Leopold–Franzens–University
Innsbruck

Institute of Computer Science

Distributed and Parallel Systems Group

Reliable data transfer with MulTFRC

Master Thesis

Supervisors: Michael Welzl

Author: Schatz Florian Bakk.techn

December 17, 2009

Eidesstattliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Contents

List of Figures

Chapter 1

Introduction

Two protocols are dominating the Internet nowadays. On the one hand there is TCP, on the other hand there is the UDP protocol. While they both operate on the transport layer of the ISO/OSI model (section 1.1), respectively the Internet model, and they are both using IP on the Internet Layer, there are several important properties differentiating one from the other. These differences will be covered in this thesis.

The Transmission Control Protocol (TCP, section 1.3) offers a reliable data transmission and implements congestion control, whereas the User Datagram Protocol (UDP, section 1.2) does not provide any reliability regarding the reception of the sent packets nor has any kind of congestion control. Congestion in the network causes TCP to decrease the sending rate, in the absence of congestion it increases the rate slowly. TCP is also able to provide a reasonable fairness between flows. But as some of the characteristics of TCP does not fit the requirements of certain Internet applications, many alternatives have been proposed. UDP is one of them and it is mainly used for content streaming. Recent studies have shown that traffic created by streaming applications are responsible for a large part of today's Internet traffic. The authors of [San09] have stated that "report findings include a dramatic shift in consumer behavior towards real-time 'experience now' applications and away from bulk download 'experience later' behavior. Compared to last year's results, real-time entertainment traffic has exploded to now account for 26.6 percent of total traffic in 2009". As UDP traffic does not reduce the sending rate in case of congestion, it is considered harmful for competing protocol flows. Therefore it would be advisable to use protocols, which incorporate some congestion control mechanisms, to create fairly distributed rates among multiple connections. As an improvement over UDP, TFRC ([Jaa04], section 1.4) has been introduced as a basis for new protocols, which provides a congestion control for multiple applications. TFRC stands for TCP-Friendly-Rate-Control. The authors of [DW08] have proposed an enhancement of TFRC, which is called MulTFRC (subsection 1.4.3). MulTFRC is able to achieve the same transmission rates as multiple TFRC connections.

The goal of this thesis is the creation of a protocol, which uses MulTFRC for congestion control and which provides a reliable packet transmission. As this protocol is based on a preexisting TFRC implementation, this thesis covers the addition of the second major difference in functionality between TCP and UDP, as several mechanisms are implemented to achieve reliability. With this addition, reliable MulTFRC will have the functionalities of TCP but will be much easier to manipulate regarding the prioritization of data flows. This thesis will investigate the possibilities to combine the congestion control capabilities of TFRC/MulTFRC and the reliable data transfer functionalities

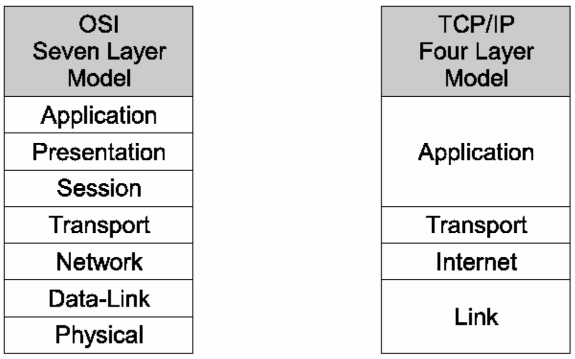


Figure 1.1: Protocol Layers

of TCP. Therefore it will give introductions about TCP, especially its reliability, and into TFRC, including the description of several existing TFRC implementations. Furthermore it will present MulTFRC. Both protocols will be evaluated regarding their possible usefulness in creating a new protocol, which will be based on TFRC-like functions for rate control, but in comparison to existing TFRC implementations, will have a reliable data transfer. The implementation and the functionalities of this protocol will be covered in this thesis. The thesis will be concluded with tests and benchmarks comparing this new protocol with TCP. The tests are used to show environments in which a protocol implementing the TFRC or MulTFRC specification for congestion control will have advantages in speed and stability.

In the following chapters basic information will be presented about the different network layers and protocols.

1.1 Protocol Layers

To obtain structure in the network, layers are used to implement and design network protocols. The idea is that every protocol is part of one layer. On one layer a set of problems is solved and all functionalities of a protocol are offered as services to the layer above and therefore to all higher layers. The highest layer can use data transmission without coping with the underlying problems. This principal is an important part of this thesis and will be discussed more thoroughly in section 3.7. The OSI Model is an abstract way of describing a layered network design. OSI Model stands for Open System Interconnection Reference Model. The protocol stack contains the Application, Presentation, Session, Transport, Network, Data-Link and Physical Layers. The IETF's Internet model [Bra89] is not designed as rigidly as the OSI model and is generally considered to consist of four layers, namely the Application, Transport, Internet and Link Layers. But there is ongoing discussion about the number of layers, ranging from three to five.

1.2 UDP: User Datagram Protocol

The UDP Protocol is one of the simplest protocols in the Internet. It has hardly any features besides the basic transmission of data. The original specification of UDP is described in RFC768 [Pos80]. It provides the transmission service of so called datagrams using IP to the application layer. For

that task no connection has to be established prior to a transmission. UDP is able to broadcast and multicast packets, sending the same packet to multiple hosts at once.
There is no congestion control or reliable delivery of packets using UDP, but the lack of these properties makes the protocol preferable in certain cases. One of these cases is video streaming, where it is more important that packets are delivered on time and loss of some packets can be tolerated. Other areas of usage are online gaming, voice-over-IP and the Domain Name System (DNS). Without the usage of congestion control mechanisms, the sending rate does not adapt to network conditions and therefore it gives a smoother rate. Because of its lack of a congestion control mechanism, it is considered harmful for responsive traffic, which results in a limited bandwidth available for UDP by some providers.
Each UDP datagram consists of application data and an UDP header. The header variables, which will be described shortly, are:

- Source Port: Port number at the source. This header field is optional.
- Destination Port: Port number at the destination.
- Length: Specifies the complete size of the datagram (header and data).
- Checksum: A checksum is calculated to check data integrity. This header field is optional in IPv4.

1.3 TCP: Transmission Control Protocol

The probably most-used transport protocol in the Internet nowadays is the Transmission Control Protocol, or TCP. It is operating on the transport layer and is therefore able to use all services provided by the lower layers, commonly it is used in conjunction with IP on the Internet Layer (also called the Network Layer), hence it is often referred to as TCP/IP.
TCP is creating a virtual channel between two hosts, providing a reliable data transmission between the sender and the receiver to all applications using TCP and it is able to handle packets which arrive out-of-order. TCP is able to handle full-duplex communication, allowing transmissions in both directions. Examples for applications using TCP are Web browsers, file transfer programs, etc. As IP does not provide reliability, TCP has to implement this functionality. To achieve reliability, the receiver of a TCP connection has to acknowledge every received packet. Without any acknowledgement or in case of an explicit packet request, a packet is considered to be lost and TCP retransmits these packets as often as necessary until the correct reception of the packet has been acknowledged. Further information about reliability are discussed in chapter 2. One property of TCP is that as soon as packet loss is detected, the sending rate is reduced according to the used congestion control mechanism.
TCP was specified in 1981 (RFC793 [Pos81]) and has been modified and enhanced in multiple ways since then. An important fact is that the original specification of TCP does not contain any congestion control at all. Congestion control has been devised in 1988 [Jac88] and has been specified with RFC2001 [Ste97] and RFC2581 [APS99].
To establish and finish a connection, TCP is using a 3-way handshake, consisting of the SYN- and FIN-flags in conjunction with acknowledgements. As soon as a connection is established, the involved sockets are accessible uniquely by the involved parties. The advantage of this 3-way handshake is that each side can be sure about the properties of the communication partner previous to the actual transmission. A TCP connection is identified by the combination of the source's and

receiver's IP addresses in conjunction with the involved port numbers, this combination is also often referred to as a socket.
Each TCP segment is containing multiple header variables which will be described shortly:

- Source Port: Port number at the source.
- Destination Port: Port number at the destination.
- Sequence Number: Identifies each segment by the byte position.
- Acknowledgement Number: TCP segments can transmit new data together with the acknowledgement of old segments. These acknowledgements are used to identify the segments which have been received successfully.
- Data Offset: Because the header length can vary in size, this field contains the size of the header.
- Reserved: Reserved space for future enhancements.
- Flags: These flag bits are used to specify additional information about the segment. For example these flags are used for the 3-way handshake.
- Window: Specifies the number of segments allowed to be sent according to the TCP flow control.
- Checksum: A checksum is calculated, so that the integrity of data can be verified.
- Urgent Pointer: This pointer is used to specify where urgent data starts.
- Options: The header is able to carry additional information by the usage of the header options.

A TCP segment is also carrying the data itself. The size of transmitted data can be chosen according to the network environment and the needs of the application.
There are different TCP variants, distinguished mostly by the used congestion control schemes and retransmission techniques. These different variants are described in section 2.2.

1.4 TFRC: TCP-Friendly-Rate-Control

TFRC [FIH$^+$03] is no real network protocol in comparison to TCP and UDP. It only provides congestion control mechanisms to the developer of a protocol or an application. Its goal is to have a low variation of throughput over time, and at the same time it is obtaining reasonably fair bandwidth consumption compared to TCP flows. A data flow is considered to be "reasonably fair if its sending rate is generally within a factor of two of the sending rate of a TCP flow under the same condition" [FIH$^+$03]. These properties make it an alternative to applications which are normally using UDP. By the usage of congestion control the sending rate is adapted depending on network conditions. This behavior is not only more fair to other flows, but it is also useful for connections using TFRC. As stated in section 1.2, the throughput of UDP may be restricted. With an active congestion control there is no need for any limitation.
While the property of having less variation may have advantages in certain areas of usage, it also has

its disadvantages. One of these disadvantages is the fact that in case of changes in the available bandwidth, TFRC is not able to respond as fast as TCP [Jaa04]. This also can cause more congestion. In the specification of TFRC the following remark it is therefore stated [FIH$^+$03]: “Thus TFRC should only be used when the application has a requirement for smooth throughput, in particular, avoiding TCP’s halving of the sending rate in response to a single packet drop. For applications that simply need to transfer as much data as possible in as short a time as possible we recommend using TCP, or if reliability is not required, using an Additive Increase Multiplicative Decrease (AIMD) congestion control scheme with similar parameters to those used by TCP” [FIH$^+$03]. TFRC is not supposed to be an alternative to TCP, but it is an effort to provide well defined congestion control methods for protocols like UDP, to prevent those from harming responsive traffic. It is a rate-based protocol, meaning that a formula is responsible for the calculation of the sending rate. To gather all necessary information for the formula both ends of the transmission has to take multiple measurements, like the round-trip-time and the packet loss rate. The specification suggests that feedback packets are sent from the receiver to the sender, where the packets are used to update the variables. The sending rate is calculated on the sender side. As soon as any of these parameters are updated, the formula is used for calculating an appropriate rate for the changed network situation. The formula specified in RFC3448 [FIH$^+$03] is:

$$X = \frac{s}{R * \sqrt{\frac{2*b*p}{3}} + t_{RTO} * (3 * \sqrt{\frac{3*b*p}{8}}) * p * (1 + 32 * p^2)}$$

- X represents the sending rate in bytes/second.
- s is the size of the packet.
- R is the round-trip-time.
- p is the loss event rate.
- t_{RTO} is the value of the TCP retransmission timer.
- b is the number of packets acknowledged with one acknowledgement.

The specification of TFRC also suggests to use the value **1** for b because many TCP implementations do not use delayed acknowledgements or they use proper byte counting. The value of t_{RTO} can be replaced by $4 * R$ which should be a close-enough approximation. Besides this formula, TFRC leaves everything else to the developer of the protocol. More about the properties of the TFRC specification will be discussed in section 2.2.
TFRC was specified originally in the RFC3448 [FIH$^+$03] in the year 2003. Since then the specification has been revised, which produced various Internet Drafts (for example [FIH$^+$07]), and has been replaced by RFC5348 in the year 2008 [FIH$^+$08]. Also additional drafts ([FIKU07], [KFS07]) have been published to improve the original TFRC specification for specific scenarios. At the time of writing, many of these documents were still being worked on. One of the above mentioned drafts is called “TCP Friendly Rate Control (TFRC): The Small-Packet (SP) Variant” [FIKU07]. As TFRC’s bandwidth formula is not well suited to handle traffic created by small packets, this TFRC variant “is intended for flows that need to send frequent small packets, with less than 1500 bytes per packet” [FIKU07]. It is able to calculate an appropriate rate for applications incorporating either fixed or varying packet sizes.
The congestion control mechanisms which are specified by TFRC can not only be used in a transport layer protocol, but also as stated in [FIH$^+$03] “in an application incorporating end-to-end congestion

control at the application level, or in the context of endpoint congestion management". There are several TFRC implementations and modifications. The most commonly known implementation is the Datagram Congestion Control Protocol (DCCP, subsection 1.4.1), but [Jaa04] also lists for example TCP-Friendly Multicast Congestion Control (TFMCC, described in subsection 1.4.2) and Adaptive TCP-Friendly-Rate-Control (ATFRC, described in [CWwL03]).

1.4.1 DCCP: Datagram Congestion Control Protocol

The Datagram Congestion Control Protocol, or DCCP, is a transport layer protocol (like TCP and UDP). DCCP is intended for applications which have the need of fast delivery instead of reliability in receipt or order of packets. It removes the task of inventing a congestion control mechanism from the developer of an application, who would use normally UDP because of timing constraints.
It provides several services to the upper layers.

- Reliable handshake for connection establishment and termination, similar to TCP.
- Possibility for feature negotiation.
- Different congestion control methods, including TCP-like and TFRC. Further mechanisms can be introduced easily.
- Path Maximum Transmission Unit discovery.
- Explicit Congestion Notification.
- Possibility for reliable acknowledgements.
- Unreliable transmission of packets.

The Datagram Congestion Control Protocol is specified in RFC4340 [KHF06] and has been included in the Linux kernel since 2006.

1.4.2 TFMCC: TCP-Friendly Multicast Congestion Control

While most protocols are only able to handle unicast communications, TCP-Friendly Multicast Congestion Control is providing a TCP-friendly rate in a multicast environment. This can be helpful for servers that stream videos to multiple hosts at the same time. Serving multiple hosts which are requesting the same information individually has proven to be ineffective. TFMCC proposes a sending rate which is valid for all connections. In comparison to TFRC, each receiver is computing the sending rate based on the bandwidth equation instead of the sender. For this calculation each receiver has to measure the round-trip-time and the loss event rate. As the transmission of feedback packets by every receiver could easily cause an overflow at the sender, only a subset of these receivers are allowed to transmit the feedback packets, including a suggested sending rate, to the sender. Only receivers which request a decreased sending rate or have a too large round-trip-time are allowed to transmit reports. These packets are used at the sender to adapt the sending rate if requested and to enable the measurement of the round-trip-time at the receivers.
One receiver is handled differently, as the receiver with the supposedly lowest throughput is chosen to be responsible for the increase of the sending rate. The communication between this receiver and

the sender is similar to the traffic in other TFRC implementations. If the receiver is removed from the communication or another receiver appears to have a lower throughput, a new receiver is in charge of the transmission of feedback packets. TFMCC, as it is described in [WH06], suggests a slight modification to the TFRC formula:

$$X = \frac{8 * s}{R * \sqrt{\frac{2*p}{3}} + (12 * \sqrt{\frac{3*p}{8}}) * p * (1 + 32 * p^2)}$$

As an alternative there exists a TFMCC-variant which adapts the packet size instead of the sending rate. This behavior has been suggested by ‘TCP-Friendly Rate Control (TFRC): The Small-Packet (SP) Variant” [FIKU07].

1.4.3 MulTFRC

MulTFRC is proposed in [DW08]. The idea is to enable a single flow to behave like multiple TFRC flows. The similar idea has already been proposed for TCP with MulTCP [CO98]. Simulating the aggression of multiple TFRC flows may have advantages in both speed and stability. Similar to TFRC, MulTFRC is only an equation for the calculation of the sending rate and no complete protocol. Therefore it can also be used in an end-to-end application, for example on top of UDP. As described in subsection 1.4.1, for example DCCP supports the addition of new congestion control mechanisms, so MulTFRC can also be added to existing protocols. The complete MulTFRC formula can be seen in subsection 3.6.1. While the simple multiplication of the sending rate calculated by the TFRC formula with a chosen factor may have similar speed potential, this possibility is less advantageous because of various side effects [OMP01]. For example low results, which may be triggered by congestion, are just multiplied too, leading to a less smooth transmission. While TFRC is supposed to compete with TCP in matters of bandwidth, this is not the case in matters of usage. Therefore this thesis investigates the idea of a reliable TFRC/MulTFRC protocol. As a special ability MulTFRC is also able to be less aggressive than a standard TCP flow. Areas of usage for this ability are transmissions which run in the background and should not interfere with other more important transmissions.

1.4.4 MPEG-4 Video Transfer with TFRC

As video streaming is an important part of today’s Internet traffic, e.g. MPEG-4 Video Transfer with TCP-friendly Rate Control Protocol has been introduced. MPEG-4 Video Transfer with TFRC is described in [MWMM01]. The basic idea behind this protocol is to use the TFRC equation to calculate the sending rate, which will then be used to determine the parameters of video encoding. If the sending rate decreases, the quality of the video is adjusted, so that the packet loss rate decreases and all necessary information can be transferred with the available sending rate.

Another approach to improve the behavior of video streaming is described in [CZ04]. It is called MULTFRC and shall not be confused with MulTFRC (subsection 1.4.3) which is used in this thesis. The basic idea of MULTFRC is the usage of multiple TFRC connections in a wireless network. In this environment TFRC has the problem that packet loss is assumed to be caused by congestion, while “the bulk of packet loss is due to error at the physical layer” [CZ04]. Therefore MULTFRC is using multiple connections to achieve an appropriate rate. This number of connections is recalculated depending on the round-trip-time.

Chapter 2

Functionalities

As TCP provides multiple sophisticated services to the application layer, the reliable TFRC/MulTFRC protocol is in need of many equivalent functionalities. The goal is a rate-based, connection-oriented, reliable transmission stream. The following points show the requirements for the new protocol:

- MulTFRC has to use either the TFRC or the MulTFRC bandwidth formula to calculate an appropriate sending rate. This is necessary to recover from any existing congestions, but also to avoid new ones. As the program is using additional techniques to achieve a reliable data stream, the result of these formulas can only be considered as an upper bound for the actual bandwidth.
- The protocol has to keep track of the buffer available on the receiving side which is necessary to prevent packet loss due to the lack of available memory. The sender is not allowed to transmit more packets at once than the receiver can handle, even if the bandwidth calculation would allow higher sending rates.
- Both sides have to discard packets which have already been received.
- The sending side has to maintain a timer for every packet that has been sent and which is yet unacknowledged. If this timer hits a certain value, the sender has to assume that the packet has been lost and therefore it has to retransmit this packet.
- If the sender is informed explicitly of packet loss, it has to retransmit the lost packet.
- The receiving side of a MulTFRC transmission has to send an acknowledgement for every packet which has been received.
- The receiver has to inform the other side if packet loss is detected.
- The receiver has to be able to store out of order packets and to insert these packets at the appropriate position.

2.1 Reliability

Reliability is one of the main differences between UDP, or any other TFRC protocol, and TCP. UDP does not adapt its behavior depending on the correct receipt of each data segment which has been sent via the network to a receiver. As an enhancement to UDP TFRC protocols only adjust its transmission rates in response to packet loss, but there are no retransmissions of lost packets. TCP on the other hand makes sure that each data segment, which has been sent, is being received correctly on the other side of the connection. Reliability in a computer network is considered to be one of the most challenging areas in development [KR05]. The main reason for this difficulty lies in the problem that the Internet Protocol (IP), which nearly all Transport Layer protocols in the Internet are using on its Internet Layer, only provides unreliable data transfer. As discussed in the chapter about the protocol layers (section 1.1), lower layers can provide all its functionalities as services to higher layers, but IP does not implement any solution for the problem of reliability.

There are several possibilities to achieve reliability in a computer network. The following three possibilities will be discussed in the duration of this chapter: Reliability in the Application Layer, in the Transport Layer and in the Internet Layer. While implementing reliability in the Application Layer may have the advantage that it can be achieved in a relatively quick and easy way without changing any of the underlying network protocols, it has the disadvantage that it can only be used in one specific application which it was implemented for. It would be preferable to have the possibility for a reliable data transfer being offered as a service from one of the lower layers. Reliability in a data transfer is an important property for many applications, but there are other cases where it is not necessary and where lost packets are preferable over the higher complexity of a reliable implementation and the time consumed for retransmissions and timeouts. So another possibility would be the implementation of reliability in the Internet Layer. If IP would offer reliability, every Transport Layer protocol using IP would therefore be reliable too. As already stated, this could have disadvantages in situations in which the high complexity, which is necessary to retransmit packets, is preventing the transmission from achieving the necessary throughput. In cases like that, a control bit could be used to decide between reliable and unreliable data communication. One advantage of the addition of reliability in the Internet Layer would lie in the fact that this way reliability would not only be guaranteed between end-to-end hosts, but also between all intermediate routers. Packet loss and data corruption could be detected at all routers on its path through the network immediately and retransmissions only come from the preceding router. But using the Internet Layer for reliability has several significant disadvantages. At every router checking has to be done if packet loss or data corruption has occurred. This leads to time and resource problems at routers. Another problem is based on the fact that, if a new Internet Layer protocol is introduced, it is necessary to implement it not only at the end hosts, but also in all intermediate routers in the network. Similar to the difficulties IPv6 has encountered to replace IPv4, a new protocol at this layer has significant problems of being spread. Additionally all Transport Layer protocols which want to gain the benefits of a new Internet Layer protocol, namely reliability serviced by the lower layer, have to be modified too. The last logical choice is the Transport Layer. TCP has already proven that end-to-end reliability can be achieved at the Transport Layer. While it cannot provide reliability in each step throughout the network, this has the advantage that the checking for packet loss and data corruption has only to be done once at the end-host. This saves computational power and time in the routers, because their task is simply the forwarding of messages to its target addresses. A reliable Transport Layer protocol can use IP as its Internet Layer protocol and can therefore use existing routers in the Internet without changes.

Considering the fact that all three possibilities mentioned above have their own advantages and disadvantages, an implementation of reliability could be done at all of these layers depending on the area of usage. Due to the fact that using the Transport Layer means creating a reliable network protocol, which can be reused by different applications and because it can still use one of today's core elements of the Internet, namely IP, without modification, this possibility is considered the most

promising one and is therefore the one implemented in most existing protocols.
The tool developed in this thesis is operating on the Application Layer and therefore the reliability is achieved also at the Application Layer.

2.1.1 Basics

As already stated, a protocol using IP or, as in the case of this document, an application incorporating UDP has to provide reliability on its own, if it is a desired ability. The basic idea behind the creation of a reliable protocol is that acknowledgements are sent for every packet from one side to the other. To limit the traffic created by this method some protocols are not acknowledging every packet, but are either sending at most one in a specific time frame or are acknowledging multiple packets at once. In most protocols an acknowledgement for a specific packet indicates that all earlier packets have been received correctly too. This is the case in TCP and also in the reliable MulTFRC program developed in this document, but can also be solved in other ways. By acknowledging a packet, the receiving side of a transmission is informing the sending side which packets have been received until the moment the acknowledgement has been sent. With this information the sender can decide if new packets can be sent or old packets have to be retransmitted. In addition to acknowledgements, the sender has to use a retransmission timer. This timer is responsible to trigger retransmissions of packets which have not been acknowledged within a specified time frame. The value of this retransmission timer has to be calculated depending on the network conditions. It has to be chosen wisely, a too large value would cause a long delay in case of packet loss, too small values would trigger unnecessary retransmissions. The minimal value has to be the round-trip-time, but in that case packets would be a retransmitted in case of any additional delay. The program developed in this document is using the specification of TCP's retransmission timer [PA00] with only slight changes. As an improvement, an explicit packet loss notification can be used. The idea behind this is the possibility for the receiver to immediately trigger the retransmission of a specific packet. This can be realized in a number of ways, for example by the usage of a specific header field or by defining that multiple acknowledgements for one packet are indicating the loss of a packet. Depending on the implementation, the lost packet can either be the packet with the next higher sequence number or with the stated sequence number. The explicit loss notification prevents the sending process from having to wait for the retransmission timer to detect packet loss. Therefore it is no crucial part for reliability, because eventually the timer is taking care of lost packets, but it can improve the performance of the protocol. Similar to TCP this MulTFRC protocol is using duplicate acknowledgements, but instead of waiting for three duplicates of the same acknowledgement, by default MulTFRC is triggering a retransmission already with the second acknowledgement for the same packet. As none of these functionalities are covered in the TFRC/MulTFRC specification, there is no guideline to implement it, but to compare effectively the behavior and speed with TCP, it has been inspired by its counterparts in TCP.

2.1.2 Go-Back-N or Selective Repeat

There are two different ways of handling retransmissions of lost packets. The simpler solution is, as soon as loss of a packet is detected, to retransmit all packets starting from the lost packet. This way of retransmission is called Go-Back-N. The more sophisticated approach is called Selective Repeat, which indicates that only the lost packet is retransmitted and then the sender is continues sending new packets. Although Selective Repeat has obviously its advantages, it also has its disadvantages. In comparison with Go-Back-N it only retransmits lost packets and can then continue with new packets.

The downside is the higher complexity of packet handling on the receiver side. It has to store all out-of-order packets until missing packets have been received and it has to insert new packets in the right position. Another disadvantage lies in the fact that recovery can last long if there is not only one lost packet missing but if multiple packets have been lost at once. A sender using Selective Repeat can only retransmit a packet after it has been requested explicitly by the receiver, therefore only one lost packet can be resent during one round-trip-time. If the receiver is able to notify the sender about multiple losses at once, the recovery can be faster. Such a technic is used by TCPSack. To illustrate the advantages and disadvantages of both methods several situations are described. As a first example, a sender transmitting twenty packets every round-trip-time is considered. If the first of these packets is lost, the sender retransmits the lost packet and the receiver receives this retransmitted packet around one round-trip-time after its intended arrival. At this point the receiver inserts the packet before the nineteen already received and buffered packets with higher sequence numbers. It takes around one round trip time to recover from a single packet loss. On the other hand, if multiple packets are lost at once, Selective Repeat does not perform very well. Reusing the above example of a sender sending twenty packets every round-trip-time, we now assume that five packets in a row are lost. We have shown that it takes one round-trip-time to recover from one lost packet, while the receiver had to store nineteen packets at that time. With five lost packets this only covers the first packet, but in comparison to the former example it still has to store the previously received packets. For the second lost packet it takes another round-trip-time to be received, while at this time another nineteen new packets have arrived, which sums up to thirty-eight stored packets. If we continue this example to the end, it would have taken around five round-trip-times to recover from five lost packets, while at the end the buffer at the receiver was containing ninety-five packets. The example shows that the time and memory consumption is increasing significantly with every lost packet. This leads to a point where it is preferable to completely retransmit all packets from a certain point in the transmission. There is also an additional downside of Selective Repeat. As the example above demonstrated the memory consumption of the receiving side can grow very large if several packets are lost in one round-trip-time, memory considerations have to be taken into account. The sending side has to make sure that newly sent packets can also be stored at the receiver in case a packet with a lower sequence number is lost. Therefore, a value is chosen or calculated depending on the properties of the receiver, which represents the number of packets allowed in the network at once. This value is called the window size. The sending process is keeping track of all sent and still unacknowledged packets. This number of packets must not be higher than the value of the window size. The downside of this approach is that as soon as the window size is reached, no new packets are allowed to be sent until an acknowledgement for a new packet arrives. In comparison to the Selective Repeat approach, the Go-Back-N method does not adapt its behavior depending on how many packets have been lost in one round-trip-time, as it is retransmitting all packets starting with the first lost packet. Therefore out-of-order packets arriving at the receiver are discarded because they will be resent. This behavior is responsible for the fact that this approach will transfer much more packets than the counterpart. Taking the example from above in which the sender sends twenty packets in one round-trip-time it can be noted that if all twenty packets are lost, it will take the same time to recover from these losses as if only one of these packets has been lost. It takes around one round-trip-time to retransmit all lost packets, and as the property of Go-Back-N suggests no packets has to be stored at the receiver. It is important to note that the example disregarded the effect of packet loss on the sending rate when congestion control is in place. The goal of this section has been the comparison of the two different approaches. As a result of the above example, the tool developed in this thesis is using both variants for retransmissions.

2.2 Congestion Control

The goal of congestion control is to adapt the sending rate according to different situations in the network. If congestion is detected, the sender decreases the sending rate, while it can increase the rate otherwise. There are many possible ways to implement congestion control, this tool is using the congestion control functionalities described in the original TFRC specification [FIH$^+$03]. This section describes the properties of congestion control used in TCP and will give an overview of the TFRC proposal. As the program also implements MulTFRC, these changes will be also covered. As congestion control depends on the detection of lost packets, some of these mechanisms are already covered in subsection 2.1.1.

UDP does not provide any kind of congestion control as it is always sending at a constant rate. Therefore it does not react to packet loss and congestion. The original TCP specification contains no congestion control and only implements a window-based flow control which prevents that more packets arrive at the receiving side than can be handled. The theory is similar to the window size described in subsection 2.1.2, making sure that there is buffer available for the storage of out-of-order packets. In TCP this window is called receiver window and is allowing only a limited number of sent but unacknowledged packets at any moment. This method protects the receiver from overflow. The receiver window value is conveyed to the sender in each acknowledgement. This behavior is described in RFC813 [Cla82].

As the congestion collapse of the network was experienced, TCP has been enhanced with congestion control mechanisms: Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery. Congestion control introduces another window called the congestion window or **cwnd**. As stated in section 1.3, multiple TCP variants exist. The first TCP implementation is called **TCP Tahoe** and supports Slow Start and Congestion Avoidance. In the **Slow Start** phase TCP has an exponential increase of the congestion window which is increased by one for every received acknowledgement. As an example, a connection is able to send one packet in the first round-trip-time; in the second round-trip-time, two packets can be transmitted and after the acknowledgements for both of these packets have arrived four packets can be sent. In the **Congestion Avoidance** phase the rate is increased linearly, increased by one packet every round-trip time. A new connection starts with the **cwnd** set to one packet and TCP is in the Slow Start phase. By the usage of that behavior, the transmission is started slowly instead of having high speed in the beginning which is a way of probing the network. In the case of TCP Tahoe, this increase is stopped as soon as packet loss is detected. One problem of this TCP variant is that packet loss can only be detected by a timeout which introduces long delay. As timeout is experienced the **cwnd** is reset to one packet and TCP enters the Slow Start phase. TCP has an additional variable called the Slow Start threshold (SSthresh) and at the time of packet loss this variable is set to half of the congestion window. TCP Tahoe implements another phase of operating Congestion Avoidance. As already said, in case of packet loss, the congestion window is allowed to increase the sending rate starting from one using the Slow Start phase(doubling the rate every round-trip-time). It is in this phase until it reaches SSthresh and from that point on the congestion window is only allowed to increase linearly (Congestion Avoidance), ensuring that the transmission rate approaches the available bandwidth slowly.
As an improvement over TCP Tahoe, **TCP Reno** was developed. It is able to detect packet loss more efficiently. If multiple acknowledgments are received for the same packet, the packet is assumed to be lost and it is retransmitted immediately. This is denoted as **Fast Retransmit**. The sender waits for three duplicate acknowledgements for the same packets before deciding that the first unacknowledged packet is lost. Waiting for three duplicate acknowledgements is used because of

the possibility that packets have arrived in the wrong order. If the packet loss is detected in this way, it is resent before the retransmission timer has expired, therefore it is called Fast Retransmit. TCP Reno still implements timeout loss detection. If too many packets or acknowledgements are lost, the only way that TCP can detect packet loss is a timeout. In case of a timeout, TCP Reno behaves in the same way as TCP Tahoe. It reduces its cwnd to one packet and enters the **Slow Start** phase. If loss is detected by three duplicate acknowledgements, TCP enters the **Fast Recovery** phase. In the Fast Recovery phase the sender halves the current window and sets SSthresh and cwnd to that value. The cwnd is increased by one packet for every duplicate acknowledgement which has been received. After the missing packet has been retransmitted, the sender continues sending new packets. As soon as a new acknowledgement arrives, the value of SSthresh is used for the congestion window. It continues its operation in the Congestion Avoidance phase. This approach works well for single packet loss, but does not cope well for multiple packet losses. To cope with multiple lost packets, an enhancement of TCP Reno, called **New Reno**, exists. The main difference is that it does not exit Fast Recovery until all packets in the window have been acknowledged. Another extension of TCP Reno is **TCP Selective Acknowledgement** which acknowledges packets selectively instead of cumulatively. Therefore multiple packet losses can be conveyed in a single acknowledgement and they can be retransmitted in a single round-trip-time. There are several other TCP variants implementing different congestion control algorithms which deviate significantly from the standard TCP protocol and therefore their descriptions are omitted here.

TFRC/MulTFRC does not specify a Slow Start mechanism but it is considered useful to increase the speed of the transmission in the beginning instead of starting with an arbitrary value. Therefore a function has been implemented to increase slowly the speed until packet loss occurs. Afterwards the TFRC/MulTFRC properties are responsible for congestion control. Although this reliable implementation is using two windows to keep track of unacknowledged packets, TFRC and MulTFRC are still rate-based protocols. They use throughput equations to calculate the rate of a flow. The rate is calculated depending on the round-trip-time, the loss event rate, the number of lost packets in a loss event and some additional parameters which are constant during a connection (i.e. the number of packets acknowledged by a single acknowledgement, by default 1, and the number of TFRC flows MulTFRC should emulate). If congestion and therefore packet loss occurs, the function will adapt the sending rate based on new values using the equations instead of applying any separate congestion control methods. The equation of TFRC is specified in RFC3448 ([FIH$^+$03], section 1.4), while the formula of MulTFRC has been proposed in [DW08] and its implementation is described in subsection 3.6.1. In this implementation the result of the TFRC/MulTFRC formula is only one of several parameters responsible for the actual achieved throughput. The only difference between TFRC and MulTFRC is the used throughput equation, there are no additional changes in flow control.

The two windows are used to keep track of the network buffer (subsection 3.8.1) and the application buffer (subsection 3.8.2). As both windows are operating in the same way, it would have been possible to merge these values and only consider the smaller window. As the window for the application buffer is only necessary for the Selective Repeat approach (subsection 2.1.2) both windows have been kept to demonstrate the necessary additions for this retransmission variant. The two implemented windows are implemented as a protection from receiver overflow. The maximum number of packets which can be sent without acknowledgements is limited. If this maximum is reached, new packets can only be sent if an acknowledgement for a new packet arrives. This means that packets have left the network and there is new available space. In case of congestion it takes one round-trip-time for the retransmitted packet to reach the receiver and therefore acknowledgements of new packets are delayed.

Chapter 3

Implementation

As a basis for the implementation of the new reliable TFRC/MulTFRC protocol an already existing program has been used. The tool is based upon the code developed by Jörg Widmer in the year 2000 [Wid00] and has been written in C.
This product includes software developed by the Computer Science Department at University College London (section A.10) which has been developed between 1995 and 1997 and is responsible for the unreliable transmission of packets. The C code is using UDP for packet transmission. As UDP does not provide the services described in chapter 2, neither congestion control nor reliability of transmission, these functionalities are implemented on the application layer (section 1.1). The TFRC specification has been already implemented by Jörg Widmer while MulTFRC has been added here. The basic implementation does not include reliability which is another outcome of this thesis.

The following section will only cover the important extensions and changes done to the code from [Wid00] and their influence on the programs behavior, therefore it is not covering all aspects of the tool.

3.1 Installation

This section will cover the installation and configuration necessary to be able to execute the program that offers a reliable data transfer using the TFRC and MulTFRC protocol. As this code was mainly developed using Ubuntu system (section A.5) as the operating system, certain system specific commands are executed and would have to be replaced in case another operating system is used. The program has been tested on multiple Linux-distributions but it is not possible to run the program on 64-bit operating systems. To achieve support several adjustments would have to be made, replacing all 32-bit-libraries with 64-bit-libraries.
The program consists of two directories, one is containing the NetUDP-code developed by the Computer Science Department at University College London (section A.10) and has only been slightly adapted. This directory is called *common*. The second directory is called *source* which contains the basic program including all TFRC/MulTFRC functions.
In case that the program files are obtained or downloaded from other sources, some files may need a change in permissions. These files are called *install.sh* and *configure* which are contained in the *common*-folder, while there are *makeperl.sh* and *configure* in the *source*-directory. All of these files need the permission to be executed. The following code is an example for giving permissions to execute to one of these files.

```
chmod 744 install.sh
```

A clean installation has shown that all necessary packets are already delivered in an up-to-data Ubuntu, but in older Ubuntu-distributions it was necessary to ensure that all necessary tools are available, therefore an additional package called *build-essential* had to be installed. “This package contains an informational list of packages which are considered essential for building Debian packages.”[bui] As this is an Ubuntu-specific command, required development packages would have to be installed manually if an operating system does not have a similar function and the packages are not already provided:

```
sudo apt-get install build-essential
```

A script provided by Jörg Widmer is responsible for the compilation of all files:

```
./install.sh
```

Depending on the characteristics of the operating system it is necessary that the *source*-directory is included in the PATH-environment for the correct execution of the program, as Jörg Widmer has suggested [Wid00].
For obvious reasons it is crucial that no firewall is blocking the communication created by the two processes. As the program is using UDP to transmit packets, firewalls would also identify the traffic as UDP.

3.2 Execution

The idea behind the command line interface has been to implement an equivalent tool to other file transfer protocols which are also working via command line. So the command prompt has been developed in a way that it resembles the command prompt of already existing protocols. Next to the program name, the IP-address of the communication partner and an unused port number have to be stated (additionally to this port, the next higher one is also used for the transmission). The chosen port has to be accessible, therefore it is wise to choose a port number higher than 49151. As additional parameters the name of the file which should be transferred and its destination name has to be entered. Several other parameters have already been provided by the original code.

```
./sender IP-address port sourcefile destinationfile [ttl] [-m mulTFRC] [-b buffer
     size] [-y max. bitrate] [-i min. bitrate] [-c const. bitrate]
```

On the receiving side a process has also to be started, so that this process receives incoming packets and stores its data. While this process seems to be additional in comparison to existing ones like FTP, SFTP and others, where a file transfer can be initiated by one side, this is based on the fact that in case of FTP or SFTP there is a background running server handling all requests. Similar to servers, it is necessary to start the receiver prior to the sender.

```
./receiver IP-address port [ttl] [-n lognr] [-w num. weights] [-m mulTFRC] [-x
    buffer multiplicator]
```

The original code provides the possibility to adapt the time-to-live value on both sides of the communication (**ttl**).
At the sender it is also possible to confine the throughput within defined boundaries using the

parameters **y** and **i**. By the usage of parameter **c** it is possible to use a constant bitrate instead of the result of the TFRC or MulTFRC equation. Even with a constant bitrate, the two implemented windows (subsection 3.8.1, subsection 3.8.2) are in effect, preventing the tool from achieving the set bitrate.
At the receiver it is possible to add a number to the file name which is used to create a log of the transmission with the parameter **n**. Parameter **w** can be used to change the number of weights which are used to calculate the loss history for TFRC.
As an additional parameter a new one has been introduced to read the value for MulTFRC (the number of TFRC flows MulTFRC should behave like). Parameter **m** must be a positive real number. With **m** equal to **1**, the standard TFRC equation is used, while the program uses the new MulTFRC equation in case of all other values. As this parameter can be set on both sides of the communication, if the value is set at the sender, it will overwrite the receiver's parameter. Without this parameters value **m** equal to 1 is used. It is also possible to change the maximum number of packets which can be buffered at once at the receiver using the parameter **b**.
To improve the behavior of the program, a parameter is added which acts as a multiplier for the network buffer. More about the effects of this variable is discussed in subsection 4.1.2.

3.3 Advanced Configuration

Besides many command-line arguments, this program also provides additional configuration possibilities. The necessary parameters are stored in the file *config.h* in the *source*-directory. This file is used on both sides of the transmission and must be therefore modified at the sender and the receiver. The following values can be modified:

- CLOCK_GRANULARITY: This value defines the clock granularity for the computation of TCP's retransmission timer at the sender. It is set in milliseconds and the default value is chosen to be 10 ms. See subsection 3.7.4.

- MAX_WAIT: It specifies the maximum amount of time between packets. If this time has passed, the program terminates. It is set in seconds with 10 seconds as the default value. This value is used at both sides of the transmission and have not to be equal. MAX_WAIT is used in subsection 3.9.2.

- DATA_SIZE: As this simulated protocol has to transfer data, a predefined data field is specified with 1400 as the default size. Larger values are not allowed, but smaller values can be used for testing purposes. As the name of the file is transferred in the first packet, the length of the name can be at most the value of DATA_SIZE. It is crucial that both sides of the transmission have the same value.

- DUPACKS_FOR_RETRANSMISSION: As discussed in subsection 2.1.1, retransmissions are triggered by a single duplicate acknowledgement for a packet. This property can be changed using this value as it specifies the number of duplicate acknowledgements necessary to cause a retransmission. The minimal value for this variable is one, as zero would trigger a retransmission with the first acknowledgement. TCP retransmits packets after three duplicate acknowledgements. It is crucial that both sides of the transmission have the same value.

Besides these parameters, this file also contains many important parts of the program which should not be adapted without reason. As this file is a header for the C source code, it has to be recompiled in case of modification (see section 3.1).

3.4 Sending and receiving processes

The transmission of data needs two sides. Although both sides are sending and receiving packets, one side is called the sender and the other one is called the receiver.
The sender is in possession of the original file and has the task to transmit the data of the file to a second host. The sender consists of two processes which are both independent loops. These processes have both access to the same information with the usage of shared memory. Due to speed limitations, the access to this shared memory is not exclusive to one process but is solved with other variables to allow access to all program values at all times (subsection 3.7.5). In every loop iteration the first process is creating one packet with all its necessary data (see subsection 3.5.1) and is sending it to the receiver. To provide the program with the desired sending rate, this process is set dormant for a calculated time and is afterwards resuming with the transmission of new packets in the following iterations. This loop is also containing the retransmission timer (subsection 3.7.4).
To be able to receive acknowledgements, a second process is running constantly with the purpose of listening to the network socket for incoming packets. This is calculating the smoothed round-trip-time and the value for the retransmission timer with the received variables. As a failsafe, this process is also responsible to terminate the complete program if no packets have arrived in a specified time (section subsection 3.9.2).

The main process of the receiver is using incoming packets to retrieve the data within it and stores the content to a file. To offer reliable data transmission the receiver is acknowledging received packets and is informing the sender of lost packets. It is also responsible for the calculation of the bandwidth based on either the TFRC or MulTFRC formula and has to send these information to the sender. In comparison to the processes at the sender, this process is both responsible for both sending and receiving. This is possible because the receiver is only sending packets after it has received one.
As described in subsection 3.9.2, an additional thread is acting as the failsafe on this side to terminate the receiver.
In total the program is using four processes to simulate the TFRC and MulTFRC protocol, two on each sides.

3.5 Packets

TCP does not have a special acknowledging packet, as it uses a normal packet sent from the receiver to the sender and conveys an acknowledgement in a special header field. This has the advantage that along with an acknowledgement new data can be transmitted from the receiver to the sender. This implementation of TFRC/MulTFRC does not need this kind of two way communication, it only needs the transmission of acknowledgements and measurement variables to the sender. Different variables are sent in the other direction from the sender to the receiver. For this reason the packets that are transmitted from the receiver to sender are significantly different and therefore also smaller than the packets sent in the other direction.

3.5.1 Packet sent by the sender

A data field in the packet is responsible for the transmission of data from the sender to the receiver. The size of this data field is defined by the value **DATA_SIZE**. The header of a packet contains the following fields:

packet_no	number of sent packets
mseq	sequence number
dseq	reserved for additional sequence number
ts	timestamp used in TFRC/MulTFRC computation
rtt	round-trip-time
bitrate	current bitrate
round	round
mode	TFRC-mode
start_flag	flag to identify first packet
end_flag	flag to identify last packet
data_size	size of used data array
timestamp	timestamp used for retransmission timer
rto	value of retransmission timer at time of transmission
mulTFRC	number of TFRC connections which are simulated
buffer_size	number of packets which can be stored at the receiver

New values which have been added in comparison to the original code are printed **bold**. These new variables are **packet_no**, **start_flag**, **end_flag**, **data_size**, **timestamp**, **rto**, **mulTFRC** and **buffer_size**. The variables **packet_no** and **mseq** are both counting the number of sent packets but have a significant different meaning. **mseq** is set to a specific value representing the ID of the sent packet and it is unique for each packet, while the value **packet_no** is always increased by one for every packet that leaves the sender. Therefore **packet_no** must have at least the value of **mseq**. Figure 3.1 shows the case that after the packet with **mseq** and **packet_no** equal to **1** has been acknowledged successfully, the subsequent packet (**mseq** and **packet_no** both equal to **2**) is lost. For the retransmission of this packet, the value for **packet_no** has to be increased to **3** while the value of **mseq** is still **2**. This simple example is presented just to illustrate this difference and not to show a completely detailed behavior of the protocol and therefore is quite simplified. Two numbers are introduced because TFRC and

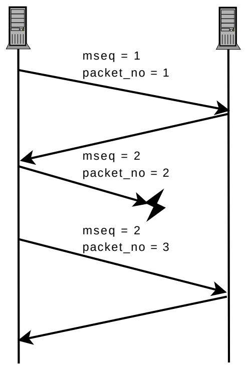


Figure 3.1: Comparison between variables **mseq** and **packet_no**

MulTFRC have a different congestion control than TCP. **mseq** is used for ensuring reliable data transfer and **packet_no** is used for TFRC or MulTFRC loss event rate measurement. Combining this two values would increase significantly the complexity of the program. In comparison to a TCP header, this header uses a packet number as the sequence number whereas TCP uses the position of the first data byte of a packet in the byte stream.
As the original code does not support data transmission, a data field has been added. Because this data field has a predefined size (**DATA_SIZE**), the program can use just a part of it. Therefore it is also necessary to inform the receiving side of how much actual data is in the data field. Under normal circumstances only in the first (which carries the name of the file) and last packet (only carries as much data as is left at the sending process) the value of **data_size** should be lower than **DATA_SIZE**, otherwise these values should be identical.
The values **start_flag** and **end_flag** are used as flags to identify the first and the last packet of a transmission. The differentiation of these packets from the rest is necessary for a transmission restart (**start_flag**) and termination (**end_flag**).
Although the original code already sends a timestamp (**ts**) which is used with the TFRC formula for bandwidth computation, another timestamp (**timestamp**) has been added for the retransmission timer. These two variables are using a different format and therefore have not been merged. It would have been possible to recalculate the value of **ts** to have the required format, but as this would have to be done several time, two separate values are in the header.
Because the MulTFRC formula is in need of the value of the retransmission timer (**rto**), this has also been added to the packets header. TFRC uses 4*rtt as an approximation for the retransmission timer and MulTFRC uses an calculation of the retransmission timer that reassembles that of the standard TCP protocol. The number of TFRC connections (**mulTFRC**) is transferred for the same formula.
The value **buffer_size** represents the number of additional packets which should be allowed to be stored at the receiving side.
The packet also contains the current value of the round-trip-time(**rtt**) and the current bitrate(**bitrate**, is used for the log file). **round** is counting the number of acknowledgements which have arrived at the sender and is used at the receiver for logging. The variable **mode** represents the phase in which the sender is. It can be either *M_INITRTT* (before the first acknowledgement arrives), *M_SLOWSTART* (before the first packet loss) or *M_Normal* (TFRC or MulTFRC equation is used for bandwidth computation).
Comparing this packet with the packet of TCP (see section 1.3) several important observations can be made:

- The sequence number of a packet is identified by counting packets instead of the position of the first byte.

- The data field has a fixed size, although not everything has to be used.
- No port numbers are necessary. As TFRC/MulTFRC uses UDP for the transmission of packets there is no need to state the port number in the TFRC/MulTFRC header.

3.5.2 Acknowledgement

The header of an acknowledging packet, in TFRC known as a feedback packet, contains the following fields (parameters written in **bold** have been added for this extension):

mseq	sequence number
dseq	reserved for additional sequence number
ts	timestamp used in TFRC computation
b_rep	throughput of receiver
b_exp	bandwidth result of TFRC/MulTFRC formula
rto	value of retransmission timer
ignore_packet	flag to identify packets which should be ignored
recvbuf	size of socket buffer at the receiver
timestamp	timestamp used for retransmission timer

Similar to the packet transmitted from the sending side to the receiver, a second timestamp (**timestamp**) has been added for the retransmission timer.
Additionally appended to the acknowledgement is a flag to identify packets (**ignore_packet**) which are only used to update different variables but are not supposed to acknowledge any packets.
The value **recvbuf** is calculated at the receiver and informs the sender of the number of bytes which can be stored at the socket at once.
The header also contains a field to transmit the retransmission timer (**rto**, as the sender has to calculate the retransmission timer itself, this would not be necessary). The variable **b_exp** is conveying the result of the TFRC or the MulTFRC formula, while **b_rep** is the throughput of the receiver.

3.6 MulTFRC

This section describes the changes necessary to extend the existing TFRC implementation with MulTFRC. It is split into two parts. The first part is covering the added MulTFRC equation (the original TFRC equation is still in the code), while the second part describes the changes in the loss calculation.

3.6.1 Rate calculation

A new function (Listing 3.1) has been implemented which calculates the rate at the receiver using the MulTFRC formula.

Listing 3.1: Bandwidith

```
/*
The MulTFRC equation needs other variables for the equation
n  ... number of TFRC-flows
pe ... loss event probability of the cumulative flow
b  ... number of packets acknowledged by one packet - is 1
RTT... round trip time
T  ... value for retransmission timer
j  ... average number of packets lost in a loss event
*/
double mulTFRC_equation(double n, double pe, double b, double RTT, double T,
    double j) {
        double j1 = j;
        if(n<12)j1=n-n*pow((n-1)/n,j1);
        if (j1 > n)
                j1 = ceil(n);
        double a = sqrt(pe * b * j1 * (24 * n * n + pe * b * j1 * (n * n - 4  * n
             * j1 + 4 * j1 * j1)));
        double x = (j1 * pe * b * (2 * j1 - n) + a) / (6 * n * n * pe);
        double w = n * x / (2 * b ) * (1 + 3 * n / j1);
        double z = T * (1 + 32 * pe * pe) / (1 - pe);
        double q1 = j * n / w;
        if (q1 > n)
                q1 = n;
        double q;
        if ((q1 * z / (x * RTT)) >= n) {
                q = n;
        } else {
                q = q1 * z / (x * RTT);
        }
        return (((1 - q / n) / (pe * x * RTT) + q / (z * (1 - pe)))*PKT_SIZE);
}
```

3.6.2 Loss calculation

In addition to the loss event rate used by TFRC, the MulTFRC formula (subsection 3.6.1) needs a value which represents the average number of packets lost in a loss event. Therefore a function (Listing 3.2) is created which checks for every packet if a packet has arrived in order. For this function the value **packet_no** has been added to the header of the packet. This variable can detect easily packet losses even if there has been multiple retransmissions of packets.
After each packet arrival the function **update_history** is executed. The variable **high_offs** is representing the highest packet number which has been received until that moment. If the packet number is lower than expected, it will be discarded (it means that, because of a network error, a duplicate packet has been received). If it is the next in order packet the value is updated. If the packet number is higher than the expected message number, a new packet loss is detected. According to the specification of MulTFRC, packet losses which happen within one round-trip-time are handled differently and are all stored in one value of an array.

Listing 3.2: Loss Calculation 1

```
/*
mseq     ...equals packet_no in packet header
arrival ...time of arrival of last packet
rtt      ...round trip time
*/
int update_history(u_int32 mseq, int64 arrival, double rtt) {
        static int64 lastloss = 0;
        int i;
        mseq_offs = mseq;

        if (mseq_offs > high_offs + 1) { /* new loss */
                if ((arrival - lastloss > rtt * MS_TO_US)) { /* new loss event */
                        ++num_loss;
                        if (num_loss >= HIST_SIZE) {
                                fprintf(stderr, "loss history buffer overflow\n")
                                    ;
                                return 0;
                        }
                        /* shift lost packets */
                     for(i = 6; i >=0; i--) {
                                lost_packet_in_intervall[i+1]=
                                    lost_packet_in_intervall[i];
                        }
                        lost_packet_in_intervall[0] = mseq_offs-high_offs - 1;

                        log_loss(arrival);
                        lastloss = arrival;
                } else { /* new lost packet in old round trip time */
                        lost_packet_in_intervall[0]+=mseq_offs-high_offs - 1;
                        ppl[ppl_cur] += mseq_offs-high_offs - 1;
                }
                high_offs = mseq_offs;
        }
        if (mseq_offs == high_offs + 1) { /* prevent reordered packets from
            changing high_offs */
        /*              packet in order */
                high_offs = mseq_offs;
        }
        ++ppl[ppl_cur]; /* add packet to last interval */

        return 1;
}
```

As this function is calculating an array of values representing the number of packets lost in one loss event (separated by round-trip-time), these values have to be used to calculate a weighted average. Recent packet losses are more important than older packet losses, therefore each value of the array is multiplied with a weighting parameter. Listing 3.3 is using the result of the above function (Listing 3.2) to calculate the desired value.

Listing 3.3: Loss Calculation 2

```
double weight_lost_packets() {
        double weights[8] = {1.0, 1.0, 1.0, 1.0, 0.8, 0.6, 0.4, 0.2};
        double sum = 0;
        double sum_weights = 0;
        double avg_lost_packets = 0;
        int i;

        for (i = 0; i < 8; i++) {
                if (lost_packet_in_intervall[i]>0){
                        sum = sum + ( weights[i] * lost_packet_in_intervall[i] );
                        sum_weights = sum_weights + weights[i];
                }
        }
        avg_lost_packets = sum / sum_weights;
        return avg_lost_packets;
}
```

3.7 Reliability

The second major addition next to MulTFRC (subsection 1.4.3) is reliability. An important step to create reliability is the differentiation of how to handle packets and acknowledgements. This section also covers the retransmission timer and the storage of the necessary variables.

3.7.1 Receiver

After packet arrival the receiver extracts the sequence number of a received packet and distinguishes different cases depending on the extracted sequence number:

- Received sequence number is smaller than the expected sequence number: In this case the packet is considered a retransmission and therefore the data contained by the packet should have already been saved and acknowledged. Nevertheless values within the packet are used to recalculate the loss history, the round-trip-time and other TFRC values. The receiver acknowledges a retransmission with a packet containing the sequence number of the last correctly received packet.

  Special Case: If the packet is identified as the first packet by looking at the start flag, it considers this packet a part of a new transmission and therefore resets all its values and restarts with this packet. The reason for this special behavior lies in the possibility that the sender is terminated before the transmission has been completed and a new transmission has been started with different values, for example a different file. Without this special case the receiver would attach new file data to the original one and would store it under the wrong file name.

- Received sequence number is equal to the expected sequence number: This case describes a new in order packet. The receiver extracts the data field and attaches it to the previously received data. It also extracts different header values, uses it to recalculate the TFRC or MulTFRC equation and sends the result in the next acknowledgement. From that time on the receiver expects a packet with the sequence number equal to the expected sequence number increased by one.

  Special Case: If the **end_flag** of the received packets is set to **1**, this packet is considered to be the last packet of the transmission. The receiver stops its task after the acknowledgement for this packet has been sent.

- Received sequence number is bigger than the expected sequence number: A too large sequence number indicates a lost or out of order packet. A receiver reacts by sending an acknowledgement with the sequence number of the last acknowledged packet. To prevent the sender from sending multiple retransmissions of the same packet, the receiver counts the number of acknowledgements with the same sequence number and sets a bit (“ignore bit”) in the header of the acknowledgement to **1**, if more than two packets are sent (see section 3.3 for modification of this behavior). This could be improved because loss of this acknowledgement would trigger a timeout at the sender, but this is left for future work. The receiver checks if the data is already stored in the application buffer (subsection 3.8.2) and, if this is not the case, it stores the data and other values in the buffer depending on available memory.

In all three cases the receiver sends an acknowledgement to the sender with updated TFRC values and timestamps.

3.7.2 Sender

The acknowledging packet is received by the sender and its sequence number and different timing values are extracted. Four different cases can be distinguished:

- Received sequence number is smaller than the expected sequence number decreased by one: This packet does not acknowledge any new packet. The sender uses these packets only to update its runtime variables.
- Received sequence number is equal to the expected sequence number decreased by one and the ignore_packet flag is not set: This case represents the possibility that two acknowledgements in a row are received for the same packet. This can only happen if the receiving side is requesting a retransmission of the packet with the packet ID equal to the expected sequence number, therefore the sender is triggering a retransmission applying the Selective Repeat method.
- Received sequence number is exactly one number smaller than the expected sequence number and the ignore_packet flag is set: As the ignore_packet flag is set, this packet cannot trigger a retransmission. Nevertheless these packets are crucial to update the runtime variables.
- Received sequence number is bigger than or equal to the expected sequence number: This case represents the anticipated behavior of the program where maybe several but at least one new packet is acknowledged. Therefore these packets can be removed from the number of unacknowledged packets and this makes place for new packets to be transmitted. Additionally the expected sequence number is increased to the sequence number of the first not yet acknowledged packet.

3.7.3 Linked List Management

The program has to be able to retransmit packets if packets is detected to be lost. These packet losses can be triggered by multiple duplicate acknowledgements of previous already acknowledged packets or by a timeout waiting for an acknowledgement. To be able to track timeouts, the program has to keep track of all transmission times of all sent but still unacknowledged packets. As the number of unacknowledged sent packets depends on various variables like the round-trip-time and the sending rate, there is no knowledge in advance about its maximum value during execution. Therefore a solution had to be found to keep track of the time each packet was sent without unnecessary memory consumption. These requirements prevented the usage of arrays which would have been the traditional way of storing several time values.
As a possibility to achieve an unlimited number of stored values, a linked list has been implemented. This linked list contains the time a packet has been sent, the sequence number of the packet and a link to information about the next packet. Listing 3.4 is responsible to store the sequence number of a packet and the time it was sent.

Listing 3.4: Linked List 0

```
typedef struct list_node {
        int64 time;
        u_int32 packet_id;
        struct list_node *next;
} llnode;
```

Listing 3.5 is creating a new node at the end of the linked list after every transmission of a packet. The time of transmission is stored in the **time** value, **packet_id** contains the packet's sequence number. Additional conditions have to be added in case that there is no other node in the linked list. If the list has been empty, the new node is the new head and tail of the list. If there are already other nodes, this new node is added linked by the previous tail node and is considered the new tail.

Listing 3.5: Linked List 1

```
node = (llnode *)malloc(sizeof(llnode));
node->time = end;
node->packet_id = info->packets_sent;
if(tail == NULL) {
        tail = node;
}
else {
        tail->next = node;
        tail = node;
}
if(head == NULL) {
        head = node;
}
```

To improve performance it is imperative to prevent the program to run through all old nodes, which represent already acknowledged packets. Therefore nodes are removed from the list as soon as a corresponding acknowledgement has arrived. These removed nodes can then be deleted and its allocated memory can be freed.
During every loop iteration the program checks if the list is empty. If this is not the case, it compares the value of the packet ID, which is stored in the head, with the packet sequence number which was acknowledged (**expected_mseq** represents the last acknowledged packet plus one). If the sequence number in the head node is lower, the head is deleted and its successor in the list is the new head. This is repeated until the list only contains one element or the sequence number, which is stored in the head node, represents a packet which has not been acknowledged yet. Listing 3.6, Listing 3.7 and Listing 3.8 are responsible to clean up the linked list.

Listing 3.6: Linked List 2

```
while(head != tail) {
        if(head->packet_id < info->expected_mseq) {
                tmp = head->next;
                free(head);
                head = tmp;
        }
        else {
                break;
        }
}
```

Listing 3.7: Linked List 3

```
if(head == tail) {
        if(head->packet_id < info->expected_mseq) {
                tmp = head;
        head = NULL;
                tail = NULL;
                free(tmp);
        }
}
```

As a special case the program has to check if there is only one packet in the linked list. If this is the case and the **packet_id** of this node has already been acknowledged, the complete linked list is freed and all pointers are set **NULL**. This special case is shown in Listing 3.7. Listing 3.8 shows the case of a retransmission, which has been triggered by the retransmission timer, so that the program is using the Go-Back-N approach (see subsection 2.1.2) and therefore the linked list has to be freed completely. As the head of the list represents the first unacknowledged packet and this packet is the only one which can trigger a retransmission, a new value has to be stored as a transmission time for this packet and all of its already transmitted (but unacknowledged) successors.

Listing 3.8: Linked List 4

```
if(info->packets_sent <= info->expected_mseq) {
        while (head != tail) {
                tmp = head->next;
                free(head);
        head = tmp;
        }
        if(head != NULL)
                free(head);
        tail = NULL;
}
```

3.7.4 Retransmission Timer

The timer implemented in this code is based on the timer used in TCP [PA00].
The retransmission timer is the last possible packet loss detection if there was no explicit loss notification. A timeout indicates that the program has not received an acknowledgement for a packet within a specified time interval. This time interval changes dynamically depending on the round-trip-time. As the round-trip-time is not known before the first acknowledgement arrives, it is defined as three seconds until the first arrival.
In the program the round-trip-time is stored in the variable **r**, while the current value of the retransmission timer is stored under **rto**. Listing 3.9 is calculating the retransmission timer according to the specification of TCP.

Listing 3.9: Retransmission Timer 2

```
r=(get_time()-ntohl((double)rep->timestamp))/MICROSEC;

if(srtt == 0.0) {
        srtt = r;
        rttvar = r/2;
} else {
        rttvar = (1 - (1/beta)) * rttvar + (1/beta) * abs(srtt-r);
        srtt = (1 - (1/alpha)) * srtt + (1/alpha) * r;
}
info->rto = MICROSEC * max(0.2, srtt + max(granularity / MILLISEC, 4*rttvar));
```

This retransmission timer calculation is an exact implementation of the RFC's "Computing TCP's Retransmission Timer" with one important modification. The minimum value for **rto** has been set to **0.2** seconds instead of **1** second. This value is also used for the TCP implementation in Linux. The specification says that "whenever RTO is computed, if it is less than 1 second then the RTO SHOULD be rounded up to 1 second" but also that it may be shown that "a smaller minimum RTO is acceptable or superior."[PA00]

The retransmission timer is using the linked list (subsection 3.7.3) to compare the actual time of execution with the time of transmission of the first unacknowledged packet during every iteration of a loop, which is done by Listing 3.10 .

Listing 3.10: Retransmission Timer 2

```c
diff=(double)head->time-(double)get_time()+(double)(info->rto);

if((diff < 0.0) && (info->no_timeout == 0)) {

        printf("timeout while waiting for packet %lu\n",info->expected_mseq);
        /* no further timeout can occur until new packet is sent */
        info->no_timeout = 1;
        info->packets_sent = info->expected_mseq;
        /* restart receiving window */
        if(info->recvbuf > 0) {
                info->recv_window = info->recvbuf;
        }
        else {
                /* prevent deadlock */
                info->recv_window = 1000000000;
        }
}
```

If the transmission time, which is stored in the head of the list, is larger than the actual time plus the calculated time for the retransmission timer, a timeout is indicated. An additional variable is checked to prevent the same packet to timeout again before the retransmission itself takes place. This variable is called **no_timeout**. It is set to **1** if a timeout is detected and is set to **0** if a new packet is transmitted. Unless **no_timeout** is **0** there can be no further timeout, otherwise every loop iteration detects a timeout and triggers a retransmission until the new packet is sent. The intended behavior is to only retransmit once and then wait as long as the retransmission timer value represents. Only then a new timeout can occur.
As soon as a new timeout is detected, the parameter **packets_sent** is set to the value of the sequence number of the packet which triggered the timeout.
To prevent deadlocks from occurring, which can be caused when a packet is lost and the receiving window stored at the sender has reached a level which prevents new packets to be sent, the variable **recv_window** is set to its original value, so that the sender can retransmit packets. If the first packet in a communication is triggering a timeout, the sender has no knowledge of the size of the receiving window, so it is set to a predefined value, which is overwritten as soon as acknowledgement arrive.

3.7.5 Interprocess Communication

As the sending process of this project is based on two processes running simultaneously, an important part of the development was the ability to access certain variables from both processes. Among these variables are the bitrate, the round-trip-time and the number of already sent packets. These variables were stored using shared memory so that both processes have the same variables. The most challenging part was the management of the linked list (subsection 3.7.3) for the retransmission timer (subsection 3.7.4). The difficulty in this case is based on the fact that acknowledgements and new packets are being handled by different processes of the sender.
To be able to handle an infinite number of sent packets a linked list is used to store all transmission times of unacknowledged packets. As there is no obvious solution to create a linked list shared between multiple processes, there have been several different solution attempts. One of these attempts is based on pipe communication. The idea is to send a signal containing the packet ID from the sending to the receiving process after every packet's transmission. The receiving process stores the time a signal has been received with the according packet ID. So the receiving side has an up-to-date linked list at all times.

Listing 3.11 receives the sequence number of the sent packet and in case the new packet ID is smaller than previously stored IDs, the linked list is cleared completely.

Listing 3.11: Interprocess Communication

```c
if((n = read( fd[0], pipe_buffer, 100)) > 0) { /* child reads from pipe */
        actual = get_time();
        sscanf (pipe_buffer,"%lu",&pipe_mseq);

        if(pipe_mseq <= info->expected_mseq) {
                while (head != NULL) {
                        tmp = head->next;
                        free(head);
                        head = tmp;
                }
                tail = NULL;
        }

        /* addition of time into the linked list*/
        node = (llnode *)malloc(sizeof(llnode));
        node->time = actual;
        node->next = tail;
        tail = node;
        if(head == NULL) {
                head = node;
        }
}
```

This version worked well in small tests, but created errors if there were too many packets sent in a small time frame. Therefore it was replaced by a linked list handled completely by the sending process. The receiving process is informing the sending process of the arrived acknowledgements using shared memory. This shared memory is accessible from both processes at all times instead of limiting the access to only one process. Other means are used to ensure the integrity of these data values.

3.8 Buffer

There are two different buffers used in the program. The first buffer is the buffer of the socket and is offered by the operating system. In this document this buffer is referred to as the network buffer. It receives packets from the network interface and stores them until the application retrieves the packet. The program has no real influence on this buffer as it gives one packet after another to the application. The second buffer is called the application buffer, because it is implemented in the application itself. It is used to store the data of the packet and other important information, but not the packet itself. This buffer is under complete control of the application, its only boundaries are based on the memory consumption of the complete program.

3.8.1 Network Buffer

To prevent an overflow of data at the receiver, the sender stores a value which represents the size of the network buffer. There are different ways to obtain this value, but Listing 3.12 is using the function ”getsockopt()” instead of reading ”/proc/sys/net/core/rmem_default” to get the buffer size, although the value should be identical [Get].

Listing 3.12: Network Buffer

```
/* new function to get the buffersize at the receiver, returns buffersize in
    bytes */
int udp_getrecvbuf(socket_udp *s)
{
        int rcvbuf=0;           /* Receive buffer size */
        socklen_t optlen;       /* Option length */
        optlen = sizeof rcvbuf;
        getsockopt(s->fd,SOL_SOCKET,SO_RCVBUF,&rcvbuf,&optlen);
        return rcvbuf;
}
```

This value is transmitted to the sender in every packet (just in case a packet is lost), the sender uses it and adds or subtracts the size of a packet each time a packet is sent or acknowledged. As soon as the receiving window is smaller than the packet size, it waits until the receiving window has been increased.
TCP has a serialized handshake, the program developed in this thesis does not have that, which means, it sends a couple of data packets to the receiver before the first packet is acknowledged, so it does not have the buffer size at that point. For that reason a default value (1000000000 byte) is used as the buffer size until it is overwritten by the true value received within an acknowledgement. The properties of this TFRC implementation should prevent the sender to overflow the receiver by engaging the *Slow Start mode*.

3.8.2 Application Buffer

For the program to handle out of order packets, it has to be able to store all relevant information. Among the information which is necessary is the packet ID, the data itself, the size of the data and if this packet is the last packet of the transmission. Other values like the round-trip-time have to be used immediately for necessary calculations. The buffer elements have been defined in Listing 3.13 using a linked list..

Listing 3.13: Application Buffer 0

```
typedef struct buffer_for_data {
        u_int32 packet_id;
        char data[DATA_SIZE];
        int data_size;
        int end_flag;
        struct buffer_for_data *next;
} data_buffer;
```

Listing 3.14 is responsible to store the data of a packet in the buffer. If an out of order packet arrives (with higher sequence number) and there is space in the buffer, a new node for the linked list is created and is filled with all necessary data of the packet. This node is then inserted in the right place of the linked list.

Listing 3.14: Application Buffer 1

```
if(packets_in_buffer < buffer_size) {
        /* insert new data into the buffer */
        /* addition of new node into the linked list*/
        node = (data_buffer *)malloc(sizeof(data_buffer));
        node->data_size = data_size;
        node->end_flag = end_flag;
        if(node->end_flag == 1)
```

```
                printf("EndFlag stored\n");
        memcpy(node->data, packet->data, data_size);
        node->packet_id = mseq;
        if(head == NULL) {
                head = node;
                tail = node;
                packets_in_buffer++;
        } else {
                if(node->packet_id > tail->packet_id) {
                        tail->next = node;
                        tail = node;
                        packets_in_buffer++;
                } else {
                        if(node->packet_id < head->packet_id) {
                                node->next = head;
                                head = node;
                                packets_in_buffer++;
                        } else {
                                tmp=head;
                                while(node->packet_id > tmp->packet_id) {
                                        if (node->packet_id < tmp->next->
                                            packet_id) {
                                                node->next=tmp->next;
                                                tmp->next=node;
                                                packets_in_buffer++;
                                                break;
                                        } else {
                                                tmp = tmp->next;
                                        }
                                } /* end loop for packet insert */
                        } /* end head check */
                } /* end tail check*/
        } /* end head check */
} /* end buffer_size check */
```

After a new packet is received, the program is checking if the following sequence number is already stored in the application buffer (Listing 3.15).
If the program is detecting a packet within a buffer, it checks if the ID of the packet stored in the first element of the linked list is equal to the value of the variable **expected_mseq**. If this is the case, the data of the stored packet is written to the file in the intended position and the packet data is removed from the buffer by updating the linked list.
This is repeated until the buffer is either empty or the sequence number in the buffer is higher than the expected packet ID.

Listing 3.15: Application Buffer 2

```
/* check if expected_mseq is already in buffer */
while(packets_in_buffer > 0) {
        if(expected_mseq == head->packet_id) {
                if(head->end_flag==1) {
                        char received_data_for_last_file[head->data_size];
                        memcpy(received_data_for_last_file, head->data, head->
                            data_size);
                        printf("EndFlag received\n");
                        info->end_receiver=1;
                        fwrite(received_data_for_last_file, sizeof(char), head->
                            data_size, file_pointer);
                } else {
                        memcpy(received_data, head->data, head->data_size);
                        fwrite(received_data, sizeof(char), data_size,
                            file_pointer);
                }
```

```
		if(head == tail) {
			tmp = head;
			head = NULL;
			tail = NULL;
			free(tmp);
		} else {
			tmp = head->next;
			free(head);
			head = tmp;
		}
		expected_mseq++;
		packets_in_buffer--;
	}
	else {
		break;
	}
}
```

The sender is keeping track of the number of packets which have sent but are still unacknowledged. This number of packets is not supposed to get bigger than the number of packets which the other side is able to store, otherwise these packets could just get lost at the receiver. The behavior of the sender is similar to that handling the network buffer (subsection 3.8.1), as soon as packets are sent it is subtracted from a counting variable and if new acknowledgements arrive, the variable is increased again, allowing the sender to send new packets.

3.9 Improvements

Additionally to the reliability and the MulTFRC bandwidth calculation, several other modifications have been added to the original implementation. These additions include a file transfer and a program termination in certain cases.

3.9.1 File Transfer

The implementation developed in this thesis includes a real file transfer which distinguishes it from the code it is based on.
Due to the fact that this protocol is simulated on the application layer, this file transfer underlies a certain restriction, which is based on the file reading process. With certain file types it is not possible to read and afterwards store it to achieve the original file. This problem is not based on the transmission from one PC to another, but because of the properties of the file reader in C.

3.9.2 Termination

The original code, which this implementation is based upon, has two ways for correct termination.

- Time: As an additional parameter in the command prompt it is possible to state a running time for the program, so that if it reaches this time it terminates.
- Maximum number of packets: A maximum number of packets is specified in a configuration file. If the number of packets sent has reached this number the program terminates.

Both termination reasons where developed for a program used just for testing purposes. As this reliable TFRC implementation has the purpose to transfer files, it had to be changed. Two new clauses had to be added, replacing the original ones:

- Time: In comparison to the original code, this implementation does not check the total running time of the program, but the time between two subsequent packets. If the sender or the receiver has not received a packet from the other side for an in the configuration file specified time (see section 3.3), it determines that either the network is out of order or the other side has stopped sending and it terminates. This prevents a process from running an infinite amount of time in case the other side has crashed. But as an improvement to the original code, it is possible to transmit files over a large period of time. This termination condition is checked at both sides of the transmission.
- End_Flag: The sender sets the **end_flag** in the last packet to **1** and as soon as the receiver has sent the acknowledgement to this packet, the receiver terminates. The sender terminates as soon as all necessary packets have been acknowledged. This possibility describes the case that the file transfer has been successful and the processes are terminating correctly. In case that the last packet is received before other packets, the receiver stores the **end_flag** until all other packets are received.

While the **end_flag** termination is straight forward at the receiver, it may cause problems at the sender. This is due to the fact that the receiver stops immediately its operation after sending the acknowledgement for the last packet. If this acknowledgement is lost, the sender will retransmit the last packet several times before it realizes that the receiver has stopped working and therefore the sender will also terminate. This behavior does not cause any problems in the data transmission itself, but may cause unwanted error messages at the sender. Once again, this problem is based on the lack of a real handshake like TCP uses.
Another problem is caused by the C-function which has been used to listen for the incoming packets. As this function is blocking the process until a packet arrives, an additional thread had to be used at the receiver to check if the transmission is still in progress (at the sender there is already a second thread which is taking care of this task). Listing 3.16 implements the thread which checks if new packets have arrived once every interval, which is specified by **MAX_WAIT**. Afterwards this thread sleeps for the specified time period. This thread distinguishes three different cases:

- The **end_flag** has been received: A received **end_flag** suggests an ended transmission and therefore the program can be terminated.
- New packets have arrived since the last check: The receiver continues as there is still some ongoing communication
- No new packets have arrived since the last check: As there have been no new packets that have arrived in a specified time frame, the program assumes that either the sender or the network has stopped its operation and therefore the receiver is also terminated.

Listing 3.16: Termination Control

```
if(pid == 0) {
        while(termination_control == 0) {
                wait(MAX_WAIT * MICROSEC);
                if(info->end_receiver == 1){
                        // terminate program
                        kill(getpid(), SIGKILL); // kill process
                        termination_control = 1;
                }
                if( last_packet_for_termination < info->sendpkt) {
                        // continue
                        last_packet_for_termination = info->sendpkt;
                } else {
                        // terminate program
                        if(kill(getppid(), SIGKILL) < 0)
                                printf("Error\n");
                        printf("Timeout because no packet has arrived in %u
                            seconds\n", MAX_WAIT);
                        termination_control = 1;
                        fclose(h_file);
                }
        }
}
```

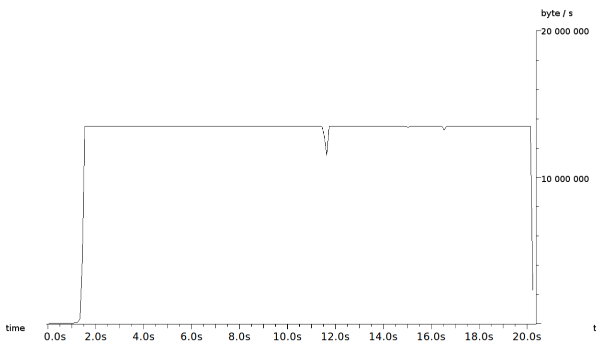
Chapter 4

Testing

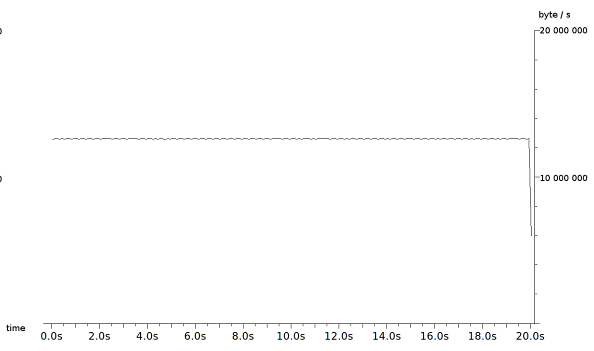
The program has been tested under different conditions and with different parameters. These conditions contain various network simulations using Linux traffic control. Linux traffic control allows a user to specify parameters like bandwidth, delay and probability of packet loss in a controlled environment.
The program itself is executed with different values for MulTFRC, and different buffer sizes for the receiver. This section has been split into two parts. The first part describes the effect of multiple connections on the data transmission speed while the second part is covering the differences of TCP and TFRC/MulTFRC competing for available bandwidth.

Figures: For all figures the x-axis represents the time while the y-axis is the flow's throughput. The throughput is measured every tenth of a second. All bandwidth tests are conducted at the receiver side. Therefore only the actual transmission bandwidth is considered instead of the bandwidth used by the sender, which may be higher than the available bandwidth of the network. The program used to capture packets is called Wireshark [Wir] and is used to produce the figures.

Remark: The program is using UDP to simulate TFRC/MulTFRC and this fact is resulting in the possibility of special handling of the flows in comparison to TCP. UDP is considered harmful to competing connections as it has no congestion control. Therefore several routers and other devices limit the throughput of UDP flows to protect responsive flows (like TCP). Therefore all tests with TCP and TFRC connections competing against each other are conducted with limited bandwidth at the end-hosts, so that the amount of traffic is not reaching the link speed.
All results representing 1-MulTFRC are conducted with the standard TFRC formula instead of the MulTFRC formula.



(a) MulTFRC



(b) TCP

Figure 4.1: 100-Megabit-LAN

4.1 Speedup

This section covers the possible improvement of transmission times if MulTFRC is used. Therefore several network conditions are simulated in which TFRC and MulTFRC (with different values) is tested. Besides the possible speedup, it is also demonstrating the difference in behavior in comparison to TCP.

4.1.1 100-Megabit-LAN

The program is executed without any additional constraints in a 100-Megabit-Local-Area-Network. The round-trip-time of this network is 0.2ms and the buffer is able to store up to 2000 packets.
As already one TFRC flow is using all available bandwidth, there is no possible speed improvement with the usage of MulTFRC. Therefore 4.1a can represent both a TFRC and a MulTFRC flow. The small drop in the throughput after eleven seconds is caused by random congestion at the router.
Using a more aggressive MulTFRC flow has no (positive) influence on the transmission in an 100-Megabitnetwork.
A TCP connection cannot achieve a higher throughput (as TFRC/MulTFRC is already using all available bandwidth), which can be seen in 4.1b, but it has still the major advantage that the speed in the beginning of the transmission is increasing faster than with TFRC. For the TFRC connection it takes over one second to achieve the maximum speed.
This problem is based on the way the implementation is increasing slowly the sending speed until a first packet loss occurs. From that point the result of the TFRC/MulTFRC formula is replacing the Slow Start mechanisms. This behavior is not covered in the TFRC specification. A possible improvement of this TFRC implementation could be achieved by modifying the Slow Start behavior. Independent of the network situation, the interval between the first and the second packet is about half a second. It would be possible to decrease the waiting time between the first two packets and to achieve a higher speed in the beginning. Another speedup could be gained by increasing the sending rate faster but this could also lead to more packet loss in some situations.

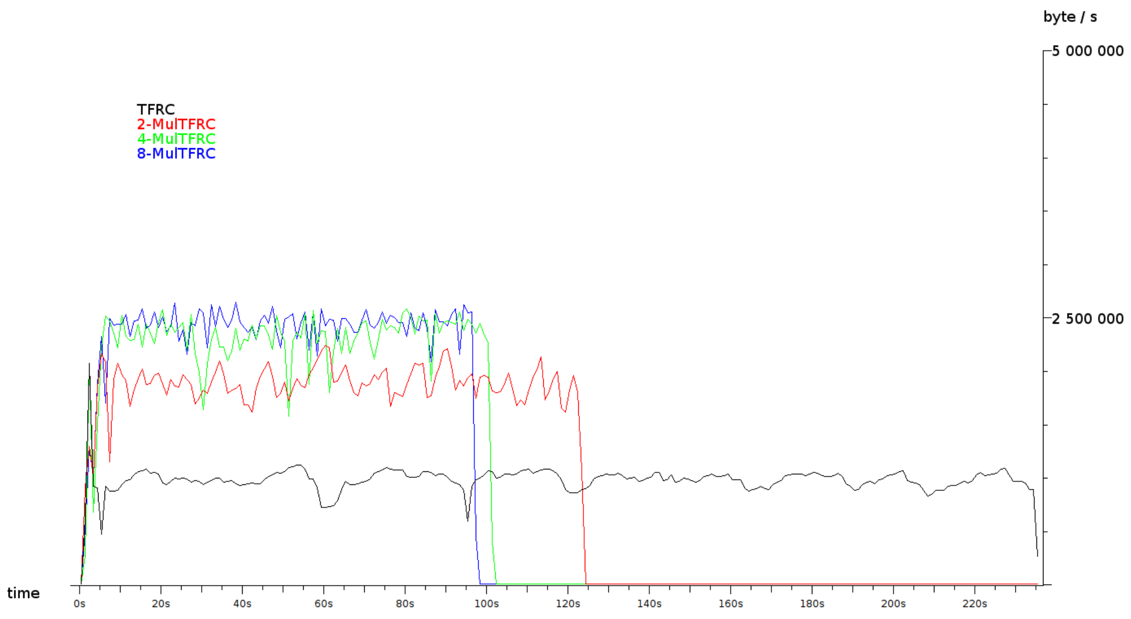


Figure 4.2: TFRC and 2-, 4- and 8-MulTFRC in a 32 MBit-LAN with a 20ms delay

4.1.2 32MBit-20ms Delay

In this example a network is simulated which has specific properties. The maximum bandwidth is set to 32 MBit/s and one percent of all packets is supposed to be lost. This packet loss is not random but affects approximately every hundredth packet. Additional to the default delay of the LAN, every packet has a delay of 20ms. All these conditions are also valid for the acknowledgements. The following code is used to simulate the network:

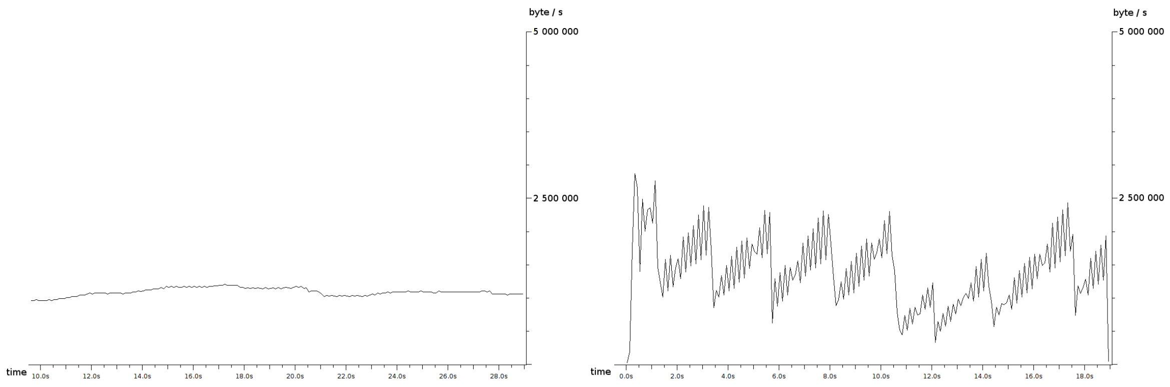
Listing 4.1: Setting the network environment

```
sudo tc qdisc add dev eth0 root handle 1:0 netem delay 20ms
sudo tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 32mbit buffer 300000
    limit 30000
sudo tc qdisc add dev eth0 parent 10:1 red limit 300000 min 3000 max 240000 avpkt
     1488 burst 60 probability 0.01
tc qdisc
```

Five different tests have been conducted to be aware of the speedup achieved by the usage of MulTFRC with higher values. Figure 4.2 shows the transfer of a single, identical file with MulTFRC which behaves like one, two, four and eight TFRC connections. The transmission time of a file, which has to be split into 153426 packets, can be decreased from 235 seconds it takes for one single flow to 125, 103 and 98 seconds it takes for various values for MulTFRC. These results are shown in Table 4.1. MulTFRC is able to provide a faster transmission than TFRC and also TCP could provide as it is using available bandwidth more efficiently than other competitors. While there is a nearly linear speedup with 2-MulTFRC (which behaves like two TFRC flows) in comparison to only a single flow, higher values for MulTFRC have much less influence. The transfer speed of the file cannot be increased significantly with a value higher than four.
Even if by the usage of much higher values for MulTFRC a higher sending rate would be the result of the formula, several parameters like the network or the application buffer are preventing the program from further speedup. These parameters may also cause the loss of stability with higher transmission

n-MulTFRC	average time of transmission in seconds
1	235
2	125
4	103
8	98
12	95

Table 4.1: Comparison of transmission times with different values for MulTFRC



(a) TFRC (b) TCP

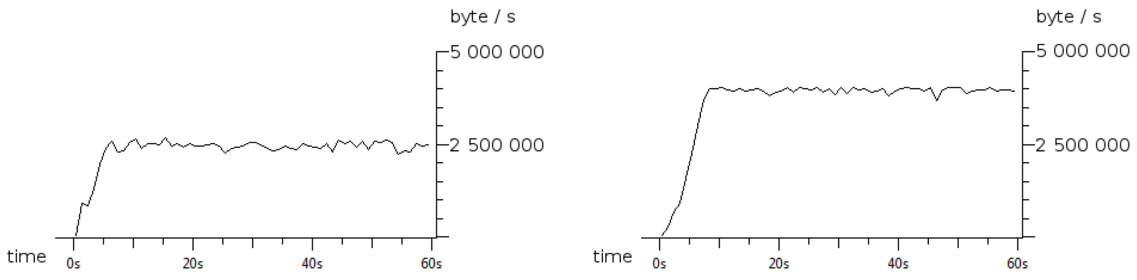
Figure 4.3: 32-Megabit-LAN with a 20ms delay

rates. An important feature of TFRC is to have a throughput similar to TCP, but with a smoother sending rate. Therefore Figures 4.3a and 4.3b show the sending rate of a TCP flow and a TFRC flow simulated in this network environment. To have a good comparison, only twenty seconds of each test are shown. Because TFRC is testing the boundaries of the network slower, the behavior in the beginning is different than after a couple of seconds. Therefore Figure 4.3a shows the TFRC flow starting after ten seconds. TCP behaves always the same therefore there is no need to adapt the starting point of Figure 4.3b.
The tests have shown that the TFRC flow has a much smoother sending rate than the TCP flow without any of its pikes.

Improvement In the test system the network buffer size is 112640 bytes, while a packet has the size of 1460 bytes. As the program is only transmitting packets which can be buffered at the receiver, it is allowed to send 77 packets at maximum before receiving an acknowledgement. If the round-trip-time is large, this barrier prevents a higher transmission rate. This buffer size is chosen for the case that every packet which is unacknowledged has to be stored in the network buffer but this case is quite improbable. Under normal circumstances, only a part of these packets are in the network buffer of the receiver. The other packets may be in the network itself or at the receiving application. It is also possible that the packets have already been received and that its acknowledgements are either on the way back or already in the network buffer of the sending process. Therefore a possible speed improvement can be gained by sending more packets than the network buffer can store.
The negative aspect of this proposition is that if the network buffer of the receiver is full, all incoming packets have to be dropped and therefore afterwards retransmitted. By allowing the sender to transmit

n-MulTFRC	average time of transmission in seconds					
	1 * buffer	2 * buffer	3 * buffer	4 * buffer	5 * buffer	10 * buffer
8	98	71	65	67	68	69
12	95	69	65	67	68	74

Table 4.2: Comparison of transmission times with increased buffer sizes

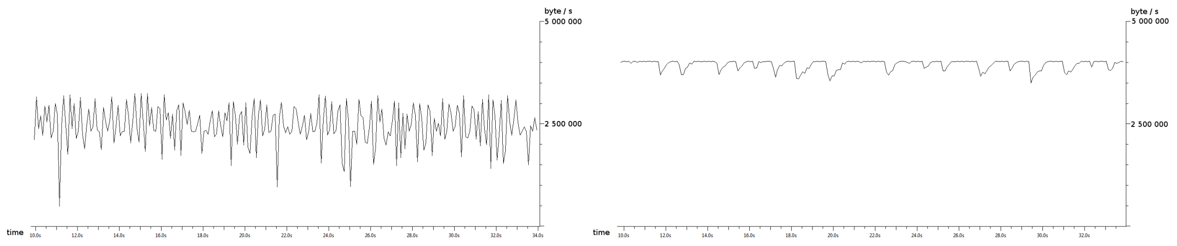


(a) reliable 8-MulTFRC (b) unreliable 8-MulTFRC

Figure 4.4: 32-Megabit-LAN: Comparison of reliable and unreliable 8-MulTFRC

more packets than the buffer can store an overall speed improvement can be gained. But it is not wise to allow too many packets in the network at a certain time. Table 4.2 demonstrates that too many packets at once are resulting in a longer file transmission time due to the fact that there have to be much more retransmissions. This is caused by many lost packets which have arrived at the receiver but cannot be stored in the network buffer (see subsection 3.8.1).

To investigate the influence of the implemented windows, reliable MulTFRC is compared with unreliable MulTFRC. In the simulated network the influence of reliability for a single TFRC flow is marginal. The file transmission time is nearly identical (235 for reliable TFRC, 232 seconds for unreliable TFRC). The difference can be explained as about one percent of all packets are lost and need to be retransmitted to achieve reliability. The windows have no significant influence on the flow's rate.
Reliability has a significant influence on the transmission of 4-MulTFRC. The file transmission time is decreased from 103 to 83 seconds if no reliability is implemented. As the retransmissions of lost packets are only responsible for a limited time increase, the remaining time is caused by the windows. Figures 4.4a and 4.4b show the comparison of reliable and unreliable 8-MulTFRC. It can be seen that the unreliable variant is more stable and has a significant higher throughput. Figures 4.5a and 4.5b show twenty second time frames of these flows. Without the implementation of reliability the file transmission time decreases from 98 to 63 seconds.



(a) reliable 8-MulTFRC (b) unreliable 8-MulTFRC

Figure 4.5: 32-Megabit-LAN: Comparison of reliable and unreliable 8-MulTFRC

4.1.3 64MBit-10ms Delay

The network simulated in this example has significant differences to the network in subsection 4.1.2. The maximum bandwidth is doubled to 64 MBit/s and the additional delay of 10ms is used for all packets. Similar to the other example, one percent of all packets should be lost. All these conditions are valid in both directions, therefore also affecting the acknowledgements. The following code is used to simulate the network:

Listing 4.2: Setting the network environment

```
sudo tc qdisc add dev eth0 root handle 1:0 netem delay 10ms
sudo tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 64mbit buffer 300000
    limit 30000
sudo tc qdisc add dev eth0 parent 10:1 red limit 300000 min 3000 max 240000 avpkt
     1488 burst 30 probability 0.01
tc qdisc
```

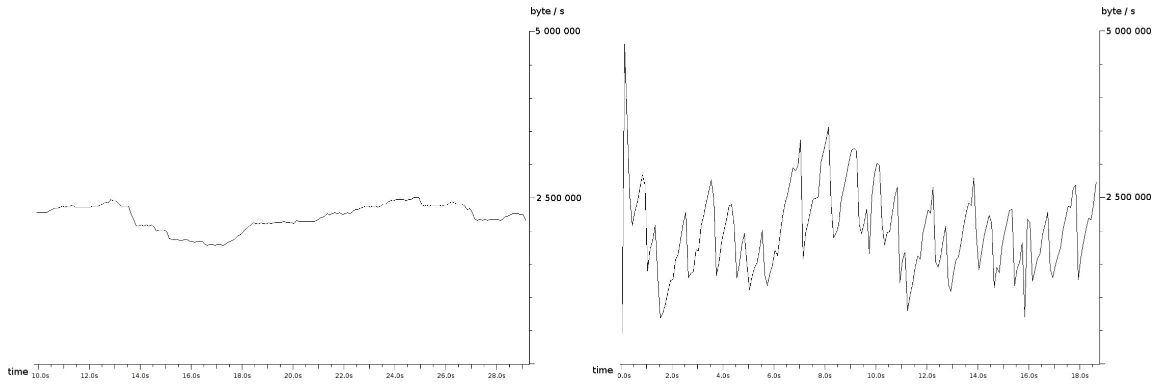
n-MulTFRC	average time of transmission in seconds
1	122
2	69
4	54
6	52
8	51
12	51

Table 4.3: Comparison of transmission times with different values for MulTFRC

Table 4.3 shows that it would be wise to use at least the value of six to eight for MulTFRC in this network environment for the best result. Similar to subsection 4.1.2, where a higher value for MulTFRC had no positive influence, it is valid in this case that the only way to gain a better result is to allow the program to send more packets than the other side can handle at once which is shown in Table 4.4. This could give the program another significant speedup which would be otherwise not achievable. Figures 4.6a and 4.6b demonstrate that in these network conditions TFRC is able to provide a much smoother sending rate than TCP does, while keeping at least the same throughput.

n-MulTFRC	average time of transmission in seconds					
	1 * buffer	2 * buffer	3 * buffer	4 * buffer	5 * buffer	10 * buffer
8	52	38	35	35	34	53
12	52	38	34	34	34	53

Table 4.4: Comparison of transmission times with increased buffer sizes



(a) TFRC (b) TCP

Figure 4.6: 64-Megabit-LAN with a 10ms delay

4.1.4 64MBit-4ms Delay

To investigate how certain parameters influence the transmission rate, the network of subsection 4.1.3 is used, but the delay is decreased from 10ms to 4ms. The following code is used to simulate the network:

Listing 4.3: Setting the network environment

```
sudo tc qdisc add dev eth0 root handle 1:0 netem delay 4ms
sudo tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 64mbit buffer 300000
    limit 30000
sudo tc qdisc add dev eth0 parent 10:1 red limit 300000 min 3000 max 240000 avpkt
     1488 burst 30 probability 0.01
tc qdisc
```

n-MulTFRC	average time of transmission in seconds
1	61
2	34
4	31
6	31
8	31
12	31

Table 4.5: Comparison of transmission times with different values for MulTFRC

It can be seen that using a smaller value as the additional delay has significant influence on the results. In comparison to the speedup achieved in subsection 4.1.3 with different values for MulTFRC, the

n-MulTFRC	average time of transmission in seconds		
	1 * buffer	2 * buffer	3 * buffer
8	31	31	31
12	31	31	32

Table 4.6: Comparison of transmission times with increased buffer sizes

best results with a small delay network can already be obtained by a smaller value which is shown in Table 4.5. Table 4.6 shows that having a network with a low delay prevents the program from achieving higher transmission rates achieved by the improvement proposed in subsection 4.1.2. The reason for that behavior is that packets are traveling through the network faster than in the previous examples. Therefore the amount of data in the network (sent but not acknowledged) is smaller which prevents the two implemented windows from interfering with the transmission.

4.1.5 100MBit-4ms Delay

In comparison to subsection 4.1.4 the bandwidth is increased in this example.

Listing 4.4: Setting the network environment

```
sudo tc qdisc add dev eth0 root handle 1:0 netem delay 4ms
sudo tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 100mbit buffer 300000
    limit 30000
sudo tc qdisc add dev eth0 parent 10:1 red limit 300000 min 3000 max 240000 avpkt
     1488 burst 30 probability 0.01
tc qdisc
```

number of flows	average time of transmission in seconds
1	62
2	34
4	28
6	27
8	27
12	26

Table 4.7: Comparison of transmission times with different values for MulTFRC

Table 4.7 shows the transmission times of the same file with different values for MulTFRC. The influence of the higher bandwidth in comparison to subsection 4.1.4 is visible by observing the transmission times when higher values are used for MulTFRC while there is no difference for TFRC or 2-MulTFRC. Using large values as a buffer multiplicator is resulting in a wide spread of results, ranging from the transmission times of tests with lower values to very high transmission times. The corresponding results are shown in Table 4.8.

An important requirement for using a high number of TFRC flows as which MulTFRC should behave like is the available bandwidth. Without free bandwidth an increase of the MulTFRC value has no effect on the overall transmission speed. The comparison between the network simulated in subsection 4.1.5 and subsection 4.1.4 shows that in an otherwise identical network an increased

n-MulTFRC	average time of transmission in seconds					
	1 * buffer	2 * buffer	3 * buffer	4 * buffer	5 * buffer	10 * buffer
12	26	25	24	26	36	100

Table 4.8: Comparison of transmission times with increased buffer sizes

bandwidth allows an improvement with higher values while there is no change with lower values. The results are shown in Table 4.9.

n-MulTFRC	64 Mbit	100 Mbit
1	61	62
2	34	34
4	31	28
6	31	27
8	31	27
12	31	26

Table 4.9: Comparison of transmission times in 64 Mbit and 100 Mbit networks

4.2 TCP vs. TFRC/MulTFRC

As the above statistics have shown, an advantage can be gained by using multiple connections instead of only one in the transmission of data. MulTFRC can use the bandwidth more efficiently than only a standard TFRC flow could. But in these scenarios the data transmission is the most dominant flow in the network, but far more interesting is the question, how MulTFRC behaves if there are multiple other flows present which all compete for the same limited available bandwidth. As UDP's sending rate is not adapting due to congestion, it is not a viable competitor, therefore only TCP will be compared with TFRC/MulTFRC.

As described previously, there may be some devices which provide TCP with a higher throughput than UDP, although the sending rate is not responsible for that difference. Figure 4.7 shows that TCP is able to use about 2/3 of the available bandwidth, while TFRC/MulTFRC is only using the rest in such an environment. To be more precise, TCP achieves an average throughput of 8800000 byte per second, while TFRC (like UDP and MulTFRC) achieves around 4100000 byte per second. Therefore a network is simulated which does not use all available bandwidth in the intermediate routers by the usage of Listing 4.5. The bandwidth is limited to 32 Mbit/s and an additional delay of 10ms is introduced in both directions.

Listing 4.5: Setting the network environment

```
sudo tc qdisc add dev eth0 root handle 1:0 netem delay 10ms
sudo tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 32mbit buffer 300000
    limit 30000
sudo tc qdisc add dev eth0 parent 10:1 red limit 300000 min 3000 max 240000 avpkt
     1488 burst 30 probability 0.01
tc qdisc
```

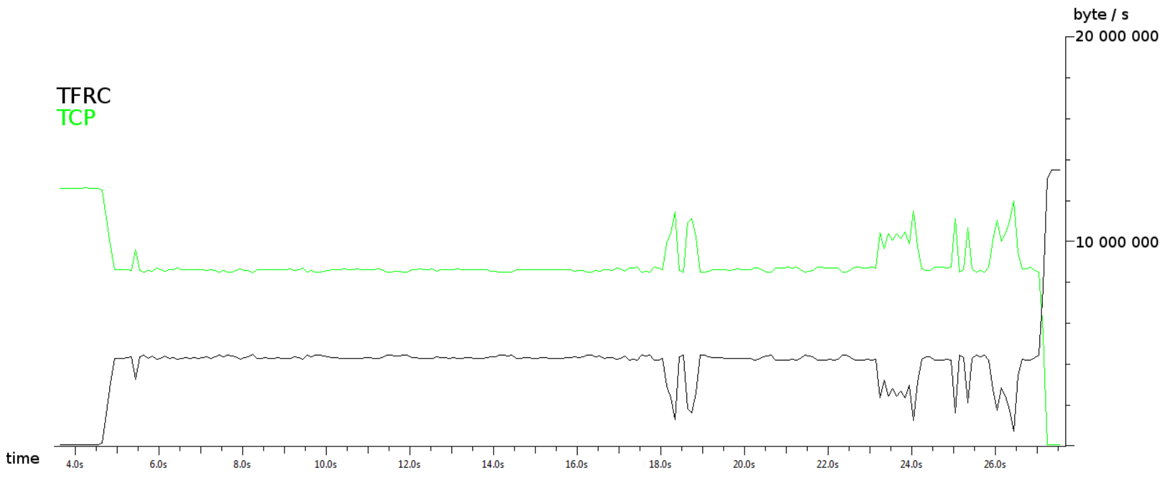


Figure 4.7: TFRC vs TCP in a 100 MBit-LAN

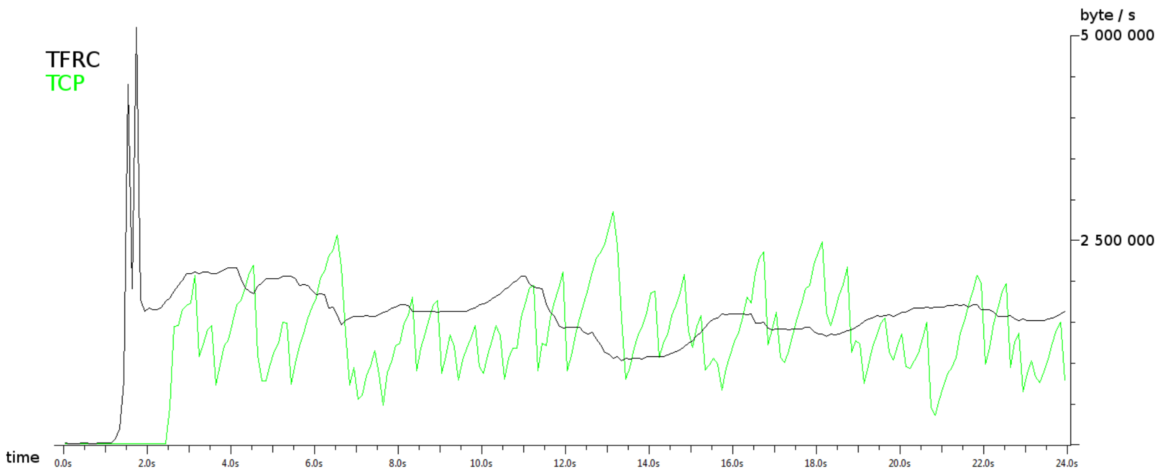


Figure 4.8: TFRC vs TCP in a 32 MBit-LAN with a 10ms delay

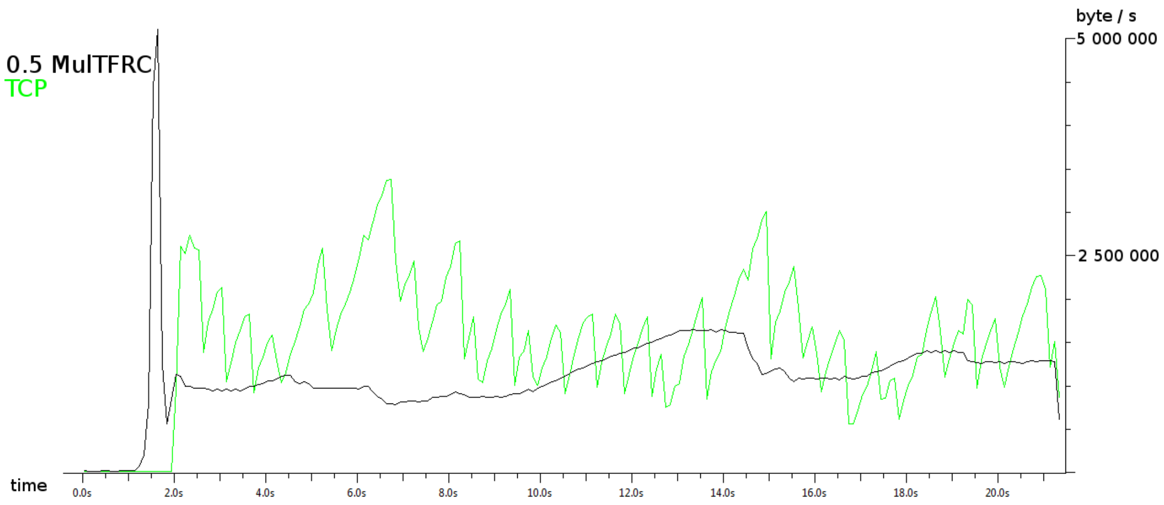


Figure 4.9: 0.5-MulTFRC vs TCP in a 32 MBit-LAN with a 10ms delay

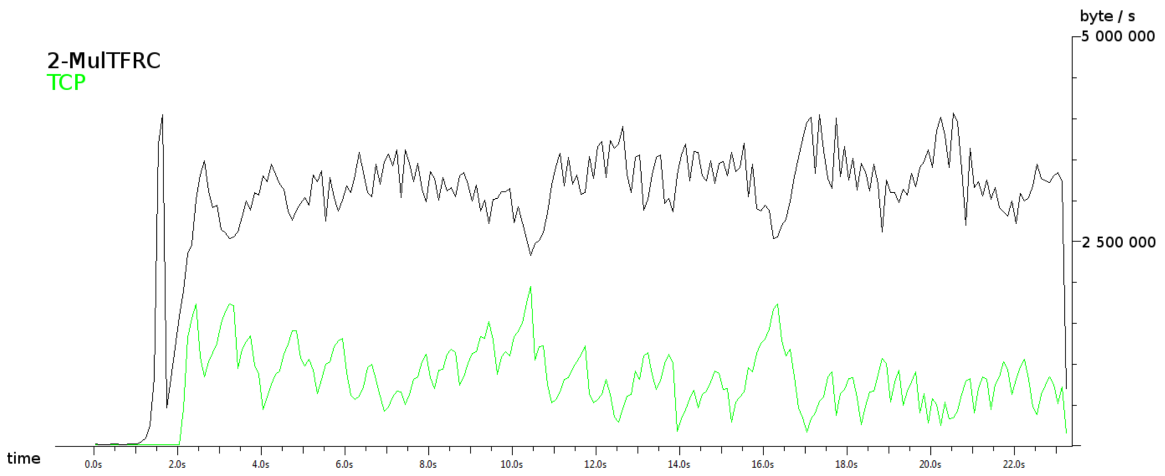


Figure 4.10: 2-MulTFRC vs TCP in a 32 MBit-LAN with a 10ms delay

Figure 4.8 shows the throughput of a single TFRC flow which gets a competitor after two seconds. It can be seen that a single TFRC flow is able to achieve a slightly higher throughput than TCP. To investigate this result more closely, the average throughput for twenty seconds is calculated, starting after three seconds. TFRC is transmitting over 1600000 byte per second on average while the average throughput of TCP is around 1410000 byte per second.
A special feature of MulTFRC is demonstrated in Figure 4.9, where 0.5-MulTFRC is competing with one TCP flow. 0.5-MulTFRC is a less aggressive TFRC flow. In this example, it is started two seconds before a TCP connection. While the behavior of this less aggressive flow is similar to TFRC in the beginning, this changes as soon as packet loss is encountered and TCP is also started. With this configuration TCP is less affected by the MulTFRC flow. TCP achieves an average throughput of 1630000 byte per second while MulTFRC achieves an average throughput of 1150000 byte per second.
By the usage of n-MulTFRC, with n bigger than one it is possible to gain a significant speedup in comparison to TCP. An aggressive flow, like 2-MulTFRC, is responsible for a slow transmission rate of TCP. By the usage of a n-MulTFRC flow which behaves like n TFRC flows, a main advantage of TFRC is lost as the connection is less stable. Figure 4.10 shows that certain flows can be prioritized in comparison to others. This may have significant advantages if important data has to be transferred in a busy network. In this specific example MulTFRC has an average throughput of 3210000 byte per second while TCP has an average throughput of 840000 byte per second.
Figure 4.11 shows the comparison of a 2-MulTFRC flow with two TCP flows. MulTFRC is able to achieve a higher throughput while still keeping at a “friendly”-level to TCP, which is the main goal behind MulTFRC. MulTFRC has an average throughput of around 2520000 byte per second while TCP achieves an average throughput of 1650000 byte per second.
Figure 4.12 shows that the simulation of 4-MulTFRC is slightly faster than four separate TCP flows. MulTFRC has an average throughput of around 2290000 byte per second while TCP achieves an average throughput of 1910000 byte per second.
The situation changes if eight connections are competing for the available bandwidth, which is demonstrated by Figure 4.13. In this example, the throughput of eight TCP connections summed up is faster than the throughput of MulTFRC behaving like eight TFRC flows. MulTFRC has an average throughput of around 2400000 byte per second, while TCP achieves an average throughput of 1820000 byte per second.

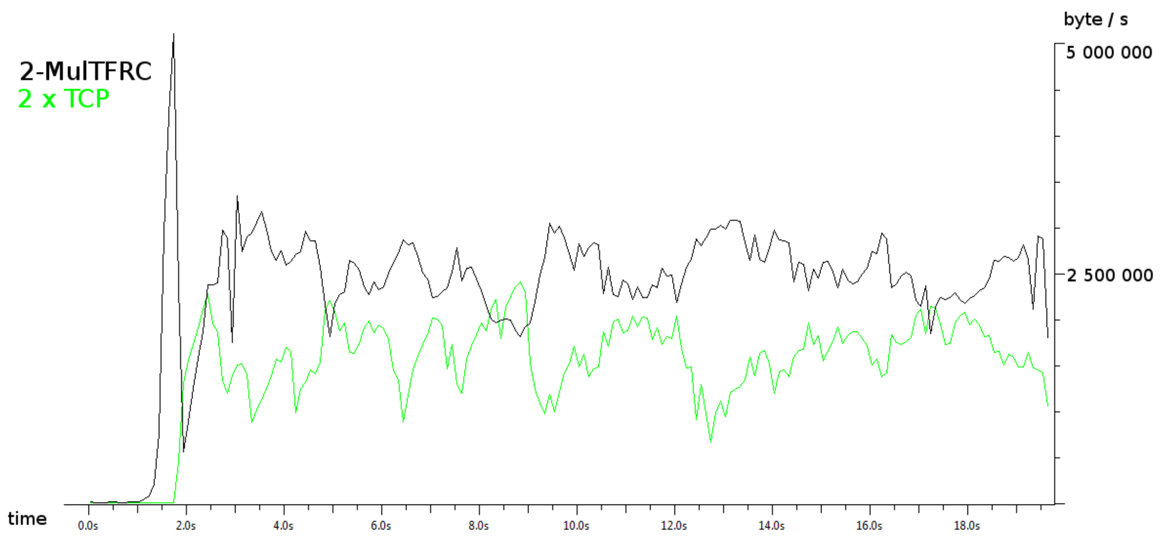


Figure 4.11: 2-MulTFRC vs 2x TCP in a 32 MBit-LAN with a 10ms delay

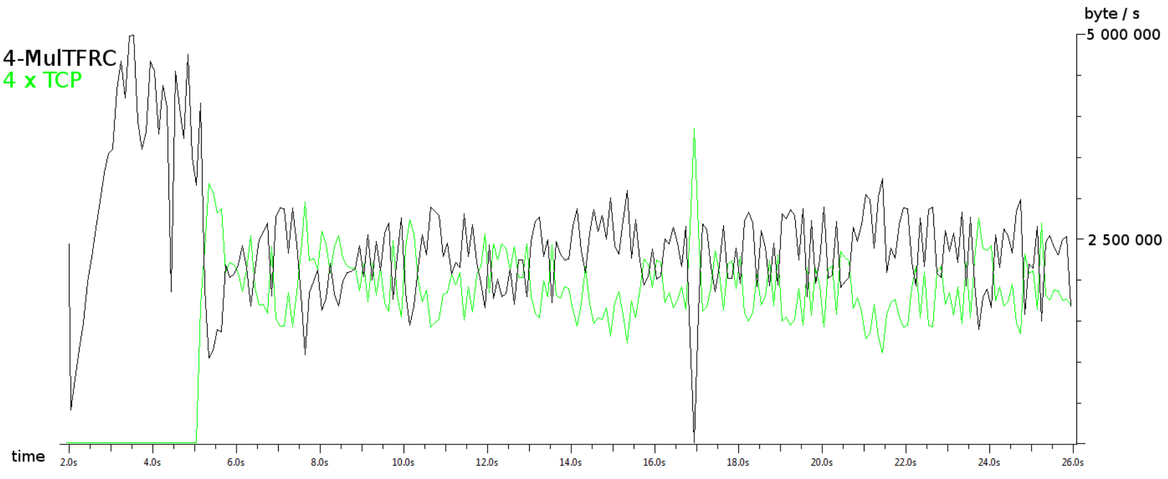


Figure 4.12: 4-MulTFRC vs 4x TCP in a 32 MBit-LAN with a 10ms delay

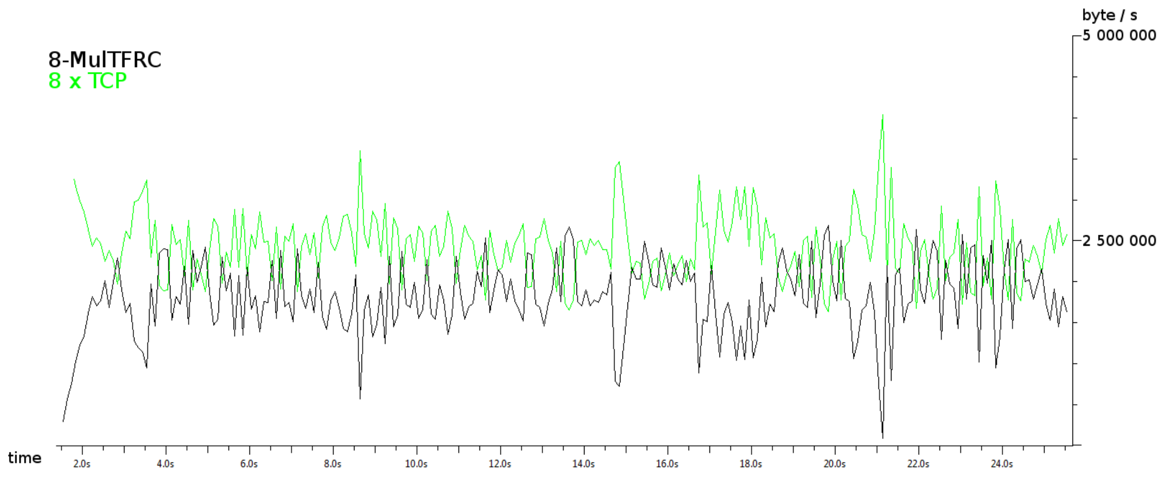
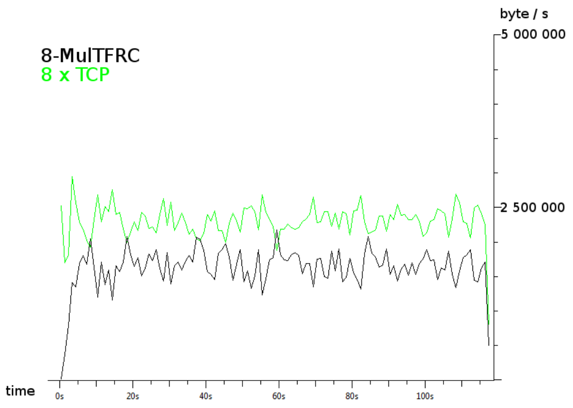
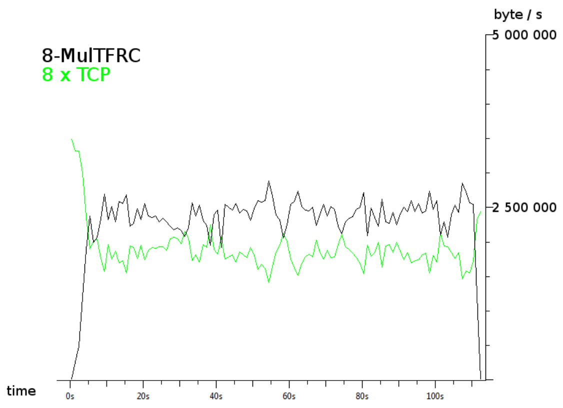


Figure 4.13: 8-MulTFRC vs 8x TCP in a 32 MBit-LAN with a 10ms delay



(a) 8-MulTFRC with normal buffer



(b) 8-MulTFRC with increased buffer

Figure 4.14: 8-MulTFRC vs 8x TCP in a 32 MBit-LAN

For further information about the reason why eight TCP connections are faster than 8-MulTFRC, additional tests have been conducted. The comparison of Figures 4.14a and 4.14b shows that the reason for the result of Figure 4.13 where TCP is faster than MulTFRC is based on the limiting factors. As discussed in example subsection 4.1.2, it is possible to improve the behavior of MulTFRC by allowing a greater transmission window than the network buffer of the application announces to the sender. With the multiplication of the buffer size with the value of three, 8-MulTFRC is transmitting faster than the sum of eight different TCP connections without significant more losses. The average throughput of MulTFRC increases to 2390000 byte per second while TCP has an average throughput of 1830000 byte per second.
This test has shown that the two implemented windows, for the network buffer (subsection 3.8.1) and the application buffer (subsection 3.8.2), have significant influence on the sending rate of the MulTFRC protocol. Therefore the flow is not as stable as if a less aggressive MulTFRC is used because in the latter case the windows may never limit packet transmissions.

All tests have shown that the TFRC/MulTFRC program is able to compete with TCP under fair circumstances. The only visible disadvantage is the slow beginning of a transmission, although that could be adapted with a modification of the Slow Start behavior. As soon as the bandwidth equation is replacing the Slow Start mode, it provides a fast but still fair sending rate. This sending rate is valid for one connection which acts like multiple TFRC flows, while every single TCP flow is reacting only on its own. With a low number as the MulTFRC parameter the transmission rate is much more stable in comparison to TCP, while with a higher number several factors can prevent such a stable behavior. As already mentioned, the bandwidth formula is only calculating an upper boundary for the actual sending rate but two windows have the possibility to prevent the transmission of packets. The comparison of reliable and unreliable MulTFRC has shown the significant influence of reliability on the transmission. As the tests with high values for the MulTFRC factor have been competing with multiple TCP flows, all available bandwidth is consumed more aggressively which may be another reason for the loss of stability.

As this program has first been developed using pure Go-Back-N and was only later modified to be able to request single packets, an interesting observation has been made between the two variants. The Selective-Repeat approach prevents the program from transmitting more packets than necessary and therefore in the end less packets are sent than by using the, based on the implementation, more simple solution. This approach may be slower than the more trivial Go-Back-N variant. This behavior is based on the fact that an additional buffer variable has been introduced, which can prevent the sender from sending new packets and so the sending process can be idle. The same constraint can be seen in the TCP implementation.

The idea of informing the sender of a modified buffer size of the socket can be limiting the problem of blocking variables and the program can therefore achieve a decreased transfer time by using a higher window size for packet transmission. But this changes can also have a negative effect. This negative effect is encountered if there is either a complete communication breakdown for a certain while or if the window size is chosen too large. If the window is chosen larger than the actual available buffer and a communication breakdown occurs, there are more packet losses to recover from. Although the retransmission timer is eventually taking care of that problem as it is triggering a Go-Back-N behavior, this will take longer to recover. If the Window Size is chosen too large from the beginning, the sender is constantly sending more packets than the receiver is able to store at once and therefore many of these packets are lost at the receiver and have to be retransmitted.

Chapter 5

Conclusion

5.1 Conclusion

Reliable MulTFRC is an implementation of MulTFRC, which is proposed as an extension to standard TFRC. While TFRC and therefore also MulTFRC is only a specification of a congestion control mechanism, the application developed in this document is simulating a complete protocol.
The congestion control is an implementation as it is specified in [FIH$^+$03] (TFRC) and [DW08] (MulTFRC). Both formulas are accessible in the program and the user is given the possibility to choose which function to use. The tests have shown that the new MulTFRC protocol is able to compete with its existing counterpart, namely TCP. While the throughput of one TFRC flow over a longer time frame is close to the throughput created by a TCP flow, it provides a much more stable sending rate, and at the same time it is TCP-friendly. This smooth sending rate is normally only a requirement for multimedia applications, but by behaving like multiple flows it is possible to increase the overall throughput. Therefore the changes proposed in MulTFRC make it a real alternative to TCP, although TFRC was not specified to be one. Moreover it is superior to TCP as it behaves like multiple connections, while it is still considered to be n-TCP-friendly [DW08]. So MulTFRC can prioritize certain connections by acting like a number of flows, on the other hand it can also provide less aggressive flows by giving the user the possibility of simulating less than one connection. The advantage of this lies in the capability of providing less aggressive congestion control for background connections which are not supposed to interfere with other flows. Possible examples, which could use such background flows, are update services or a BitTorrent client.
Reliability is achieved by applying different functionalities, ranging from a retransmission timer to the multiple acknowledging of single packets to trigger retransmissions. The retransmissions are realized by both the Go-Back-N and the Selective Repeat approach, which are used depending on the situation. While it would have been possible to use only one of these retransmission techniques tests have shown that it is recommended to use Go-Back-N instead of the more sophisticated alternative in case the retransmission timer is triggered. Also in some extreme cases, e.g. all sent packets or acknowledgements are lost, Go-Back-N is the only option. This is due to the fact that the retransmission timer is only used if the multiple acknowledgements for one packet have failed its goal which suggests that several packets are lost at once. It would take the program applying only Selective Repeat too long to recover from multiple packet losses.
To create a full protocol, it was necessary to implement some additional parts, which are not specified in [Jaa04]. The packets structure and the Slow Start method has been mostly provided by the code of Jörg Widmer [Wid00] and only changes necessary for MulTFRC and reliability have been

implemented. To provide Selective Repeat, a buffer had to be implemented to store out-of-order packets. As additional functionalities which are not connected to MulTFRC itself, a full tool capable of transfering files and a failsafe had been introduced to terminate the program in case of errors.

The speedup possible by behaving like more than one connection may be significant in a network without realtime competition but it is always limited by certain factors. If the round-trip-time in a network is too high the program has to stop sending packets because it cannot be assured that the receiver is able to store it. In this case increasing the MulTFRC factor has no influence on the speed but tests have shown that by allowing the sender to send more packets than the receiver can handle, an additional speedup can achieved. But this has to be done in an appropriate way, a too large value is contra productive. Using multiple connections is also only advantageous in terms of speed if there is enough available bandwidth.
An important study was the behavior of MulTFRC and TCP streams which compete over available bandwidth at the same time. In this case, the variables, which are preventing MulTFRC with a high value for n from gaining additional speedup, may not have influence on the achieved rate. section 4.2 showed that TFRC/MulTFRC is able to at least maintain the speed of TCP, although in many tests TFRC/MulTFRC achieved a slightly higher throughput than an equal number of TCP connections. An advantage is the possibility to give certain flows higher priorities by allowing them to behave as a number of TCP flows. All these services are not available using TCP.
As TFRC/MulTFRC is a rate-based congestion control scheme, the introduction of windows had a significant influence on the behavior of a flow. While the throughput equation is able to provide a stable sending rate, the responsibility of windows is the assurance that all sent packets can be stored in the network buffer or the application buffer. Therefore the windows may interfere with a stable transmission rate as it may prevent the transmission of packets.

5.2 Future Work

This tool provides congestion control and reliability for the UDP protocol. This functions have been implemented in the application layer, the proper way would have been in the Transport Layer. Only in this way the protocol discussed here could be a real alternative to UDP. Another advantage would be that it would not be necessary to use the same restrictions like for UDP.
To improve the program it would be advisable to modify the file transfer as its capabilities are very limited, therefore it would be an import improvement to implement a file transfer which is able to handle all existing file types.
This implementation uses a TFRC specification which is out of date. Several features have been added since, so as a possible future task an adaption to the current specification [FIH$^+$08] could be advisable. To use the Small-Packet Variant [FIKU07] could also be an additional feature because it will enable applications using small packets to achieve rates comparable to that of TCP flows using larger packet sizes.

Bibliography

[APS99] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[Bra89] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, October 1989.

[bui] Package: build-essential, http://packages.ubuntu.com/en/jaunty/build-essential.

[Cla82] D. Clark. window and acknowledgement strategy in TCP. RFC 813 (Standard), July 1982.

[CO98] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *SIGCOMM Comput. Commun. Rev.*, 28(3):53–69, 1998.

[CWwL03] Seongho Cho, Heekyoung Woo, and Jong won Lee. ATFRC: Adaptive TCP Friendly Rate Control Protocol. In *Lecture Notes in Computer Science*. SpringerVerlag, 2003.

[CZ04] Minghua Chen and Avideh Zakhor. Rate control for streaming video over wireless. In *Proc. IEEE INFOCOM, Hongkong*, pages 1181–1190, 2004.

[DW08] D. Damjanovic and M. Welzl. MulTFRC: Providing Weighted Fairness for Multimedia Applications (and others too!), 2008.

[Eth] Ethereal, http://www.ethereal.com/.

[FIH$^+$03] S. Floyd, ICIR, M. Handley, J. Padhye, Microsoft, J. Widmer, and University of Mannheim. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448, January 2003.

[FIH$^+$07] S. Floyd, ICIR, M. Handley, University College London, J. Padhye, Microsoft, J. Widmer, and University of Mannheim. TCP Friendly Rate Control (TFRC): Protocol Specification draft-ietf-dccp-rfc3448bis-02. RFC 3448 (INTERNET DRAFT), July 2007.

[FIH$^+$08] S. Floyd, ICIR, M. Handley, University College London, J. Padhye, Microsoft, J. Widmer, and DoCoMo. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348, September 2008.

[FIKU07] Sally Floyd, ICIR, Eddie Kohler, and UCLA. TCP Friendly Rate Control (TFRC): The Small Packet (SP) Variant. RFC 4828 (Experimental), April 2007.

[Get] Getting Socket Options: http://vesti.matf.bg.ac.yu/manuals/lspe/snode=101.html.

[Gnu] GCC, the GNU Compiler Collection, http://www.gnu.org/software/gcc/gcc.html.

[Jaa04] Sven Jaap. Tcp-friendly rate control (tfrc). Technical report, Technical University of Braunschweig, January 2004.

[Jac88] Van Jacobson. Congestion avoidance and control, 1988.

[KFS07] E. Kohler, S. Floyd, and A. Sathiaseelan. Faster Restart for TCP Friendly Rate Control (TFRC) draft-ietf-dccp-tfrc-faster-restart-04.txt. INTERNET DRAFT, September 2007.

[KHF06] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340, March 2006.

[Kil] Kile, http://kile.sourceforge.net/.

[KR05] James F. Kurose and Keith W. Ross. *computer networking - a top-down approach featuring the internet*. Addiscon Wesley, third edition, 2005.

[Lat] Latex, http://www.latex-project.org/.

[MiK] MiKTeX, http://www.miktex.org/.

[MWMM01] M. Miyabayashi, N. Wakamiya, M. Murata, and H. Miyahara. MPEG-4 Video Transfer with TCP-friendly Rate Control Protocol, 2001.

[NISa] http://www.dps.uibk.ac.at/∼sven/archive/doc/eurongi-linz06-notes.txt.

[NISb] NISTnet, http://snad.ncsl.nist.gov/nistnet/index.html.

[OMP01] David E. Ott and Ketan Mayer-Patel. Transport-level protocol coordination in cluster-to-cluster applications. In *In Proceedings of 2002 USENIX Annual Technical Conference*, pages 147–159, 2001.

[PA00] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), November 2000.

[Pos80] J. Postel. User Datagram Protocol. RFC 768, August 1980.

[Pos81] J. Postel. Transmission Control Protocol (TCP). RFC 793 (Standard), September 1981. Updated by RFC 3168.

[San09] Sandvine. 2009 Global Broadband Phenomena. 2009. 2009 Global Broadband Phenomena - Executive Summary.pdf.

[Ste97] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, January 1997.

[tcp] tcpdump, http://www.tcpdump.org/.

[Ubu] Ubuntu, http://www.ubuntu.com/.

[WH06] J. Widmer and M. Handley. TCP-friendly Multicast Congestion Control, May 2006.

[Wid00] J. Widmer. TFRC - Experimental Code: http://www.icir.org/tfrc/code/, 2000.

[Win] WinEDT, http://www.winedt.com/.

[Wir] Wireshark, http://www.wireshark.org/.

Index

Appendix A

Related Software

This section provides an overview of the used software.

A.1 Latex

LaTeX [Lat] provides a high-level language for TeX, which delivers a standardized, high-quality book-like output on every machine where it is used. It is based on source code which generates the layout based on well defined guidelines, but which can be adapted according to the users's wishes. Many text components like numbering, cross-referencing, tables and figures are provided by source code commands. LaTeX is often used in areas like mathematics because of its handling of formulas. Bibliographies can be stored in sperate files and can therefore be easily shared between different documents. Documents are most commonly exported to the PDF- or PS-format.

A.2 MiKTeX

MiKTeX [MiK] is a free TeX-implementation for Windows and was used in conjunction with WinEdt (section A.3) in the creation of this document. MiKTeX offers a graphical user interface to install and update individual packages needed for the compilation of documents. It is planned that future releases will feature Linux-support.

A.3 WinEdt

WinEdt [Win] is a shareware editor for Windows operating systems. It provides the user with many helpful functionalities in conjunction with LaTeX, although it also supports other languages like HTML. Features like text-highlighting and spell-checking make it a viable alternative to normal text editors. The generation of the different document types like PDF and PS can be done in a graphical environment.

A.4 Kile

Kile [Kil] is a user friendly TeX/LaTeX editor for the KDE desktop environment and features many of the advantages provided by WinEdt. KDE is available for many architectures such as PC, PowerPC (Mac for example) and SPARC. It has been used as an alternative to WinEdt (section A.3).

A.5 Ubuntu

Ubuntu [Ubu] is a Linux-distribution based on Debian and is a free and open source operating system that is community developed, usable for laptops, desktops and servers. While Ubuntu is using GNOME as the Desktop-environment, there also distributions which are using KDE and Xfce. It is available as a 32-bit and 64-bit version, although the TFRC/MulTFRC program is in need of specific 32-bit libraries and can therefore not be executed in a 64-bit environment.
This implementation was developed on computers running Ubuntu 7.10 and 8.04 and 8.10 and finally 9.04.

A.6 GnuCC

GnuCC [Gnu] stands for GNU Compiler Collection and includes front ends for C, C++, Objective-C, Fortran, Java, and Ada, among others, as well as libraries for these languages. It has been produced by the GNU project and is used as the default compiler in many operating systems.

A.7 tcpdump

In the duration of the development of this project tcpdump [tcp] is used as a tool to check incoming and outgoing packets. It is able to capture all kind of packets but can also be configured to ignore certain packet streams and so it only stores streams which are interesting to the user. It is able to have a live output or to store it to a file.

A.8 Wireshark

Wireshark [Wir] is the successor of Ethereal [Eth] and offers a GUI frontend to the capturing abilities similar to tcpdump. It is also possible to use filters on capturing logs. Wireshark offers different statistical operations to the user, for example all throughput graphs used in this document were a direct output of the program.

A.9 NISTnet

As a tool to simulate real world traffic the network emulator NISTnet [NISb] had been used. To be able to use this tool with an actual Linux kernel, version 2.0.12c and appropriate patches have to be

aligned. Problems encountered during installation could be solved by a guideline posted by Sven Hessler [NISa]. Additional to this guideline it is important to have the X development kit.
Due to different problems during installation and execution of NISTnet, the usage of this tool has been dropped and instead the traffic control which is built-in in the Linux kernel is used to simulate different network scenarios.

A.10 NetUDP

The MulTFRC code is using the data transfer developed at University College London between 1995 and 1997. *Copyright (c) 1995-97 University College London All rights reserved.*