Report: File Transfer

Wu, Wei, 2012220748
Kölker, Stephan, 2015403217

December 14, 2015

1 Source Code and Algorithms

1.1 Language and Libraries

We implemented the file transfer program by using Java without any external libraries. The socket connection was built using the `java.net` package.

1.2 Program Architecture

The core classes of our program consists of four different classes: `FileTransferClient` and `FileReceiveHandler` on the client side and `FileTransferServer` and `FileSendHandler` on the server side. `FileTransferClient` and `FileTransferServer` create the socket connection, a thread pool with multiple threads and hand the threads as well as the file to be transferred to the executing thread.
To transfer a file, we use `FileSendHandler` and `FileReceiveHandler`. They implement both the interface `Runnable`, so they can be executed in a thread. We first transfer the full file path, then the size of the file to the sender. After that, the actual file content follows. Several chunks (size: about 10KB) will be read and sent iteratively since the system probably cannot hold the whole file into memory at once. On the receiver's side, we also read the file in the same chunks and write them to file.
For each file we use a single TCP connection to achieve the highest possible concurrency. But as the number of threads increases, we will end up with a bottleneck because of the thousands of tiny files. To avoid a bottleneck in computing power and bandwidth, we create a zip folder of the folder "thousands_of_tiny_files" and transfer this via an own TCP connection. On the other side, the zip folder is unzipped and the actual files are stored to the hard disk.
Besides the above-mentioned classes, we use some helper classes: `Config` to read from the configuration file, `DirectoryManager` to create directory paths recursively or `ZipUtil` that we use to "zip" a directory.
The main class is `Main`. Starting the program takes one parameter: "server" or "client". All settings are defined in `config.properties` in the `resources` folder.

1.3 Program Sequence

The following sequence diagram illustrates the optimal program flow without any errors. In the development of this application, we focused on the file transmission and transmis-

sion possibilities to enhance the throughput. So, we mostly ignored error handling. In case of errors, the application will directly throw java exceptions.

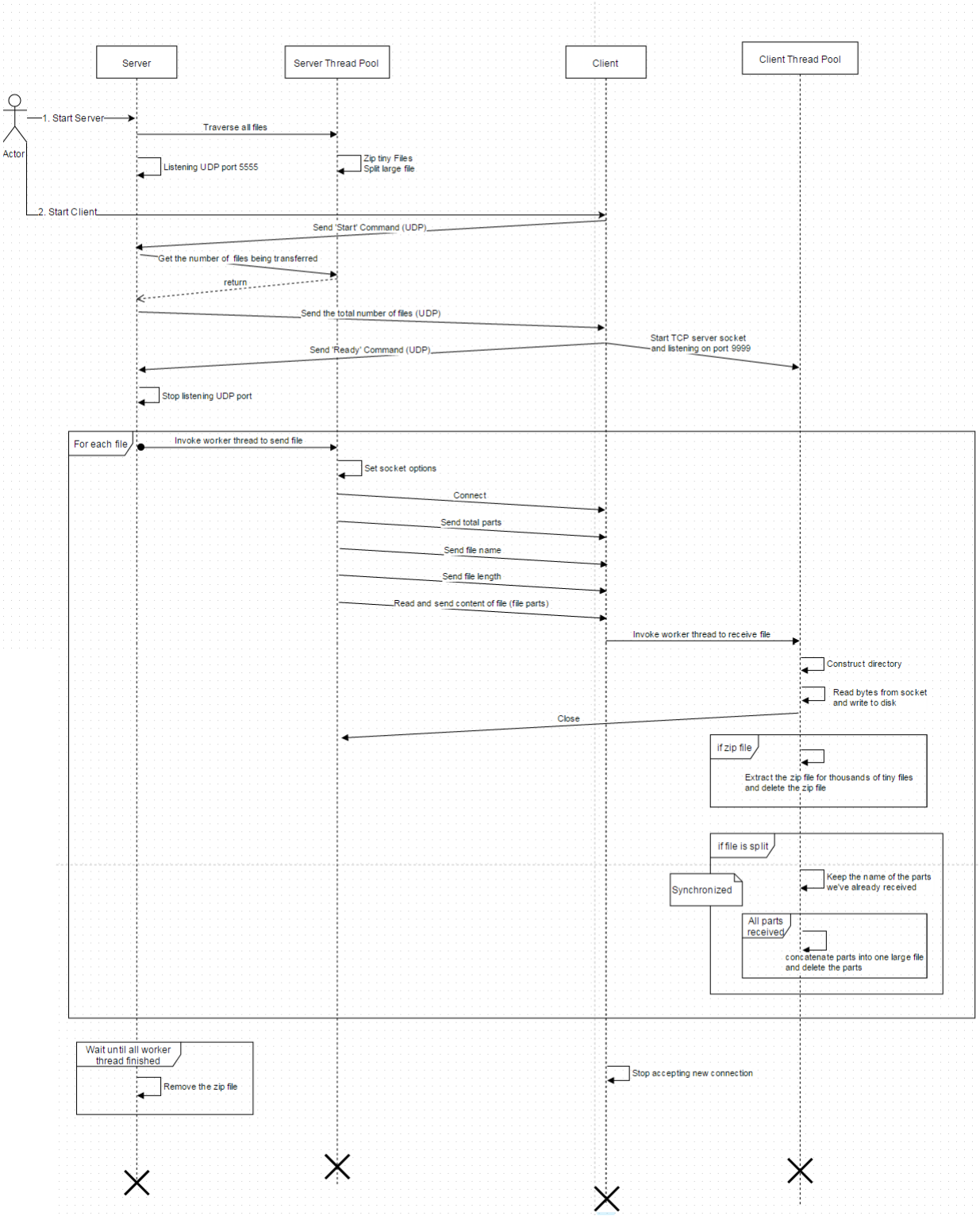

Figure 1: Program flow without errors

1.4 Alternatives and Decisions

During the development process, we tested various alternative classes whether they perform better than the above-mentioned classes.
The very first decision was: C++ or Java? Although C++ is known to be much faster and more efficient than Java, it also depends on the programmer and on the optimization possibilities. We are convinced that C++ does not offer much optimization possibilities in this small application and hence, the run-time of the programs in both languages should be more or less equal. We also decided for Java because one of us was not very familiar with programming in C++.
As an alternative to the `java.net` socket classes, the channel classes in the `java.nio.channels` package provide a more low-level approach in Java to perform IO operations. It provides various file- and socket-related classes. But in some speed tests, we could not determine any time difference between both method and hence, we decided for the classical IO classes.
The transmission of the largest file consumes the most transmission time. To avoid this bottleneck, it would be useful to split this file into several parts (e.g. 4 parts) and transfer them independently. The problem in this alternative is the file reconstruction at the receiver's side. When using the same file name, we can determine which parts belong together and with an additional byte that determines their order, even ordering would not be a problem. But writing concurrently from several threads into one file is much more complicated: Since every file part has its own offset and length, we must leave this offset free in the file. Current file systems do not support leaving x bytes free and then write y bytes. A possible solution would be first to write x zeros and then y bytes. This is very ineffective.
Another solution is to store all parts whose predecessor is not completely arrived yet in the memory and write them to disk as soon as their predecessor was written to disk. But this maybe could cause memory overflows. If for example the first file part arrives at last, the whole file has to be stored in the memory. This is the reason why we did not implemented it.
Yet another solution is to store the file parts as separate files on the disk and then concat them together after finishing writing all file parts. This approach is very prone to slow disk read and write time. Even if we merge the files together by appending all other file contents to one file, we will rewrite at least 50% of the file content to the disk, which is a very ineffective. The more file parts we choose, the higher this amount becomes. Our tests have shown that splitting the file into two parts is much slower (approximately 55s on the test system) than transferring it at a whole (approximately 38s on the same test system).
Another try was to implement UDP-based transmission for the small files. Since UDP transmissions have less overhead than TCP, the transmission would be much faster and in the test system, we have a closed world without any collisions and a very small transfer line. Nevertheless, the correctness of the transferred data is important and using UDP without ensuring the correctness is nothing but a game of luck. Of course, one could simply use a hash method as a checksum of every message make sure, the data is correct. But we decided against this approach, because even with a checksum the data could still be false.

2 Enhancements

We set the socket buffer size of both, sender and receiver, to 62 KBytes. The reason we chose this size is that we try to minimize the number of packets. We wanted to choose a buffer size of about 64 KBytes. Given a MTU of 1500 Bytes, a TCP packet has $1500-20-20 = 1460$ Bytes of payload. Because of IP fragmentation, only $\lfloor 1460/8 \rfloor * 8 = 1456$ Bytes can be transmitted. $64000/1456 \approx 43.96$ and $43 * 1456 = 62608 \approx 62 KBytes$. So, in theory we should have an average packet size of near 1456 bytes which is the payload of the packets. Besides, we set the disk buffer size (the buffer containing the data that is written do disk) to $200 * socketbuffer = 12400000$ bytes. We hope, this will improve our application throughput in the test system.
Moreover, we turn of Nagle's algorithm to make sure, the data is immediately sent after flushing it. We also set SO linger to an interval of 60.
Furthermore, we tried to enhance the network's MTU, but did not found a way to manipulate this using Java (except by calling a bash script). Therefore, we left this possibility out.
It is also possible to adapt the number of threads in our system. We decided to use two threads. The more threads we use, the more overhead we gain through context switching. Moreover, the total CPU processing time of our application is divided between all threads. So, we must find the optimal number of threads that minimizes the transmission time. We identified the transfer of the largest file as the main bottleneck on the transmission. When we transfer the largest file, we can simultaneously transfer all other data and that transmission will still be faster than the transmission of the largest file. Since we do not split the large file, it makes no sense to increase the number of threads.