ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Semester Thesis

Aggressive TCP

Serkan Bozyigit
bserkan@student.ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Thomas Locher, Stefan Schmid

Department of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
August 20, 2007

Contents

Chapter 1

Introduction

The goal of this thesis was to study a transport protocol in order to make the BitTorrent client "BitThief" presented in [4] more selfish. In particular we wanted to find out if it is possible for two collaborating BitThief clients to exchange data in a faster manner than with a regular TCP connection.
First of all, TCP was examined to identify some weaknesses on which could be worked on. One of them was found to be the "slow recovery" occuring during the process of a multiple file download. Having started an n-files download at some point one file will eventually be finished, but contrary to intuition the remaining $n - 1$ files will not immediately be assigned the released bandwidth, they will rather get it with a delay of a few RTTs which, depending on the connection quality, might be quite large.
When looking at the maximization of throughput TCP's congestion control mechanism and the acknowledgements needed for its proper function might be regarded as another "weakness". Data "overhead" is probably the better word for the second weakness and consequently, with our goal in mind, it should be reduced as much as possible.

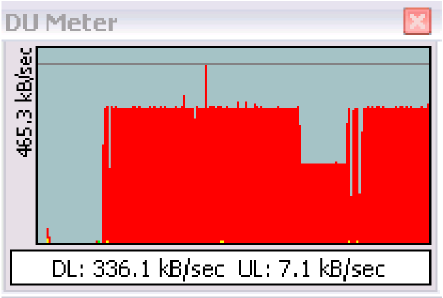
Second, a protocol had to be conceived which was more aggressive in the way that it did not respect any congestion in the network and implicitly did not show the above mentioned weaknesses. Both weaknesses are tackled by the fact that UDP is used as an underlying transport protocol for the conceived protocol named "sTCP". UDP's missing congestion avoidance mechanism could enable sTCP to achieve a higher throughput in the ideal case or in the worst overwhelm the forwarding and/or receiving nodes with the transmitted volume.

Chapter 2

TCP Examined

2.1 Slow Recovery

While surfing the Internet the user is downloading many different files, be it consciously or unconsciously. To simplify things a bit, let us have a look at the case where the user starts to download two sufficiently large files at the same time by clicking on according links in his browser. We are considering this scenario, because downloads from a web server are usually done using TCP[1]. “Sufficiently large” means TCP has a chance to find its maximum of throughput, this of course depends on the connection bandwidth, RTT[2], etc. and, of course, there is no other process downloading in the background. The connections of both files are now sharing more or less the same amount of bandwidth, i.e. half of the *full* bandwidth. At some point one of the files’ download terminates. What one would expect is that the second file’s download, that is still going on, gets the full bandwidth. If you take a look at the throughput display of your browser you see that this is not the case and TCP just speeds up very slowly. Let us call this behavior “*TCP’s slow recovery*”. As an example, which is shown in Figure 2.1, two files were downloaded from the same server and the throughput was captured with the DUMeter[3] utility. The obvious performance drop was caused at the termination of a file’s download. Again, instead of recovering rapidly the throughput stays at the same value for quite a while until it reaches the maximum throughput again.



DL: 336.1 kB/sec UL: 7.1 kB/sec

Figure 2.1: TCP’s slow recovery

What is the difference between the start of a connection and the above mentioned incident? In both cases the throughput has to be driven to a maximum, but they do not behave the same way.

TCP has a built-in mechanism called “slow start” and it has two distinct phases: the

[1] Transmission Control Protocol

[2] round-trip time

[3] http://www.dumeter.com/

"exponential growth" and the "linear growth" phase. The growth refers to the TCP congestion window, which increases exponentially until either an acknowledgement gets lost or a predetermined threshold value is reached[4].

If a loss occurs TCP assumes that it happened due to congestion in the network and therefore reduces the load. Once one of these events appear, TCP enters the linear growth phase where the congestion window increases linearly until an acknowledgement drops again.

Hence in the start phase of a download TCP reaches its maximum in a few RTTs, but once it has switched to the linear phase, it has its difficulties to elevate the throughput of a possible "half-used" connection and that is why we observe the slow recovery.

How the parameters for the growth- and stepping-back-rate of the algorithm are set highly depends on the variant of TCP that is used. Since there is no way for the algorithm to distinguish losses due to congestion from losses caused by other sources, this behavior can have enormous consequences for certain networks such as wireless networks. These are (more) susceptible to random packet loss due to signal attenuation etc. and thus might have huge performance reductions since each time a packet is lost the congestion window is being reduced and consequently the performance suffers.

Some TCP variants were specially conceived for wireless networks, for example *TCP Westwood*[5] or *TCP Veno*[5]. The latter tries to guess if the occured loss was due to a random packet error or a congestion loss by considering a special threshold. Since this is beyond the scope of this semester thesis, the interested reader might deepen his/her knowledge in the corresponding literature.

2.2 Second Working Point

In order to come up with a protocol that has a higher throughput we will first have a look at a characteristic of TCP.

The *Slow Recovery* discussed in the previous section only appears due to the built-in congestion control and congestion avoidance mechansim of TCP. An Internet without this mechanism would not work properly and we would not have any reliable and stable network. In such a case, no protocol intrinsic property would prevent the sender from overwhelming the receiver and/or the nodes in between. This issue is tackled by the sender by constantly evaluating the acknowledgements returned by the receiver. The evaluation results in an estimation of the congestion in the network and helps the sender to accordingly adjust his sending speed. Obviously these acknowledgements are crucial to the mechanism and have extra to be added to the raw data that has to be sent. From a throughput point of view an overhead has been introduced with this mechanism, which also means that we have found a "point of attack".

The protocol presented in Chapter 3 will try to avoid this overhead as much as possible by using UDP and a little bit of optimism[6], but of course some kind of overhead *has* to be present, since we want to be sure to receive the file without any missing segments.

[4] http://en.wikipedia.org/wiki/Slow-start

[5] http://linuxgazette.net/135/pfeiffer.html

[6] We hope that the file will be transmitted on the first attempt and any possible reparations will be undertaken after this attempt, contrary to regular TCP

Chapter 3

The Protocol

3.1 From regular TCP to a self-made TCP

TCP is widely used in the Internet and among others also for data transfer in the BitTorrent network. Every single peer of that network uses a client and each client on its part uses TCP to send data packets to many other users.

To repair TCP's slow recovery we can not simply build an additional layer on it that manipulates the data, because the issue is inherent to TCP's algorithms and we would not get rid of the flaw by doing so. One possible way to tackle the issue would consist of changing TCP itself. An obstacle that would lie in front of us is the fact that TCP is implemented in the operating system. Besides the difficulties of understanding and editing corresponding code in certain operating systems or even installing a client that changes OS code, there is a great chance that the alteration of TCP might incur unforeseeable side-effects, which would affect other applications. Of course before deploying such a protocol on a broad audience one would need to perform a thorough analysis. Another approach will be presented in this thesis because the previous approach would presumably imply an effort which would go beyond the scope of this semester thesis.

Following the initial goal, to make BitThief more selfish in some way, and given the hurdles above, a protocol was to be made that is situated between the transport layer (without using TCP) and the application, i.e. BitThief. Since the remainder of this thesis is about a self-made TCP-like protocol it is therefore referred to as "sTCP".

A good protocol candidate to build upon is UDP[1]. It does not have any of TCP's nice features like reliability or ordering of data segments. But what it does is, it serves as a underlying transporter of data. Consequently a custom-made protocol can be packed into UDP packets with the properties the user wishes to have. This way reliability and as well as control of order of the sent and received data can be guaranteed. The main reason why a custom-made protocol was needed at all was TCP's slow recovery. It only occurs because TCP resides in the *linear* growth phase and cannot find the maximum in a fast manner. sTCP addresses this issue by ignoring any possible congestion (there is no congestion window or the like), meaning the sender of a file will transmit as fast as it can, possibly overwhelming the receiver or the nodes in between or conversely, the transmission might even be faster in an optimal situation compared to a TCP transmission. The ignoring of congestion allows us to transmit with full bandwidth while in the same position the TCP connection would still try to recover. Hence resulting in a more aggressive TCP-like protocol.

[1] User Datagram Protocol

3.2 Description

There are at least two parties involved in a peer-to-peer network: a sender and a receiver. sTCP assumes that the application which is using the protocol provides the file(s) and the IP address to which these have to be sent.
First, both the sender and the receiver are in the first phase out of two: the initialization. The receiver is completely unaware of what and how much it is receiving, so the sender's first step is to extract this information and put it into an *initialization packet.* The receiver, on the other hand, is listening on a well known port for this packet to arrive and as soon as there is a proper *initalization packet* coming in, it forks a seperate thread that will read the following transmitted data. The forking of a thread enables the receiver to listen concurrently for further file transfers while it is already receiving the previous file. If the receiver was able to understand the initalization message it sends back an *initialization acknowledgement* so that both can enter the second phase of the protocol.
As soon as the sender gets the *initalization acknowledgement* it begins to transmit the file. The UDP packets in transit consist, among other things, of a payload that is entirely filled with sTCP's data[2]. The sending part of the protocol is very optimistic and assumes that all packets it sends will be received in one flow and that is also why it is "rushing" through the file. "Rushing", because there are no acknowledgement messages it has to wait for. Arrived at the end of the file it sends a *"through with the file packet"* which tells the receiver *the sender* has gone through the file and is finished with sending for now.
If the receiver got all packets in order in one flow, we are almost finished now with the whole procedure, but that is rarely the case since we are using unreliable UDP. Each *data packet,* besides the data itself, consists of a packet identifying byte and start position byte number which determine the payload's position in the file. The actual length of the raw data can be calculated by looking at the UDP packet size and subtracting preceding bytes. With this information the receiver can store the data and keep track of which data chunks are missing. Once it receives the sender's *"through with the file packet"* it assembles the positions of the missing bytes in a list and transfers this list to the sender, who is either waiting for this list or a *"final acknowledgement packet"* to arrive. If the sender receives the list the whole procedure of the second phase starts all over again except that the sender processes the data chunks from the list instead of the whole file.
Once the receiver got all bits and bytes it finally transmits the *"final acknowledgement packet".* The protocol's sequence diagram is shown in Figure 3.1.

Why is this list-based "repair strategy" used? In an earlier version of the protocol there was a third phase, where the reparation took place. The receiver still had his missing-bytes-list but it used to request each missing data chunk individually. A coarse overview of the steps that had to be gone through would be: sending *one* request by the file receiver, reading *one* request by the file sender, reading the corresponding data from harddrive or main memory, sending back of the requested data, and rechecking of data by the receiver. Because these steps depend on each other they cannot be done concurrently. The little delays introduced by each step finally add up and deposit themselves in a bad overall throughput in this phase.
Therefore the more sophisticated way of repairing the missing parts was to request the data in a more compact form by gathering the missing-parts information and sending them all *at once.* Like this, the protocol was reduced from three to two phases, it got simplified and code had been reused.

[2] UDP has a *length* field in its header which specifies the theoretical size of the whole Datagram. It is given by a 16-bit number which amounts to $2^{16} - 1 = 65535$ Bytes, inculding the header with 8 Bytes. The IP packet has a maximum of 65535 Bytes as well from which we also have to substract its header of 20 Bytes, finally giving us an effective payload size of $65535 - 8 - 20 = 65507$ Bytes.

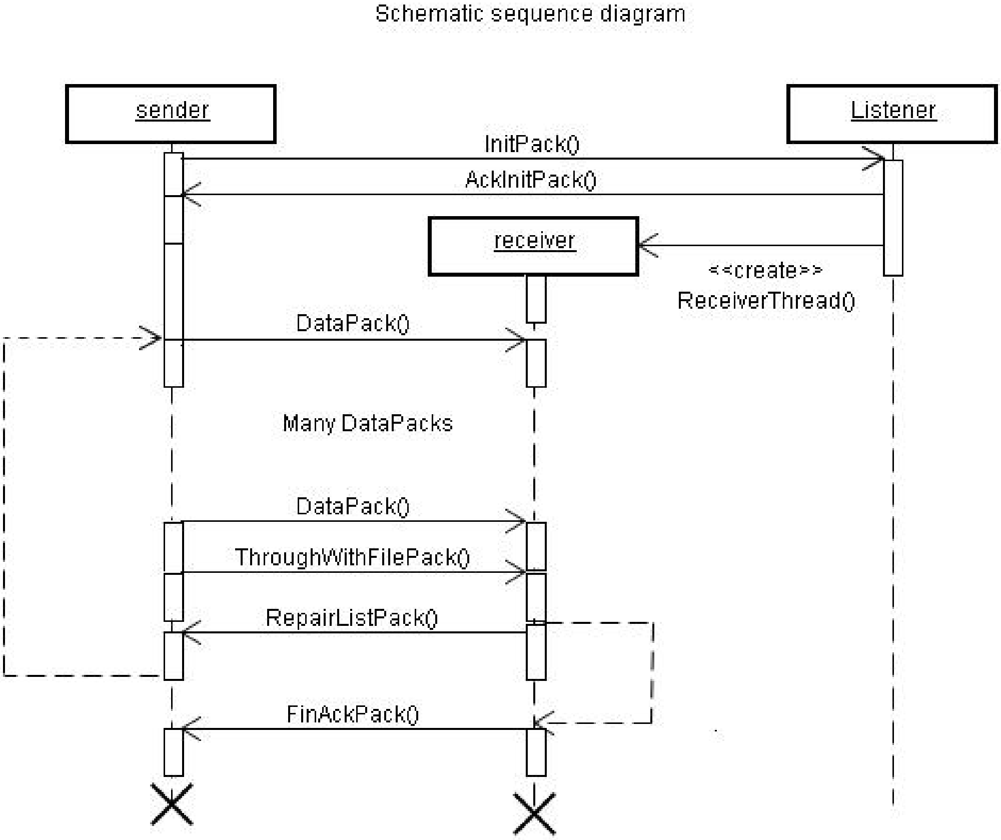


Figure 3.1: Protocol's Sequence Diagram

3.3 Packets

The following section is dedicated to the cornerstones of every protocol: the packets. See the Appendix on Page 26 for the exact format of each packet.
Additionally to the described fields in each packet below there is always one byte that identifies the packet as such in order to distinguish packets from one another.

Ping Packet (PingPacket):
There are seven different kinds of packets introduced in this protocol. In Figure 3.1 you can only see six of them in action, the missing one is the "Ping Packet". This packet is used at regular intervals throughout the second phase of sTCP and is needed to estimate the RTT between the peers which in turn is needed to compute the timeouts. The Ping Packet is sent every *second*[3] by the receiver and as soon as the sender receives the ping it responds with a "pong". If several ping messages stay unanswered the file receiver assumes that the file sender timed out and it will not get any further data packets, therefore it cancels the unfinished file transfer. On the sender's side there is only a "static" timeout used, if the receiver does not respond within a specific amout of time (15 seconds) after the last data packet's transmission it also assumes a timeout on the receivers part and terminates the file transfer.
In general it was rather difficult to find a good metric for the estimation of the RTT and hence calculating the timeouts out of it. If we used the RTT directly from the measurement we would get a very fluctuant estimation of the timeout. Since the Internet

[3] There is no specific reason why exactly *one* second has been chosen

and its connections are very dynamic the receiver would detect an ill-founded timeout in case of a small RTT followed by a relatively large one. But the following formula that I rediscovered in the "Vernetzte Systeme" lecture's slides seemed to be pretty reasonable:

$$EstimatedRTT = (1 - \alpha) \cdot EstimatedRTT + \alpha \cdot SampleRTT \tag{3.1}$$

With this equation, where α is 0.125 and $SampleRTT$ is the latest RTT measurement, an *exponential weighted moving average* is achieved and we end up with a "smoother" RTT estimation.
For the timeouts a similar formula was used which was as well taken from the above mentioned slides:

$$Timeout = EstimatedRTT + 4 \cdot Deviation \tag{3.2}$$

$$Deviation = (1 - \beta) \cdot Deviation + \beta \cdot |SampleRTT - EstimatedRTT|$$

This formula, with β being 0.125, also makes sure that if we have large variations in the $EstimatedRTT$ we also get a larger *safety margin*, because we do not want the file transfer to be aborted too early.
In an earlier version of sTCP both peers used to play a continuous "ping-pong". As soon as one of them received a ping packet it immediately returned a pong and vice versa. Turns out this high frequency of RTT estimation is way too often and consumes too much of the cpu time resulting in a very bad throughput.

Initialization Packet (InitPacket):
The receiver does not know much about the file it is going to receive, so the sender uses this packet in the beginning of a file transfer to inform the receiver about file name and size.

Initialization Acknowledgement Packet (AckInitPacket):
As soon as the receiver got the initialization packet it has to tell the sender by sending an initialization acknowledgement that it is ready to receive the file.

Data Packet (DataPacket):
This is the packet that is used the most. After the initialization the sender splits the file into many segments and sends them in such packets to the receiver. Besides the raw data there is also the starting position within the original file included.

Through With File Packet (ThroughWithFilePacket):
After many Data Packets the sender is finally finished with his job so far, and tells this the receiver by sending him a ThroughWithFilePacket. If the receiver does not respond with a FinAckPacket within 15 seconds it assumes his counterpart has timed out or has successfully received all parts, but the FinAckPacket got lost in transit. Either way the sender is finished with his job. It is not finished if a RepairListPacket arrives.

Repair List Packet (RepairListPacket):
This packet is used if the receiver did not get the entire file and suddenly receives a ThroughWithFilePacket from the sender. Then it assembles the information of all his missing parts into this packet and sends it to the sender and waits for them to be retransmitted.

Final Acknowledgement Packet (FinAckPacket):
In case the receiver got all segments of the file it finally sends a FinAckPacket to the sender stating that it is finished now.
The simple hierarchy of all packets can be seen in Figure 3.2. Again, see the Appendix on page 26 for the exact format of the packets.

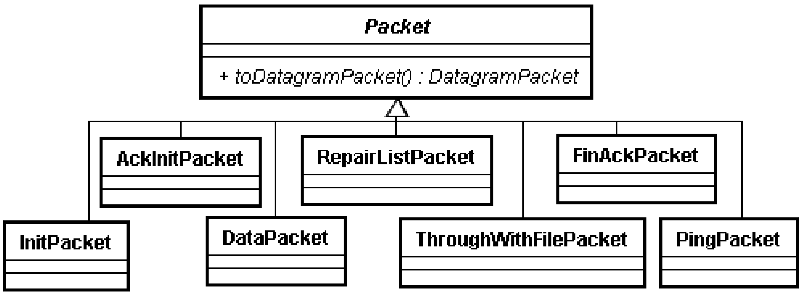


Figure 3.2: Class diagram of the involved packets

3.4 Implementation

There are basically three categories of classes: One category consists of classes belonging to the sender, one category belonging to the receiver and the third one are several auxiliary classes which do not have a side-specific purpose and may be used by both participants. Some of them are used as “surrounding” classes which for example instantiate the sender’s or receiver’s main classes or manage their connections, others are rather utilities which visualize the throughput and the like. The most important classes and their methods are presented in the following subsections.

3.4.1 Sender

In Figure 3.3 you can see the sender’s class diagram and at the top of the figure its most basic class: the ***DataPacketFactory***. As the name suggests, here are the packets are created and sent, but of course it is a bit more intricate than that. First, `sendInit()` has to be invoked to provide the receiver with the necessary file information. After the receiver acknowledged the initialization, `sendFile()` is invoked. Because we want to be able to send and receive data at the same time two threads are forked in this method, one – simply called *DataSender* – sends out the demanded data, while the other one takes care of the “control messages” which come in from the file receiver.

The corresponding ***ControlPacketReceiver*** class handles these control messages by listening with a blocking `receive(...)` method to *Ping-*, *FinAck-* or *RepairListPackets* in a `while(true)`-loop. A responding “pong” (actually still a PingPacket) will be returned in case of a PingPacket coming in. If a FinAckPacket arrives the loop is exited and a flag in the *data sending* object is set accordingly by invoking `setFinAckReceived()` on the sender, finally enabling threads of both to terminate. Also on arrival of the third packet a flag is set in the concurrently running sender thread and the *repair list* is passed on. Prior to starting the blocking `receive(...)` method the used *DatagramSocket* socket’s timeout was set to 15 seconds, which is a constant defined in an auxiliary class called *SomeUtilities*. By the way, this class contains various global constants and commonly used converter methods.

On the other hand, the ***DataSender*** is busy with sending DataPackets. It also consists of a `while`-loop which will break as soon as a final acknowledgement comes in. The corresponding flag, as mentioned before, can be set by the concurrently running *ControlPacketReceiver* object who has a reference to the DataSender. The `readListAndResend()` method can as well be invoked by that receiver to set a boolean flag in case of an incoming ReadListPacket. This flag causes the DataSender to fork another thread that runs an instance of ***ListReaderAndSender*** which simply fills the *PacketQueue* with the receiver’s missing file segments. The DataSender also utilizes an instance of the class *Packetizer* which takes care of the file's readout and conversion of it into DataPackets.

The ***Packetizer*** creates DataPackets by invoking `addToQueue(...)` which actually does the readout of the file (with the help of some auxiliary classes like *FileReader*) and adds them to a PacketQueue. Its `hasPacket()` method is convenient for callers and enables them to check for any waiting packets in the PacketQueue. If there is at least one packet, `getPacketToSend()` can be invoked which causes the PacketQueue length to be reduced by one and the according packet is handed over to the caller.

That ***PacketQueue*** is accessed by two threads, namely *DataSender* and *ListReaderAndSender*. Thus it has the special property that it is thread-safely accessible, meaning *tail*-insertion and *head*-retrieval are possible. Besides DataPackets this queue can also contain ThroughWithFilePackets, since it is able to insert any subclass of *Packet*.

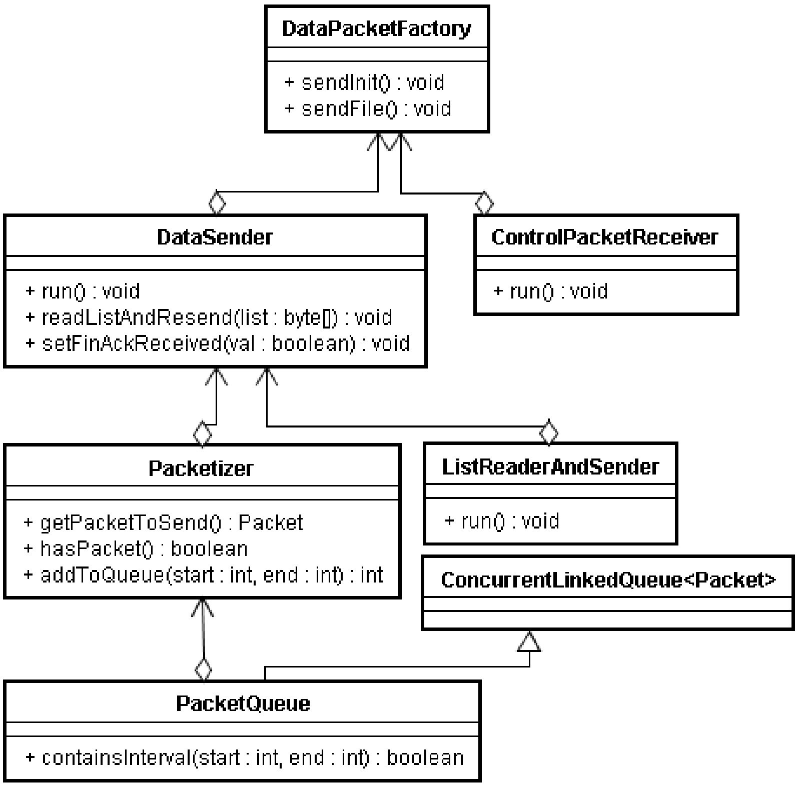

DataPacketFactory
+ sendInit() : void
+ sendFile() : void
DataSender
+ run() : void
+ readListAndResend(list : byte[]) : void
+ setFinAckReceived(val : boolean) : void
ControlPacketReceiver
+ run() : void
Packetizer
+ getPacketToSend() : Packet
+ hasPacket() : boolean
+ addToQueue(start : int, end : int) : int
ListReaderAndSender
+ run() : void
ConcurrentLinkedQueue<Packet>
PacketQueue
+ containsInterval(start : int, end : int) : boolean

Figure 3.3: Most important classes and methods used by the sender

3.4.2 Receiver

The receiver's class diagram shown in Figure 3.4 is a little bit simpler. The ***UDPServerSocket*** class is in some way the counterpart of the initialization message. The idea how to use this class would be to have a `while(listening)`-loop in which an instance of UDPServerSocket is called with its `accept()` method. In this method there is the DatagramSocket's blocking `receive(...)` method used that waits until a user connects to the known port 11111. When a user connects to this socket, the initialization is handled in `accept()` with the helper method `receiveInit()`, and it is finished by sending back an initialization acknowledgement. After the initialization acknowledgement went out the method returns a *DataAssembler*-instance which operates on a different port and is going to receive the following DataPackets.

The ***DataAssembler*** has one important public method: `receive()`. Besides timeout handling with the help of an *Estimator*, there is a `switch` statement which distinguishes the following packet types: *DataPacket*, *PingPacket* and *ThroughWithFilePacket*.
The `handleDataPacket(...)` method is called when a DataPacket is received. The received packets are redirected to a *MergingLinkedList* which manages the received and missing file segments.
Each second a PingPacket is sent out by `sendPingPacket()` and eventually there will be a PingPacket that returns, upon the arrival the ping will be registered in the *Estimator*.
An arriving ThroughWithFilePacket has the effect that `sendRepairList()` will be invoked which on its part requests the *MerginLinkedList* for missing packets and consequently, if there are any, they will be transmitted to the file sender.
And finally `sendFinAck()` is invoked if the MergingLinkedList reports that all segements are received.

The ***MeringLinkedList*** appeared quite a lot of times until now because it is an integral part of the receiver's classes. It is a special linked list that can incoporate *Interval* objects by using `addSorted(Interval)`. These Intervals can have three states: *missing*, *received* or *requested*, which are used to keep the overview over the file's segments. According to state and position in the linked list, the intervals can be merged by using `mergeListEntries()` to melt consecutive intervals into one interval. The missing-interval-list can be requested by invoking `getMissingIntervals()` like it is done by DataAssembler's `sendRepairList()` method. `isCompletelyReceived()` returns true as soon as the list contains only one element with the size of the transmitted file and is in the state "received".

The ***Estimator*** class was already partly described in Chapter 3.3 within the PingPacket description. Its method `log(...)` can take two kind of arguments. If a ping comes in, the RTT and timeout are calculated and updated according to Formula 3.1 and 3.2, the latter can be retrieved by calling `getTimeoutValue()`. Otherwise, if data arrives, the size of the received data can be passed to `log(...)` to keep track of the throughput. The current download speed can always be retrieved by using `getCurrentSpeed()` which returns the moving average over the last ten inserted values. A moving average is used in order to even out sharp peaks and drops.
Favorably at the end of the transmission one can call `printDiagram()` to see a visualization of the throughput over the elapsed time.

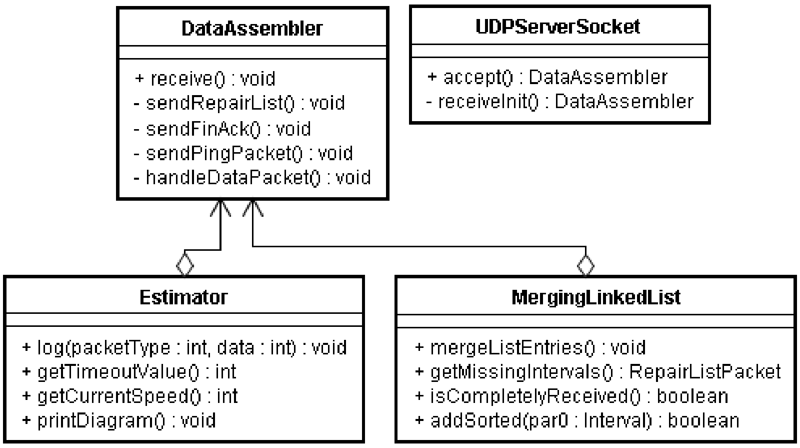

DataAssembler
+ receive() : void
- sendRepairList() : void
- sendFinAck() : void
- sendPingPacket() : void
- handleDataPacket() : void
UDPServerSocket
+ accept() : DataAssembler
- receiveInit() : DataAssembler
Estimator
+ log(packetType : int, data : int) : void
+ getTimeoutValue() : int
+ getCurrentSpeed() : int
+ printDiagram() : void
MergingLinkedList
+ mergeListEntries() : void
+ getMissingIntervals() : RepairListPacket
+ isCompletelyReceived() : boolean
+ addSorted(par0 : Interval) : boolean

Figure 3.4: Most important classes and methods used by the receiver

Chapter 4

Evaluation

4.1 TCP's Slow Recovery

In order to compare a TCP transmission and a transmission with sTCP, similar experiments have to be performed, i.e. the same file(s) have to be transmitted over the same physical connection for both protocols.

After having implemented sTCP, an implementation for TCP had to be composed. To see if the implementation is correct I first tried to send only one file. There should not be any problem due to TCP's slow recovery since we only have one file in one thread using one socket connection. If you compare the plots in Figures 4.1 and 4.2 which were captured in a wireless LAN, you can see that TCP has a lower throughput and thus must have overhead compared to our protocol. In these figures the TCP connection had an average speed of 596 kBytes/s as opposed to sTCP's 711 kBytes/s. The more or less 16% lower throughput is probably due to TCP's acknowledgements which implicitly cause a contention between data packets and themselves in the medium (air). Both packets have to compete for the medium because only one of them can be in the air at a given time, otherwise boht their signals will distort each other and will be unreadable by the receiver. It must also be mentioned that we were lucky in Figure 4.2 since there was no packet loss which had to be repaired.

The Figure 4.3 shows an example where some packet losses happened right at the beginning of the transmission. These consecutive losses are rather special and are discussed in Chapter 4.2 on page 17. The second drop at almost the end of the transmission is due to the sending of the *RepairListPacket(s)* which contain the missing segments from the drop in the beginning.

Before we extend the current implementation to more than one connections let us summarize what possiblities we have to do so:

- The most straightforward way to send multiple files is to have a TCP socket for each file and each socket gets its own thread for the transmission.
- The second way to send multiple files is almost the same as before, but instead of having n threads for n files, we just use one thread for n files combined with *Time Division Multiplexing*. This means after each `send(data_segment)` method's execution it is another socket's turn until all files are transmitted.
- The third way of sending multiple files is rather intricate. The idea consists of only using one socket and one thread for all files, but then the data segments from different files have to be made distinguishable on the application layer. But this method does not work if you try to send files to more than one user, because we only have one socket which cannot have multiple destination addresses.

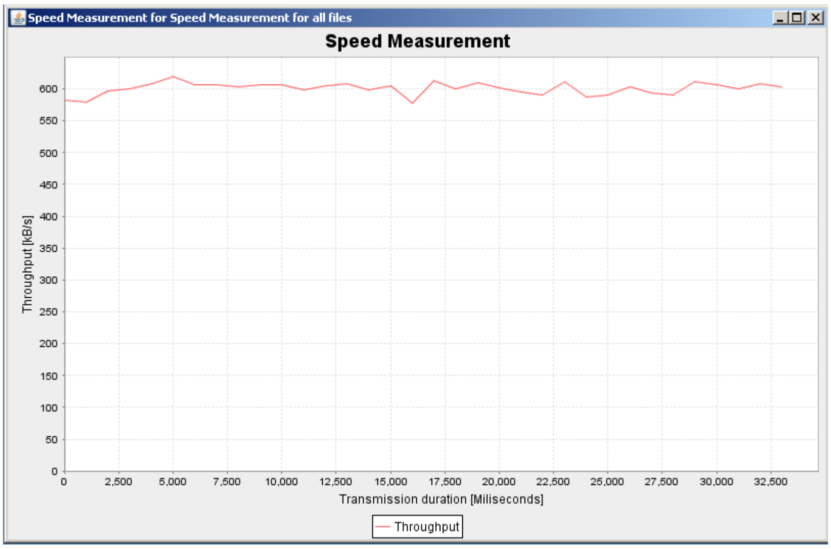


Figure 4.1: One file transmission using TCP

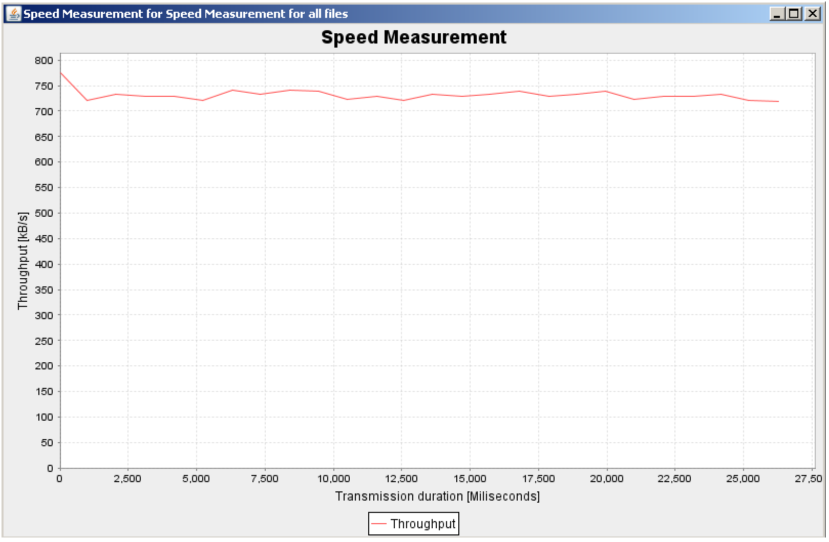


Figure 4.2: One file transmission using sTCP

The abstraction of the first approach to use a thread for each transmission is very appealing and thus the decision which one of these possibilities to take was made pretty fast.
In Figure 4.4, you can see the graph of two files whose transmission started at the same time but where one of these is smaller in size than the other. At about 18 seconds the first file was finished transmitting, but there is no noticeable change of performance. In fact that is not only the case in this picture, but also there was not one single measurement where I was able to detect TCP's slow recovery!

What went wrong? Since it also turned out that my decision to take the first approach was rather thriftless with resources, because I did not know how many simultaneous threads one can use at a time (around 13 threads/connections/files), I decided to change the implementation to the second approach, on the one hand to rule out the probability of some kind of "mis-implementation" and on the other hand to make the whole program a bit more efficient. "Mis-implementation" is between quotation marks because the files the sender transmits to the receiver arrive like they ought to, we just do not experience the slow recovery.

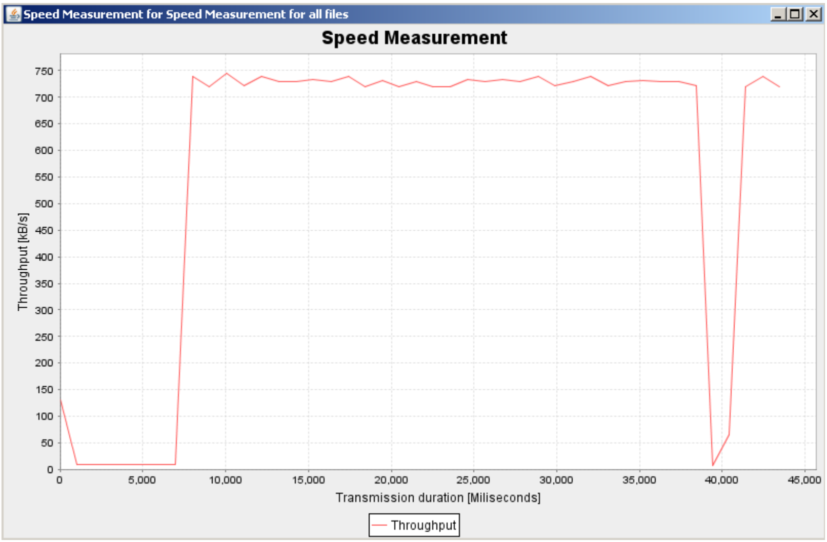


Figure 4.3: Transmitting two files using sTCP

Having changed the code to the second approach, I made remeasurements only to find the same result again.
A reason we do not see any slow recovery-like thing might be the low RTT. The measurements were all done in a wired or wireless LAN and might not quite be compareable with downloads from the Internet. TCP's linear growth rate is based on some RTTs and is updated accordingly. This means if we have a RTT of a few milliseconds the performance will also be "updated" within some milliseconds and thus will not be visible in the graphs which are only visualized every second. If we made the update intervals narrower down to 100ms like in Figure 4.5 you cannot see anything conspicuous. This tells us that either:

- TCP's slow recovery does not occur at all
- or it occurs but the variance in the graph is too high so that we do not notice it
- or the update intervall is still too broad.

In Figure 4.6 you can see a measurement with an update interval of 10ms which obviously makes the graph unreadable and useless.

4.2 Difficulties with Wireless

After some experiments in a wired LAN, I observed an interesting behavior of sTCP in the wireless LAN which did not happen when TCP was used for file transfers. In the already mentioned Figure 4.3 you can see a quite long drop in the beginning. With no obvious reason the transmission resumes again after about 7 seconds. Because the two laptops which were used to measure the transmission stood right next to each other, we can exclude the unavailability of the link due to long distance. Additionally the receiving laptop has a LED installed which is blinking if it receives any data. This LED was blinking the whole duration in which the sender was transmitting, including the part where allegedly there was not any throughput.
With some debugging effort it was possible to localize the piece of code where the execution got stuck. In fact it was the datagram `receive()` method of Java's UDP implementation which was waiting for packets. Since this bug did not appear in the wired LAN there must be something wrong below the transport layer.

With these observations I conclude to a possible explanation: It might be that the 802.11b link layer protocol has its difficulties with the full-packed UDP packets that are used in sTCP and this leads to some kind of congestion or packet unreadability. It would have been interesting to find out why the reception was detained, since I can rule out the possibility of a mis-implementation.

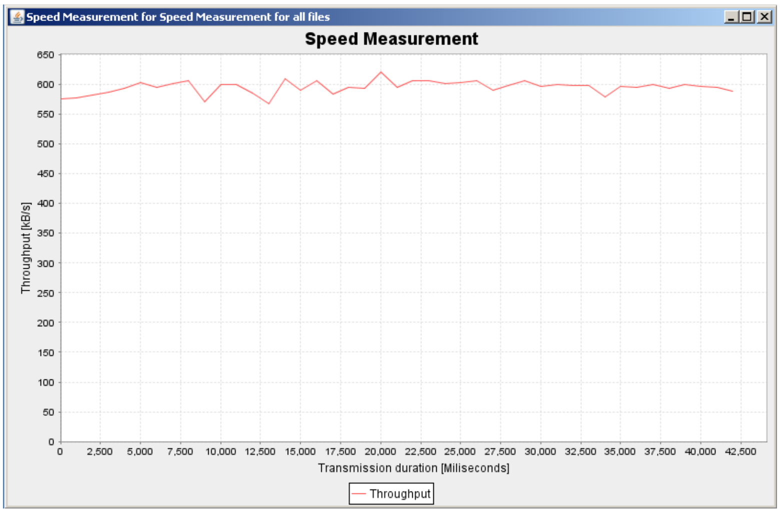


Figure 4.4: Two files transmission using TCP

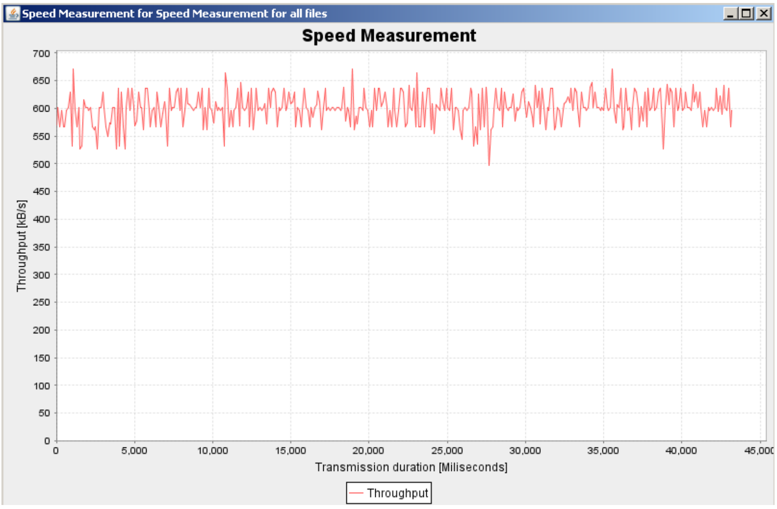


Figure 4.5: Two files transmitting with an update intervall of 100ms

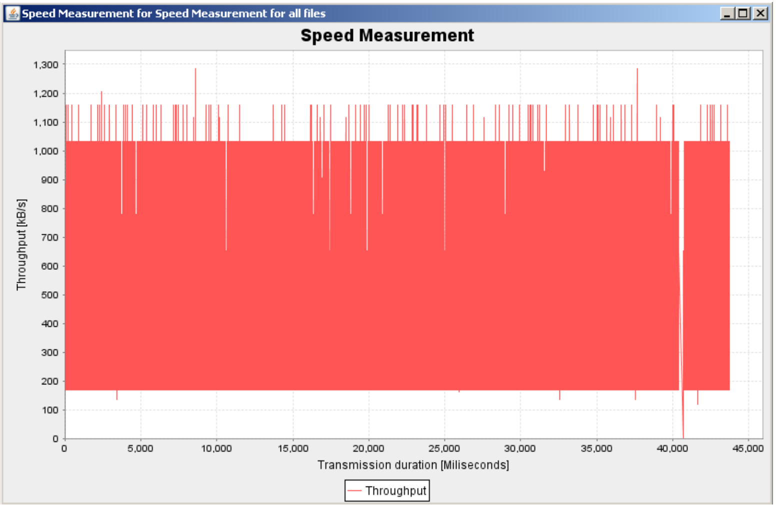


Figure 4.6: Two files transmitting with an update intervall of 10ms

Chapter 5

Future Work

This thesis might be used as a basis for further work. Here is some food for thoughts:

- It would be interesting to find out why the “wireless bug” described in Chapter 4.2 even occurs. Is the link layer really congested or is there another reason?
  Before using this protocol any further it is important to correct this bug since wireless access to the Internet becomes more and more important.

- In order not to be too aggressive, one could use some sort of mechanism that controls the size of the UDP packets. But it is questionable if one should even tackle this sort of problem, since this approach would head towards TCP and its congestion avoidance mechanism, which we tried to avoid on purpose.

- Due to lack of time I was not able to test sTCP over connections with high(er) delay. Before building the protocol into BitThief it would be a good idea to make some measurements and compare the results to a TCP measurement and see if it is worth to use this protocol instead of TCP.

- Some possible work for a more selfish BitThief: Instead of using sTCP, one could try to extend BitThief with the ideas of the paper *TCP Congestion Control with Misbehaving Receiver*[8]. In this paper three little, independent changes to the TCP protocol on receiver’s side are presented to defeat the congestion control mechanism. It would be in particular interesting, because one “only” needs to alter the receiver’s side and thus is also able to use it with peers that do not need the altered version of TCP, contrary to the protocol introduced in this thesis.
  The paper is from ’99 so before trying to implement anything one should check if the TCP versions of newer operating systems prevent this kind of spoofing.

Chapter 6

Remarks and Conclusions

6.1 Implementing a Transfer Protocol

“We will just make a protocol that has such-and-such properties and then . . . ”, as simple as it sounds, I quickly observed difficulties as I elaborated on the matter. Under discussion with some fellow students I realized that they were not aware of how intricate designing a protocol was. The development of a protocol is not trivial, and I think it is a good idea to show some example questions that arose and point out a few obstacles I ran across.

First of all, I was confused about the layered model, sure I knew the idea and heard about it in lectures and as well programmed a little chat-client, but I was not really aware of it until I truly had to think about, for example, packet delivery on the transport layer. How do I distinguish a UDP packet from another one? Where does a packet start and where is the ending in a data stream? Do I even have to care about it on my layer?

One category of difficulties I encountered were the various kinds of “variables” that had to be considered and determined. What role does the packet size play? What about the buffer size? And how much of the file can/should be read into the main memory at a time? What about errors and timeouts? Should they be fixed or depend on RTT?

Another category was the clash of efficient-programming and object-oriented concepts respecting-programming. Since we are computer scientists it is not a bad idea to stick to the object-oriented concepts, but in favor of efficiency I sometimes decided against the concepts.
For example: To prevent *leaking* or *capturing* (Aliasing)[1] of arrays one should make a copy of the array under certain circumstances and not pass the reference of the object. Copying the object would at least double the memory usage and include more work for the garbage collector. Nowadays files up to many hundreds of megabytes are quite common, so it is an important efficiency question.
Or what about *getters* and *setters*? Should I use getter and setter methods for object fields or should I rather access the fields directly and save some computation time?

These and many similar questions arose and were often a reason I spent quite a lot of time “googling” the answers, not always successfully though.

[1] http://sct.inf.ethz.ch/teaching/ws2006/KOOP/date06/lecture_06_-_aliasing.pdf

6.2 Conclusion on Semester Thesis

The objective of this thesis, to create an aggressive TCP-like protocol, has been achieved. Files can be sent and received successfully between two peers that both run sTCP. Files are sent in a way such that congestion control mechanisms are not used and congestion is totally ignored, which one might call "aggressive". To decide whether sTCP is usable for BitThief a more thorough analysis should be performed.

It was interesting and challenging to conceive my own protocol with all its odds and ends, and during the thesis it turned out that developing a protocol takes more time than one would anticipate.

Besides a more detailed comprehension of the upper part of the layered network model, I also got a better understanding of the Java Technology which was very useful for other lectures as well as instructive for myself.

Chapter 7

References and Related Work

7.1 References

- TCP – http://en.wikipedia.org/wiki/Transmission_Control_Protocol
- UDP – http://en.wikipedia.org/wiki/User_Datagram_Protocol
- BitThief – http://dcg.ethz.ch/projects/bitthief/
- "Vernetzte Systeme" lecture slides – http://dcg.ethz.ch/lectures/ss07/vs/material/chapter3/chapter3_1.pdf
- JFreeChart – http://www.jfree.org/jfreechart/

7.2 Related Work

In order to get a general overview of computer networks, network layers and protocols, in particular TCP, the reader is referred to [3], [11] and [12]. A very thorough summary of TCP, especially about the differences between the various TCP versions and their implications is given in [7]. Another analysis of TCP's evolution over the years is done in [6].

A different approach on aggressiveness (using TCP, though) compared to this thesis by altering the ACK sending is examined in [8], as already mentioned before. The TCP ACK problem, whether to use cumulative ACKs and risk a false congestion detection or acknowledgement of every packet, has received attention in [2].

Wireless links are more susceptible to random packet loss and thus impact the performance of TCP. Concerned with their improvement, alterations were proposed at different layers for example at the transport layer [5] and at the network layer [1].

This thesis evolved through the idea to extend the BitTorrent client presented in [4]. [10] is a paper concentrating on faithfulness in BitTorrent networks. By analyzing existing algorithms, they help building provably faithful Internet protocols.

Bibliography

[1] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving tcp/ip performance over wireless networks. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 2–11, New York, NY, USA, 1995. ACM Press.

[2] Anna R. Karlin, Claire Kenyon, and Dana Randall. Dynamic tcp acknowledgement and other stories about e/(e-1). In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 502–509, New York, NY, USA, 2001. ACM Press.

[3] James F. Kurose and Keith Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[5] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297, New York, NY, USA, 2001. ACM Press.

[6] Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the internet, 2004.

[7] Wael Noureddine and Fouad Tobagi. The transmission control protocol.

[8] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion contorl with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

[9] Stefan Schmid and Roger Wattenhofer. A tcp with guaranteed performance in networks with dynamic congestion and random wireless losses. In *WICON '06: Proceedings of the 2nd annual international workshop on Wireless internet*, page 9, New York, NY, USA, 2006. ACM Press.

[10] J. Shneidman, D. Parkes, and L. Massoulie. Faithfulness in internet algorithms. In *Proc. SIGCOMM Workshop on Practice and Theory of Incentives and Game Theory in Networked Systems (PINS'04), Portland, OR, USA*, September 2004.

[11] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[12] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

List of Figures

Appendix: Formats

Table 1: InitPacket's format

Offset	0	8	16	272	336
Length	8	8	256	64	32
Content	1	0x0. . . 0	blabla	0x0. . . 0	0x0. . . 0
Description	packet kind	filename's length	filename	filesize	return port

Table 2: AckInitPacket's format

Offset	0	8
Length	8	32
Content	2	0x0. . . 0
Description	packet kind	receiver's port

Table 3: DataPacket's format

Offset	0	8	40
Length	8	32	n
Content	3	0x0. . . 0	0x0. . . 0
Description	packet kind	segment start position	data

Table 4: PingPacket's format

Offset	0
Length	8
Content	5
Description	packet kind

Table 5: FinAckPacket's format

Offset	0
Length	8
Content	6
Description	packet kind

Table 6: ThroughWithFilePacket's format

Offset	0
Length	8
Content	7
Description	packet kind

Table 7: RepairListPacket's format

Offset	0	8	40	72	…
Length	8	32	32	32	…
Content	8	0x0…0	0x0…0	0x0…0	…
Description	packet kind	# missing segments	start segment1	end segment1	…