Proceedings of the 11th International Conference on Applied Informatics
Eger, Hungary, January 29–31, 2020, published at http://ceur-ws.org

Developing a High-Speed Connectionless File Transfer System with WASM Based Client

Robert Tornai, Dalma Kiss-Imre, Péter Fürjes-Benke, Zoltán Gál

University of Debrecen, Faculty of Informatics
tornai.robert@inf.unideb.hu
imre.dalma99@gmail.com
furjes.peter99@gmail.com
zgal@unideb.hu

Abstract

This paper describes an application named FFTM (Fast File Transfer Manager) that is based on Xinan Liu's Reliable File Transfer Protocol. The system consists of a server and a client program utilizing UDP connection. The aim is to transfer big files quickly by this program on busy network connections. We developed a *Java*-based minimum viable product relying on Xinan Liu's solution by adding control over chunk size. This article introduces the results of rewriting the modified software in C++ and the improvement of the client by a graphical user interface. Furthermore, we want to make this program available for as many platforms as we can. To achieve this goal we paid special attention to the WebAssembly programming language. Thanks to the generated WASM binary, our application can be used from a browser without installation.

Keywords: high-speed networking, high-performance computing, Internet, parallel communication, congestion control, traffic engineering, statistical analysis, scale independence.

MSC: 68M10, 68M12

1. Introduction

The need for handle big amount of data and make computation on them led to using supercomputers. This introduces new problems, e.g., transfer a huge amount of data between workstations and supercomputer nodes for processing. Not only the upload process is slow, but the result can be huge also to download. Even gigabit connections can slow down to a few megabit range in a busy network in real-life use cases having even packet loss [1], as we experienced this at file transfers forth and back with a server hosted in the Gyires supercomputer's data center [2].

Transmission Control Protocol (TCP) based solutions have a lot of problems with traffic control. Usually, the transfer rate fluctuates a lot, even having just a small extra data transfer on the network from other nodes. A User Datagram Protocol (UDP) based software handles big data transfers a way better. Albeit, they have a challenge of the feedback of the unsuccessfully transferred packages. By creating a UDP based high-speed file transfer system, the demand to quickly move huge data between supercomputer nodes and workstations could be satisfied.

The UFTP and UFTPD software pair mostly accomplish the needed features as transfer rate control, encryption or integration checking [3]. They cannot be used with a GUI and query the path on the server. Implementing a GUI for the UFTP software could be a solution for the server side path management and the Midnight Commander style two-pane operation. However, it would not be available in browsers, which is a basic requirement in our project. The whole source code of the two applications of UFTP and UFTPD should be modified to be able to compile for WebAssembly. Instead of this, we decided to write an own implementation designed for WASM from scratch, this way it can be run in modern browsers [4]. The chosen Qt system contained support for WebAssembly first as a tech preview in version 5.12 beta at the end of 2018.

Xinan Liu's Reliable File Transfer Protocol was chosen for the starting point of the development work, because beside its small code base it achieved the first place in CS2105 (Introduction to Computer Networks) Speed Contest AY15/16 Sem1.

Moreover, we take a glance at the possibility of the usage of the Stream Control Transmission Protocol (SCTP) [5]. SCTP is able to handle the data transfer and control feedback by utilizing both TCP and UDP protocols [6].

The structure of the paper is following: in chapter two is described the development environment of the fast file transfer application including Qt customization, SCTP support and Webassembly considerations. In chapter three reimplementation method in $C++$ of the fast file transfer application is presented. Performance features of the new product are given in chapter four. Possible continuation and future research and development targets are described in chapter five.

2. Developing environment

For the platform independent development the Qt 5.14.0 stable version was chosen [7]. The work was carried out on a Debian 10.2 workstation. Qt has a lot of

desktop operating system targets as Windows, Linux and macOS. Furthermore, it supports mobile equipment as Android and iOS [8]. In recent years one of the most interesting new platforms is WebAssembly [9]. Through this technology almost native speed program running is available from modern web browsers.

The chosen programming language is the highly portable $C{+}{+}11$. In Qt, the other natural choice could be QML. However, since it is a script language, it does not suite the efficiency needed for high-speed network transfer.

The server is designed to run in headless mode. The client is built for the necessary target system natively having a GUI. The WebAssembly client is lacking SCTP support right now. Qt contains 32bit WASM target at present which is enough for our purposes, in Qt's future plans they will support 64bit as well. By switching to Qt, it enabled us to use the SCTP protocol easily through its `QSctpSocket` and `QSctpServer` class. Although, we have a working SCTP file transfer system now, the performance of the TCP based FTP transfer and our SCTP and UDP based file transfer implementations need more investigation [10]. Different message sizes shall be examined to find an optimal value for later use as default. In the following development phase we will try to make SCTP work in WebAssembly also.

2.1. Customization of the Qt system

The research work started by testing the Qt network examples. We found many good solutions, but the most promising program was the `MultiStreamServer` and the `MultiStreamClient` pair utilizing the SCTP protocol [11]. Trying to build and run these sample programs in `QtCreator`, it yielded error messages because it could not found the `QSctpSocket` class. This was solved by rebuilding the Qt system by enabling the SCTP package for the Linux target. Looking for solution, it was suggested to build it statically, but Qt was slow to compile and later the building phase of the project was even slower. It was not effective enough at compilation (binary executables were about 400 MB in size). This binary target will be used for the distribution of the release version because it has no dependencies (see Figure 1). For further optimization some other arguments were also added beside omitting the static flag for testing dynamic binding. In this way compiling Qt became quicker and building with this target was also faster.

2.2. SCTP

After having compiled the whole Qt system to support SCTP [12], we could try the sample programs and investigate their features. `MultiStreamServer` and `MultiStreamClient` have three channels for three distinct features as `Movie`, `Time`, and `Chat`. This server-client pair was the base of our implementation.

A new channel was added for file transfer that is based on the `Chat` provider. It worked as expected with small files, but when we tried with bigger files, the performance was subpar. A nanosecond timer was added to the client and to the server to examine the packet sending and arrival rates. These timestamps are

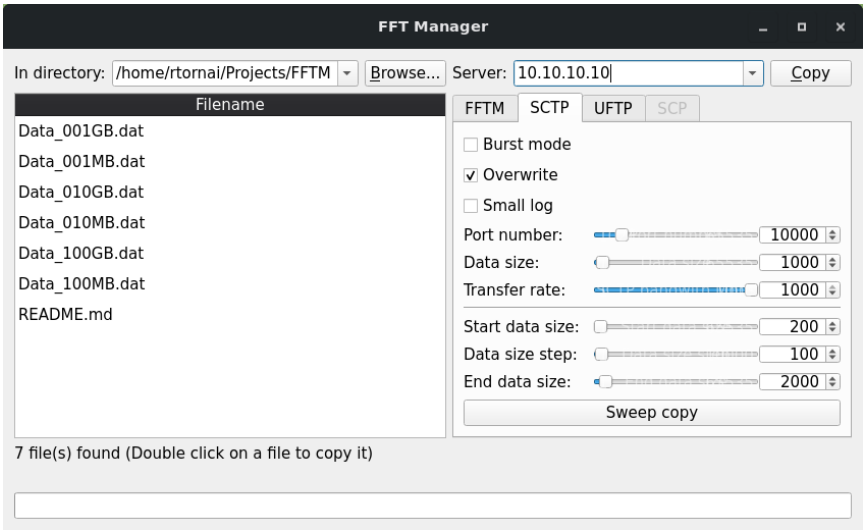

Figure 1: FFTM using SCTP

visible on the console, and they are saved in server and client log files for later processing. Using these logs we will optimize the algorithm.

2.3. File management in WebAssembly

The client was implemented in a way that it can be compiled in WebAssembly as well [13]. Reaching local resources shall be implemented in a different way. Since WebAssembly programs run in a stacked virtual machine, our program can only reach its own resources. In order to solve the problem, we need to redefine the Qt's own file access mechanism. One of the hardest challenges related to WebAssembly was to make it possible to load data files from local resources into the program. As our program runs on the WebAssembly stacked-machine, basically, we can only reach the virtual machine's filesystem. Our final solution is based on Morten Johan Sørvig's[1] example. It is implemented through a callback function mechanism. Basically, we have to upload the local files to the WebAssembly memory, and from there, the application can load in. At downloading the results of the supercomputer work, the user can download the file similarly onto his computer, but the data goes in the opposite direction. Right now, we have a test software for copying data files (see Figure 2). This will be transformed to a real file transfer client software by hiding the GUI elements needed for testing only, and it will be converted to a two pane commander style application.

[1]Morten Johan Sørvig solution is available here:
`https://github.com/msorvig/qt-webassembly-examples/tree/master/emscripten_localfiles`

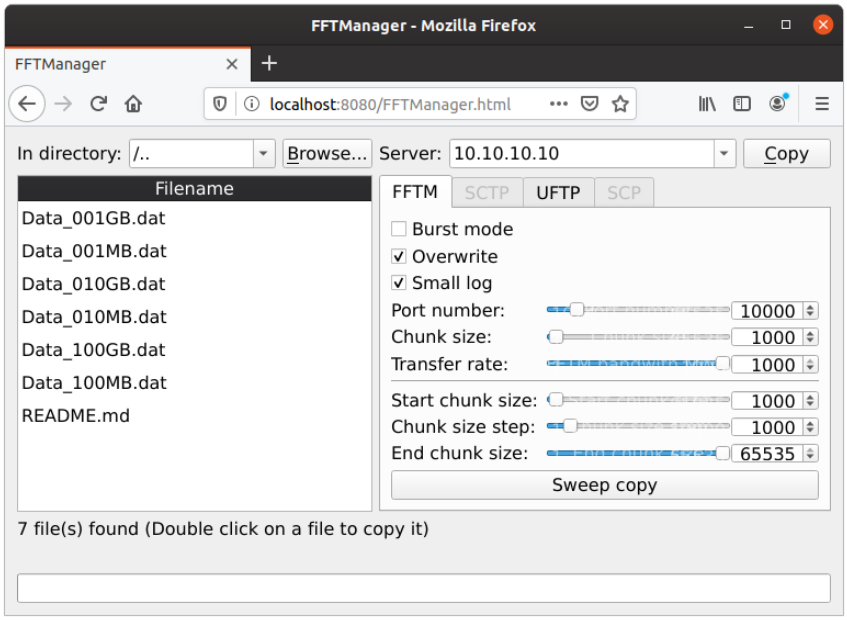

Figure 2: FFTM running in a modern web browser

3. Conclusions of *C++* reimplementation

The first solution to transfer files was using the standard FTP protocol which runs over TCP. For testing we used a gigabit network. In real life circumstances the speed fluctuated a lot in our throughout tests, and it was sustained around 40 Mbps during a lot of time. This led us to create our own file transfer implementation. The initial approach was to have a UDP data channel with a TCP control channel similar to SABUL [14] which lived on as UDT until it was abandoned in 2013 [15]. Solutions based on UDP, especially by adopting rate-based algorithms, give better performance than other alternatives according to Cosimo Anglano et al. work [16]. Later this led us to the conlusion to refine our software to use a UDP channel for the control messages also [17]. On lost of either the data packet or the acknowledge packet, a resend is needed. Instead of the usual 40 Mbps transfer rate of the FTP connection, we could reach a stable 800 Mbps with our UDP based FFTM software.

The main reason for the reimplementation was the fact that contrary to *Java*, *C++* programming language provides low-level access to the hardware. So our intention was to increase the application's performance. Qt's container classes are generally exception neutral. They pass any exception to the user while keeping their internal state valid [18]. Using try-catch blocks in *Java* had big resource requirement, so we replaced them with a more effective error handling solution. For the data transfer we used `QNetworkDatagram` and `QUdpSocket` classes. We

could measure that the CPU utilization of the *C++* code was less than the *Java* implementation's [19]. For *C++* the throughput reached 817 Mbps at chunk size of 1100 bytes, while *Java* produced 239 Mbps. At chunk size of 4000 bytes these values measured at 836 Mbps and 545 Mbps respectively. The former was a 3.42x while the latter a 1.53x speedup. The decreasement in the CPU load was 50% on average, which means a twofold acceleration (see Figure 3.). It is interesting that parallel with the speedup, the packet transmission success rate decreased from 92.6% of the *Java* software to 83.3% of the *C++* program.

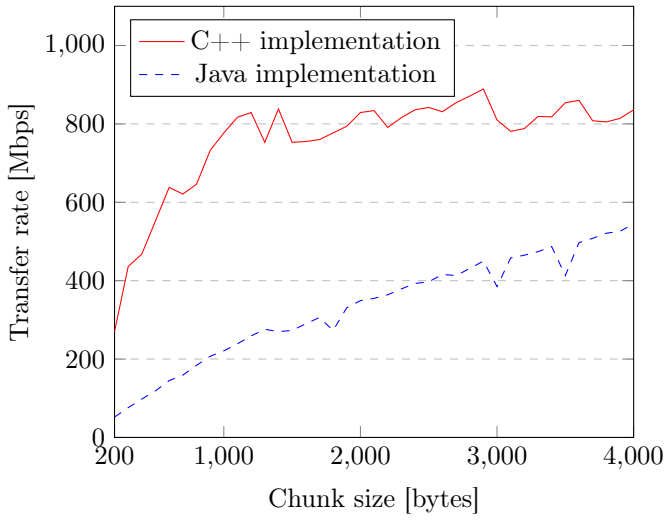

Figure 3: Throughput comparison of the implementations with different chunk sizes

The test server was a Cisco UCS C240 M5 having gigabit Ethernet connection, connected to the Gyires supercomputer. The applied virtual machine runs under VMWare's vSphere 6.7 having 20 GB memory and 8 cores of an Intel® Xeon® Gold 6130 CPU @ 2.10 GHz. The test machine has an Intel® Core™ i7 CPU 920 @ 2.67 GHz with 18 GB memory having gigabit Ethernet connection, connected to the academic Internet network.

The testing task was to transfer 60 TB data from desktop workstations to the Gyires supercomputer for processing and download the results of a smaller magnitude. This amount of data will be a typical usage in the future. At full speed on gigabit connection this means minimum 5.79 days. FTP transfers slowed down to the 40 Mbps range which could mean 138.89 days. Understandably, the realized time of the transfer is between of these two extreme values.

4. Results

Our first *Java* implementation had a UDP data channel and a TCP control channel. It was enhanced to a high-speed connectionless file transfer system by using UDP control channel relying on Xinan Liu's work. Later, reimplementing it in C++ the software became more suitable for high-speed file transfer to supercomputer nodes.

As low as 40 Mbps FTP transfers were sped up to 800 Mbps on busy connections. Using WebAssembly, the client runs in browsers. SCTP was also added to the desktop client. Reimplementing in *C++* yields a twofold speedup in execution.

5. Future work

We will investigate the effect of jumbo frames on the transfer rate with the different algorithms. MTU sizes up to 9000 bytes will be tested. Our hope is that it helps to get closer to 1000 Mbps. SCTP is disabled for the WASM target in the Qt system at this time. Qt needs modification at the configuration process. We will try to make it work in the future. The server shall be made multithreaded to service more clients efficiently. Broken transfers will be enabled to resume later. The client will be modified to be a two pane file transfer and manager software.

Acknowledgements. This paper was supported by the FIKP-20428-3/2018/FEKUTSTRAT project of the University of Debrecen, Hungary and by the QoS-HPC-IoT Laboratory. This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.

References

[1] Sawashima, H., Hori, Y., Sunahara, H., Characteristics of UDP Packet Loss: Effect of TCP Traffic, in *Proceeding of the 7th Annual Conference of the Internet Society*, Kuala Lumpur, Malaysia (June 1997), URL `https://web.archive.org/web/20160103125117/https:/www.isoc.org/inet97/proceedings/F3/F3_1.HTM`.

[2] Gyires supercomputer (May 8, 2020), URL `https://hpc.unideb.hu/hu/node/219`.

[3] Schuller, B., Pohlmann, T., UFTP: High-Performance Data Transfer for UNICORE, in M. Romberg, P. Bała, R. Müller-Pfefferkorn, D. Mallmann, editors, *7th UNICORE Summit 2011 Proceedings*, vol. IAS Series 9, Forschungszentrum Jülich GmbH, Toruń, Poland (July 2011), ISSN 1868-8489, pp. 135–142, URL `https://core.ac.uk/download/pdf/34995345.pdf#page=144`.

[4] Rourke, M., *Learn WebAssembly: Build web applications with native performance using Wasm and C/C++*, Packt Publishing Ltd., Birmingham, England, 1st edn. (September 24, 2018).

[5] Gn, V., *Multimedia Streaming in MANETs using SCTP*, LAP LAMBERT Academic Publishing, paperback edn. (2019).

[6] Khatri, S., *SCTP Performance Improvement Based on: Adaptive Retransmission Time-Out Adjustment*, LAP LAMBERT Academic Publishing, paperback edn. (2012).

[7] Lazar, G., Penea, R., *Mastering Qt 5: Create stunning cross-platform applications*, Packt Publishing Ltd., Birmingham, England (December 15, 2016).

[8] Eng, L. Z., *Qt5 C++ GUI Programming Cookbook: Practical recipes for building cross-platform GUI applications, widgets, and animations with Qt 5*, Packt Publishing Ltd., Birmingham, England, 2nd edn. (March 27, 2019).

[9] Qt for WebAssembly (December 16, 2019), URL `https://wiki.qt.io/Qt_for_WebAssembly`.

[10] Madhuri, D., Reddy, P. C., Performance comparison of TCP, UDP and SCTP in a wired network, in *2016 International Conference on Communication and Electronics Systems (ICCES)*, Coimbatore, India (October 2016), ISBN 978-1-5090-1066-0, pp. 1–6, doi:10.1109/CESYS.2016.7889934.

[11] Leung, V. C., Ribeiro, E. P., Wagner, A., Iyengar, J., *Multihomed Communication with SCTP (Stream Control Transmission Protocol)*, CRC Press Taylor & Francis Group, 1st edn. (2012).

[12] Stream Control Transmission Protocol, URL `https://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol`, the page was last edited on 29 December 2019.

[13] Gallant, G., *WebAssembly in Action*, Manning Publications Co., Shelter Island, New York, 1st edn. (December 7, 2019).

[14] Gu, Y., Grossman, R., SABUL: A Transport Protocol for Grid Computing, *Journal of Grid Computing*, vol. 1(4) (Dec 2003):pp. 377–386, ISSN 1572-9184, doi:10.1023/B:GRID.0000037553.18581.3b, URL `https://doi.org/10.1023/B:GRID.0000037553.18581.3b`.

[15] UDP-based Data Transfer Protocol (UDT), URL `https://udt.sourceforge.io/`.

[16] Anglano, C., Canonico, M., A Comparative Evaluation of High-Performance File Transfer Systems for Data-intensive Grid Applications, *13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises* (June 2004):pp. 283–288.

[17] Liu, X., ReliableFileTransferProtocol (October 30, 2015), URL `https://github.com/xinan/ReliableFileTransferProtocol/tree/master/src`.

[18] Exception Safety, URL `https://doc.qt.io/qt-5/exceptionsafety.html`, visited on 2020-01-28.

[19] Java performance, URL `https://en.wikipedia.org/wiki/Java_performance#cite_note-43`, the page was last edited on 7 November 2019.