Advanced Computer Network

Course Project Report
File Transfer Program

Introduction

This is a report for the file transfer program project for Advanced Computer Network course in Tsinghua University.

The project is to write a program to transfer files from one machine to another directly connected via an Ethernet cable. The experimental setup is as follows:

- Two servers with Ubuntu OS x64 with 3.1Ghz, 8GB memory and 1G NIC each.
- A repository is created in both server (~/Desktop/ServerFiles) and client (~/Desktop/ClientFiles) for file reception/transmission.
- IP address configuration: @Server 192.168.10.2, @Client 192.168.10.3.

The timing begins from the start of the program at client and stops when the program at client ends. Linux command named “time” will be used to measure the time taken. Final time is user time + sys time.

Language Chosen

The project give the choice between C, C++, Java and Python.

C++ is chosen because it allows memory management, which can be exploited to minimize potentially time-consuming operations like copying objects and to avoid garbage collection.

The C language has the same benefits (in a more extreme way) but we did not choose it because of the development time.

Architectures Tested

Over the course of the project, some different program architectures were tested. Initially we were oriented to use the UDP protocol, and code drafts had been completed. However, a late set informed us that a checksum (md5) would be realized at the end of the transfer to check the integrity of files. Following this, we abandoned the use of UDP in favor of TCP, we present here the TCP oriented architectures.

1. Asynchronous, Single-Client TCP

In this architecture, the server program sends the data asynchronously (the program does not block while waiting for the data to be transmitted completely).

In this architecture, one TCP connection per file is used, i.e. after a file has been sent completely, the client reconnects to the server to get the next file. This is necessary as the asynchronous operation of the networking library used (Boost Asio) is unable to stop reading exactly at the specified delimiter, which makes the data stream management problematic if the next file is to be sent right after the end of the previous file.

The result was quite satisfactory; time taken to transfer all the files during the test was about 27-28 seconds. However, we believe it can be further optimized as there are overheads of TCP 3-way handshake and slow start when the client program re-establishes connection with the server to retrieve the next file.

2. Asynchronous, Multi-Client TCP

It is similar to the previous architecture, except multiple concurrent connections between the client and the server are used. And read/write are divided by clients.

The result was rather disappointing, though not unexpected. The time taken to transfer all the files was about the same as the previous architecture. We believe it was because the bottleneck lies in the bandwidth of the link instead of the

processor, and thus parallelizing the work by sending multiple files at a time does not increase the throughput as the number of bytes received at a time by the client is still the same. Moreover, it also suffers the same drawbacks as the previous architecture.

3. IOStream

Boost iostream are based on the TCP class. It is an overlay abstraction, to make easier use of streams, sources handling, and filters. After testing, it became apparent that we have a performance loss for unnecessary functionality.

4. Synchronous, Single-Client TCP

In this architecture, the server program sends the data synchronously (the program blocks while waiting for the data to be transmitted completely).
In this architecture, one TCP connection is used for all the files in the directory, i.e. right after a file has been sent completely, the server sends the metadata (i.e. filename and size) of the next file into the TCP stream. This is to avoid the overhead of re-establishing a TCP connection (the 3-way handshake) and the TCP slow start.

Robustness and Optimizations

To ensure a good code quality and standardization, the code was compiled with many security and check flags (as you can see in the makefile).
Memory leak was also checked using Valgrind (http://valgrind.org/).
The program was carefully written such that potentially slow operations that may take up significant number clock cycles were minimized. For example, all constants are hard-coded, and the number of methods and variables are minimized to minimize lookups.
G++'s optimization flag -Ofast was chosen, after comparisons with -O2, -O3, -Os.
To optimize the read/write data on disk, each file is only read once and is pushed into the memory. This includes the huge file as well, because the size of the file is 4GB while the server/client have 8GB of memory.

The server was unthreaded because only one client needs to be managed at a time, and the server does not have to continue running after the transfer.

Work organization

We had a quick briefing every week to keep informed of the progress of the project.

The code was written and tested in a vm with a smaller configuration than the final test server. We have use Git file versioning system with Github to manage our code. The different architectures and research were distributed between us. And once the choice is made, efforts were joined in the final phase.

Conclusion

This project allowed us to discover and take control of the boost library, and become aware of the different mechanisms underlying the client-server connection.

The research work on bringing him a rewarding overview about the different ways to create a connection.