writable but not generally readable. Actually, the library provides two shared memory read functions: a blocking and a non-blocking version. The blocking version will wait for the can_read semaphore to be signaled before reading, and the non-blocking version will just read from memory immediately. The non-blocking version is needed to support the server's shared memory chunk, which contains the port that the server is running on. It is only written once at server startup, but is read every time the proxy gets a local request. See more on this in the Server and Proxy design sections below.

When memory is written, the writing thread will block until the can_write semaphore is signaled. It will then put the data into shared memory, set the bytes_written data member to the number of bytes written into memory, and signal the can_read semaphore. When memory is read through a blocking call, the reading thread will block until the can_read semaphore is signaled, and will then read until all the data written into memory is read, set the bytes_written field to zero, and signal can_write.

This system ensures that if there are two threads, a reader and a writer, possibly in two different processes, each calling the read and the write function respectively many times over, the access to the shared memory will alternate between the reader and the writer. The data will be transferred without a loss from the writer to the reader, no matter how the scheduler decides to prioritize these threads.

3.3 Server

The server is implemented using the boss-worker architecture. Essentially, there is a single boss thread that spins indefinitely listening to connections. When a connection is established on the server's port, the boss thread will open up the socket to the client, and put it on the waiting socket queue.

The queue is synchronized such that we can have a blocking dequeue call. This feature is used to dispense pending client connections to worker threads. Worker threads are allocated and started at the start of the server. They then all call the pending client queue's dequeue function, and block until requests become available. Every time the boss thread pushes a new socket onto the queue, a condition is signaled and one of the waiting worker threads wakes up to handle the request. Once the request is handled, the worker comes back for another one from the queue.

The worker threads can handle two types of requests. One is the regular HTTP GET request. The request line has the standard HTTP 1.1 syntax:

```
GET SP Request-URI SP HTTP/1.1 CRLF
```

When such a request is received, the worker will open up the local file corresponding to the requested URI and send it back through the same socket through which the request was read. If the file can not be found, or if the user running the server has no permissions to read it, or if the request itself is malformed, the worker will send back the appropriate error message.

The second type of request that the worker thread can handle is a special local get request. It has similar request line syntax:

```
LOCAL_GET SP Request-URI SP Shared-Memory-ID SP HTTP/1.1 CRLF
```

Here, in addition to the requested URI, we provide a shared memory id. These kinds of requests will only come from our own proxy server, and will mean that the proxy and the server are running on the same machine. In this case, nothing will normally be sent through the socket. Instead, the server will attempt to open the shared memory location with the specified ID, and if successful, all the other communication with the proxy will be done through the shared memory location. From here on, the shared memory location is treated just like the socket, that is, the worker will try to open the file and if it fails, it will send an error response through the memory, otherwise it will send the file through.

Note we mentioned that the proxy ‘knows’ when it is running on the same machine as the server. In order to facilitate this knowledge, when the server starts, it will open up a shared memory location of its own (with the ID specified in a common header file), and write the port that it is running on into the memory. The proxy can then use this information to determine whether to issue a local or a regular get request.

One more consideration for the http and the proxy server was the clean-up procedure. One solution here was to catch termination signals and invoke the clean up procedures that way. However, the way the servers are implemented instead are that the boss thread is actually different from the main thread. While the boss thread is spinning indefinitely accepting connections, the main thread waits for console input, and upon receiving the letter ‘q’ terminates the boss and the worker threads and cleans up shared memory and other resources, and only then quits.

3.4 Proxy

The proxy server utilizes an architecture very similar to the http server: a single boss thread, with multiple worker threads blocking on a synchronized queue of pending requests. Indeed, the two servers use the same ‘dispatcher’ code, found in the common library.

The main difference between the http and the proxy server comes in how the worker threads handle the incoming requests. When a request comes in for the proxy, the first thing it does after parsing the http header is check whether the request came for the local machine. That is done by obtaining the socket address structure for the host requested and comparing it against the socket address structure for the host name of the machine the proxy is running on. This approach is better than simply comparing names, since it handles requests both for localhost, and for different names of the machine (it works for both just the machine name, and the fully qualified machine and domain name). However, this approach is not perfect. Suppose we are running our proxy server on legolas.cc.gatech.edu. Suppose also we used some sort of Internet registration service to have the name www.legolas.edu point to Legolas’ IP address. In this case, if the request comes in for the latter name, the proxy might not actually know that the request is coming in for the local machine, since depending on how the redirection service works, it might actually give us a different socket address structure.

If through comparing the socket address structures the proxy determines that the request has come in for a server running on the local machine, then it will try to open the shared memory location created by our http server. If the memory location cannot be opened, then it is clear the either the http server is not running, or somehow it is having trouble with shared memory (for example, the user has already exceeded the per-user shared memory segment limit), and the communication is done through the socket. If the shared memory location is opened, the proxy will then read the port number on which the server is running from it, and compare it with the port number for which the request came in. If the port numbers match, then we know that the request is local and we can use shared memory. Otherwise, the communication is done through the socket.

Once it is established whether or not we can use shared memory, the proxy will take slightly different actions depending on this outcome. If shared memory cannot be used, then the proxy issues a regular HTTP GET request through a socket to the http server and reads the result through the same socket, and channels it in to the client through the client socket.

If shared memory can be used, then it gets slightly more complicated. When the proxy server is started with shared memory enabled, it will allocate a certain number of shared memory locations into a synchronized queue of free shared memory locations. The worker thread that needs to use shared memory will get a segment from the queue (possibly blocking if the queue is empty, to wait for a shared memory location to be returned by one of the other workers). When the shared memory is obtained, the worker will send the special local get request to the http server through a socket, providing both the URI for the requested file and the name of the shared memory location that is now reserved for use in this particular transaction. The worker then keeps reading from the shared memory location until nothing more can be read, and channeling the output to the client through the socket. Once the transaction is complete, the worker returns the shared memory segment to the free segments queue, so that it can be reused for another transaction.

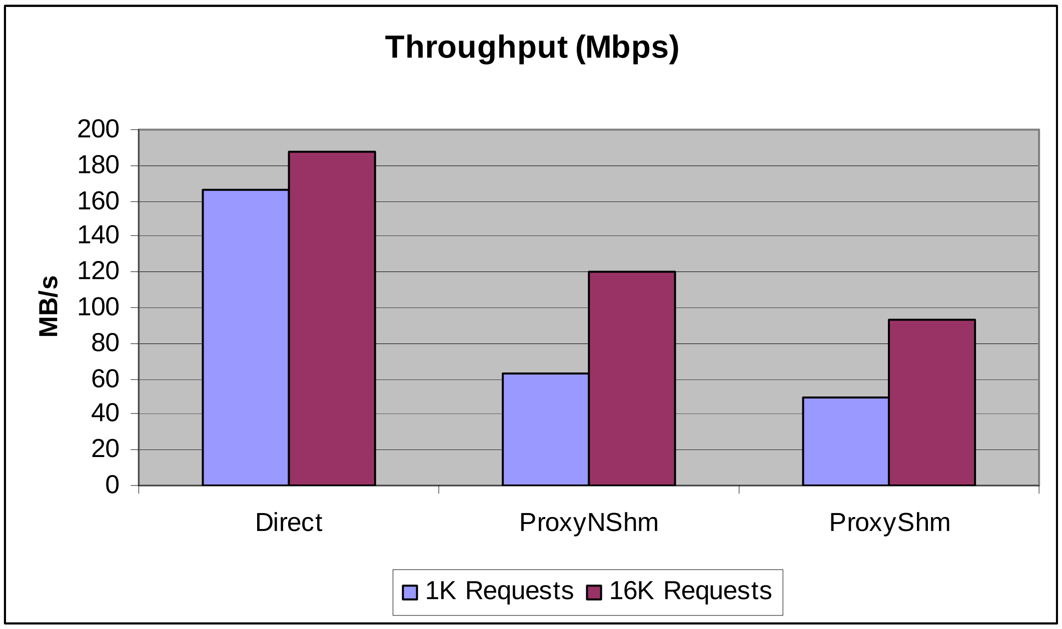
4 Tests

4.1 Test Setup

For this project, two types of tests were run. In both cases, the client had 16 worker threads, and the server and the proxy also had 16 worker threads apiece. The difference between the test sets was in the number of requests performed: 1000 and 16,000 request batches were run. All the tests were run on Aragorn at the Enterprise lab (a dual-core Red Hat machine). Both sets of tests were performed on the following three configurations: local requests directly to the server, local requests to the server through a local proxy without use of shared memory, and local requests to the server through a local proxy using shared memory. For the tests using shared memory, the proxy created 16 segments of shared memory of 2KB each. In addition, in order to eliminate the impact of caching on our timing, each set of tests was requesting files of varying sizes between 120 bytes and 2 MB.

4.2 Test Results

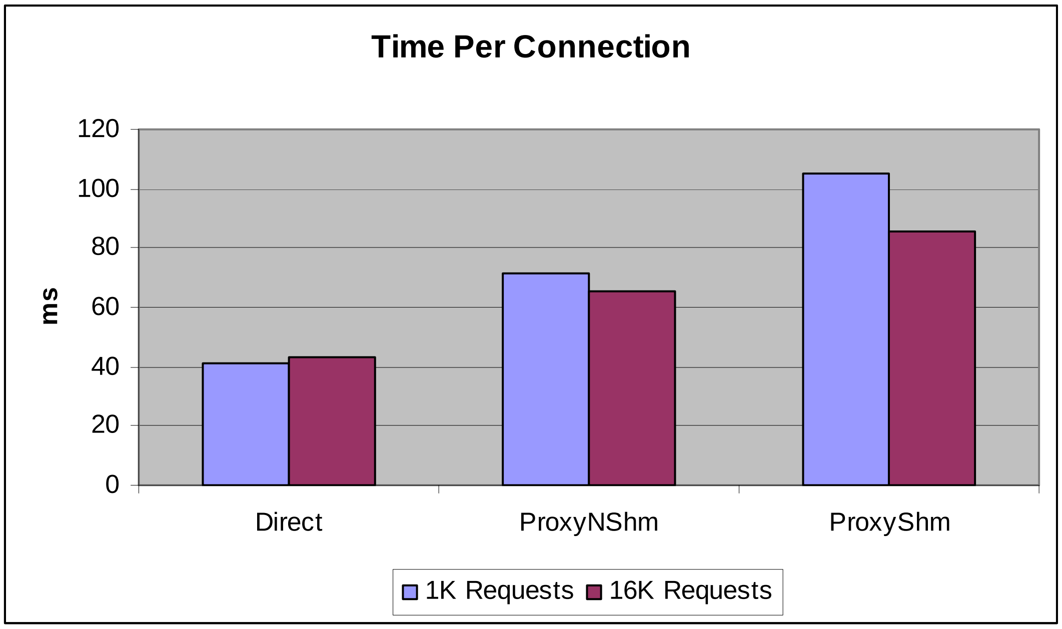
Following is the graph of throughput in megabytes per second for the tests that we ran:



The blue bars represent data for one thousand request runs and the red bars are for 16 thousand request runs. The vertical axis represents throughput in megabytes per second. Interestingly, the throughput drops as we go from using the proxy with sockets to using the proxy with shared memory. After investigating the code, several possible reasons for this were found. The primary cause of the slowdown is the proxy check for local requests; we first resolve the name of the requested host into a sockaddr structure, then make comparisons between it and the sockaddr structure for the local machine. This takes a relatively large amount of time. Next, in the proxy, we open the shared memory location started by the server so that we can read in the proper server port. This port and the one requested by the client must match before we transfer information through shared memory. Naturally, these two checks do not occur when shared memory is disabled through the command-line switch. Commenting these two checks out gives shared memory nearly equivalent performance to sockets. Finally, we use dynamic reallocation more often in shared memory than when using sockets, in order to produce the local get request for the server, which probably accounts for the remaining performance difference. We believe these are the factors that had the largest effect on the weak performance of the shared memory implementation, but there may be other possibilities, like the server's overhead for opening the shared memory and the associated semaphores, etc.

Overall, we believe that the loss in performance for shared memory over socket implementation is mostly due to latency issues, rather than bandwidth issues, otherwise, we wouldn't see an improvement in throughput for more jobs.

In addition to the throughput of the shared memory implementation being lower than that of the sockets, another strange result we obtained was latency.



The data in this graph are organized the same way as on the throughput graph, except here the vertical axis measures the average time, in milliseconds, per connection (which is measured from when the connection is established, through sending the request to the server, and up to when the server closes the connection upon completing the file transaction). The strange result here is that for the proxy, we had larger times per connection for one thousand requests than for 16 thousand requests.