MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Students' Scientific Conference 2014

Development of an Efficient Transport Protocol based on Digital Fountain

Author
Szilárd Solymos

Supervisors
Dr. Sándor Molnár
Zoltán Móczár

Budapest, Hungary

Contents

Abstract

Most of the Internet traffic is transported by TCP (Transmission Control Protocol) which applies a congestion control mechanism. This mechanism is not able to work efficiently in today's constantly changing network environments. In order to solve this issue, a number of new TCP variants have been developed which are able to utilize the network resources more efficiently by modifying the conventional congestion control algorithm. Recently, the widely used TCP versions are able to provide more efficient solutions in given environments but they do not give universal and optimal solutions to the challenges of today's ever-changing, heterogeneous network environments. Currently, it seems there is little hope that the TCP's closed-loop congestion control mechanism will be able to provide such a universal solution in the future.

In order to utilize the network resources optimally and find answers to the interoperability and efficiency issues of TCP, extensive research is now being conducted by which innovative concepts have been presented. One of these promising ideas suggests omitting congestion control completely. In this case, every entity in the network is allowed to send data at its maximum rate, which makes it possible to fully utilize the network resources. The emerging huge, mostly bursty packet loss is then compensated by applying effective erasure codes by which all the data can be restored at the receiver. In the past few years, a new transport protocol has been designed in the HSN Lab (High Speed Networks Laboratory) of the Department of Telecommunications and Media Informatics. The protocol is called DFCP (Digital Fountain based Communication Protocol) and is currently under research and development. I have implemented DFCP in the Linux kernel and carried out an extensive performance evaluation of the protocol. Instead of using a congestion control mechanism, DFCP applies efficient erasure codes in order to be able to cope with the emerging packet loss. The operation and behavior of the protocol have been analyzed on various network topologies by carrying out both testbed measurements and simulations. The preliminary results of the experiments are presented in this paper and have confirmed that DFCP is able to achieve better performance in several network scenarios compared to different TCP variants.

Firstly, following the introduction, this paper provides an insight into the operation of the most extensively used TCP versions today reviewing their congestion control algorithms. Afterwards, a detailed description is given of the underlying concept of the protocol. Subsequently, the basic operation and the features of DFCP are discussed. In the last part of the paper, firstly, the results of the validation measurements are presented. Secondly, a

comparative analysis is carried out during which the performance of DFCP is compared to commonly used TCP variants. The results of the measurements are discussed in detail and finally, the future plans and possible improvements regarding the protocol are described.

Kivonat

Napjainkban az Internet forgalmának jelentős része a TCP (Transmission Control Protocol) segítségével kerül átvitelre, amely torlódásszabályozást alkalmaz. A folyamatosan változó hálózati környezetek esetén azonban a TCP által használt torlódásszabályozási mechanizmus nem képes hatékony megoldást nyújtani. Ennek kiküszöbölésére az elmúlt évtizedekben számos TCP verziót fejlesztettek ki. A jelenleg használt TCP verziók a hagyományos torlódásvezérlő algoritmus módosításával képesek kezelni a hálózati erőforrások bizonyos esetekben fellépő alacsony kihasználtságát, és ezáltal megoldást nyújtani néhány meghatározott hálózati környezet esetén. A TCP verziók sokféleségének ugyanakkor az az ára, hogy nem adnak univerzális, optimális megoldást a napjainkban jellemző, folyamatosan változó, heterogén környezetek okozta kihívásokra. Úgy tűnik, hogy kevés remény van arra, hogy a TCP által használt zárt hurkú torlódásszabályozás a jövőben képes lesz univerzális megoldást nyújtani ezekben az esetekben.

Az ehhez a problémához kapcsolódóan elvégzett és jelenleg is folyó kutatások során több javaslat született arra, hogy hogyan lehetne hatékonyabban kihasználni a hálózati erőforrásokat. Egy alternatív megoldás a jövő Internetére nézve az lehet, hogy egyáltalán nem alkalmazunk torlódásszabályozást, és a hálózatban minden entitás esetén maximális sebességgel történik az adatküldés. Ilyen módon lehetségessé válik a hálózati erőforrások teljes mértékű kihasználása. Az adatátvitel során fellépő, főleg csomós jellegű csomagvesztést hatékony hibajavító kódolás segítségével kezelhetjük és ilyen módon történhet az elküldött adatok helyreállítása. Az elmúlt években egy ezen a koncepción alapuló transzport protokollt dolgoztunk ki a Távközlési és Médiainformatikai Tanszéken működő HSN (High Speed Networks) Laboratóriumban. Ez a protokoll DFCP (Digital Fountain based Communication Protocol) néven jelenleg is kutatás és fejlesztés alatt áll. A DFCP-t a Linux kernelben implementáltam, továbbá a protokoll számos teljesítményelemzési vizsgálatát is elvégeztem. A DFCP nem alkalmaz torlódásszabályozást és az átvitel közben történő csomagvesztéseket hatékony hibajavító kódolás alkalmazásával állítja helyre. A protokoll működését különböző hálózati topológiákon teszthálózati mérések és szimulációk segítségével elemezzük. Az eddig elvégzett mérések eredményeit ebben a dolgozatban ismertetem és ezek azt mutatják, hogy a TCP különböző verzióival összehasonlítva a DFCP számos hálózati beállítás mellett képes azoknál jobb teljesítményt elérni.

A dolgozatban a bevezetést követően áttekintem a jelenleg elterjedten alkalmazott TCP verziókat és az általuk használt torlódásszabályozási algoritmusokat, kitérve azok előnyeire és hátrányaira. Ezután ismertetem az új protokollt megalapozó koncepciót és a protokoll

alapvető működését, tulajdonságait. Az utolsó részben pedig először az elvégzett validációs mérések eredményeit mutatom be, majd egy összehasonlító elemzést láthatunk, amelynek során a DFCP teljesítményét különböző, elterjedten használt TCP verziókkal vetem össze. Az elvégzett mérések eredményeit minden esetben részletesen értékelem, végül kitérek a protokollhoz kapcsolódó továbbfejlesztési lehetőségekre.

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Dr. Sándor Molnár and Zoltán Móczár, for their useful comments, remarks and engagement throughout the summer and the semester. Their guidance, attention, patience and continuous support made it possible for me to accomplish this work including implementing the protocol, analyzing its behavior and performance and writing this conference paper.

Chapter 1

Introduction

1.1 Motivation and objectives

Nowadays, TCP is one of the most significant transport protocols providing *reliable*, *ordered* and *error-checked* delivery of a stream of octets between application-level entities. This protocol applies a congestion control mechanism in order to utilize the network resources more efficiently and manage traffic congestion avoiding congestion collapse on the early Internet. When applying congestion control, the data transfer rate of the communicating entities is determined based on factors such as packet loss and delay in the network. Considering the fact that there are numerous different network environments, ranging from high-speed backbone networks to high-delay satellite links and wireless environments, several TCP versions [1], [2] have been developed. Primarily, their aim is to optimize performance in some given network scenarios. Although these new TCP versions provide better solutions in some situations, they are not able to work efficiently in an ever-changing, heterogeneous network environment. In addition, this approach results in significant interoperability problems between the different TCP implementations. In order to solve these problems, it is required to conduct research and work out new concepts and ideas. According to a promising idea, proposed by GENI [3], applying congestion control should be completely avoided. In this case, every entity in the network is allowed to send data at its maximum rate and the emerging huge, mostly bursty packet loss is compensated by applying *effective erasure codes*. Furthermore, so as to provide fairness, a *fair scheduling algorithm* is assumed to be employed in the case of the network devices. This scheme has several benefits. Firstly, it provides an *efficient* solution since every network resource is fully utilized and all additional free capacity in the network will immediately be consumed. Moreover, the *simplicity* of the concept must be emphasized as the suggested coding scheme makes packet loss inconsequential, which can simplify network routers resulting in reduced buffer sizes, and consequently this proposal is able to provide support for all-optical networks where buffer sizing is a key issue in the case of Internet routers [4], [5], [6]. Finally, the *scalability* and the *stability* of the new approach are important factors as well since the maximum-rate transmission results in more predictable traffic patterns. However, apart from the previously mentioned characteristics and some related work discussed in a later chapter, no realization or further refinement of the idea has been published so far.

In this paper, an efficient new transport protocol is introduced that is based on the concept described above. Accordingly, instead of applying a congestion control algorithm, the protocol employs an effective erasure coding method in order to recover the lost packets in the course of maximum-rate data transmission. The performance of the protocol has been analyzed by testbed measurements and simulations performed on different network topologies. All the results are discussed in detail and compared to the performance of today's commonly used TCP variants.

My work has been carried out as part of ongoing research conducted in the *HSN Lab* (High Speed Networks Laboratory) of the Department of Telecommunications and Media Informatics. More precisely, my contribution to this research was to implement a new transport protocol, carry out its validation and comparative performance evaluation.

1.2 Outline of the work

This paper is divided into five chapters. After the introduction and the description of the problem, Chapter 2 begins with an overview of the basic concepts in computer networking. Afterwards, the currently available and widely used high-speed TCP versions are investigated highlighting the underlying principles, their congestion control mechanisms, the advantages and the drawbacks of the applied algorithms. In Chapter 3, DFCP (Digital Fountain based Communication Protocol) [7] and its underlying concept are introduced. After outlining the concept, a basic overview of the network subsystem of the Linux kernel is given. Subsequently, the chapter focuses on the operational phases of the protocol. In addition, the applied coding scheme and the parameters of the protocol are also detailed. Chapter 4 concentrates on the testbed measurements and simulations. The chapter is split into two parts. On the one hand, a validation analysis is performed so as to examine and validate the operation of the protocol. On the other hand, a comparative performance evaluation is carried out in order to compare the performance and behavior of the protocol to commonly used TCP versions under various network conditions. First of all, the network topologies and traffic scenarios are presented. Secondly, the software tools used for the measurements are briefly described and all their settings are given. Thirdly, the measurement results are provided, evaluated and compared to the performance of significant TCP variants. The closing chapter, Chapter 5, details the suggested future work and improvements.

Chapter 2

High-Speed Transport Protocols

This chapter focuses on the different transport protocols and TCP variants [2] used in high-speed networks, which are able to improve the performance of traditional TCP in given network environments with special emphasis placed on their congestion control algorithms.

2.1 Related work

In order to provide effective solutions to the challenges of various network environments, the conventional congestion control mechanism of TCP has been modified and several variants have been developed. These mechanisms might be divided into five groups. The first group, *loss-based* congestion control algorithms, includes TCP versions which use packet loss as an indication of network congestion. Since detecting packet losses gives very limited, one-bit information of the state of the network, this approach cannot provide fine granularity for congestion control. For instance, TCP Reno [8], HSTCP (High Speed TCP) [9], Scalable TCP [10], BIC TCP (Binary Increase Congestion control) [11], which is available in the Linux kernel, and CUBIC TCP [12], which is currently the default TCP variant in the Linux kernel, apply loss-based congestion control mechanisms. The second group, *delay-based* congestion control algorithms, consists of TCP versions which periodically measure the RTT (round-trip time) rather than the packet loss to determine the rate at which to send packets. RTT is the total time it takes for a segment to be sent and for an acknowledgement of that segment to be received. TCP Vegas [13] and FAST TCP [14] are examples of TCP variants which use delay-based algorithms. In addition, FAST TCP has favorable interoperability and scalability properties. The algorithms in the third group, *hybrid* or *mixed loss-delay-based* variants, involve the features of both the loss-based and the delay-based approaches in order to achieve fair bandwidth allocation and fairness among flows. For example, Compound TCP [15], available in Windows 7, Windows Vista and Windows Server 2008 operating systems, applies a hybrid mechanism. In addition to the packet loss, it also uses a delay-based component and the two components determine the packet transmission rate together. The fourth group consists of different *estimation-based* strategies. TCP Westwood [16] is an example which performs adaptive rate estimation based on the arrival rate of acknowledgement packets. Two rate estimation algorithms, the BE (Bandwidth Estimation) algorithm and the RE (Rate Estimation) algorithm, are employed to measure the appropriate transmission rate of an end-to-end path. In the case of the fifth group, congestion indications are replaced by explicit notifications, such as ECN (Explicit Congestion Notification). These congestion control mechanisms require the assistance of network routers, and therefore the modification of routers is also necessary, which is a significant disadvantage from the aspect of deployment feasibility. One of the main representatives of this group is the XCP (eXplicit Control Protocol) which generalizes the ECN proposal [17]. Instead of the one-bit congestion indication used by ECN, XCP-capable routers inform the senders about the degree of congestion at the bottleneck. In addition, XCP decouples the utilization control from fairness control. However, XCP requires the collaboration of all the routers on the data path, which is almost impossible to achieve in an incremental deployment scenario of XCP.

2.2 TCP basics

This section provides an introduction to the basic concepts in computer networking required to understand the following chapters of this paper.

The first step of congestion control, applied by TCP variants, is the detection of congestion. In the early Internet, if a timer expired due to a lost packet, the reason could have been both noise on the transmission channel and an overwhelmed router. However, packet losses relatively rarely take place due to transmission errors in the current networks since most high-speed backbone links are optical today. As a result, packet losses and timeouts are typically caused by network congestion. All the algorithms applied by TCP assume that timeouts are caused by congestion in the network. In the later sections, it will become apparent that in some cases, such as wireless transmission when packet losses may occur due to the unfavorable properties of the wireless medium, this approach is highly inefficient and suboptimal.

TCP versions use various timers for different reasons. The most important one is the *retransmission timer* [18]. TCP starts the retransmission timer when each outbound segment is handed down to IP (Internet Protocol). If no acknowledgment has been received for the data in a given segment before the timer expires, the segment is retransmitted and the timer is reset and started again. Otherwise, the timer is stopped. Choosing the right timeout value is crucial. If the timeout is too short, too many packets will be retransmitted, which may increase network congestion. If it is too long, the overall throughput may be reduced due to waiting for the acknowledgments for too long in the case of lost packets. In order to determine how long the timer should run, the value of RTT is measured and used. RTT refers to the round-trip time, which is the time required for a client to send a segment and for the server to send an acknowledgement to that segment over the network. Both the average and the variance of the arrival time of acknowledgements may fluctuate significantly within a few seconds if network congestion occurs or disappears. So as to provide an approximation of the RTT, a dynamic algorithm is employed. The state of the network is continuously monitored and the value of RTT is appropriately updated. In the case of TCP, this algorithm was introduced by Jacobson and it works as follows. The TCP keeps track of each connection and maintains their corresponding RTT variables where RTT denotes the best known approximation of the round-trip time to the given destination. When sending a segment, TCP initiates a timer and measures the time it takes for the acknowledgement to that segment to arrive. On the arrival of the acknowledgement, it calculates the total time, M, and refreshes the value of RTT according to Equation (2.1):

$$RTT = \alpha \cdot RTT + (1 - \alpha) \cdot M \tag{2.1}$$

In this case, α is a weighting factor that is typically set to 7/8. In order to determine the value of the timeout, early implementations of TCP commonly used the formula $\beta \cdot RTT$, where the value of β was always 2. As this constant value was inflexible, the algorithm was modified based on Jacobson's proposal in 1988. As a result, an additional variable D, the deviation, was added to the original algorithm. Whenever an acknowledgement is received, the difference between the expected and observed values, $|RTT - M|$, is computed. The value of D is then calculated as a moving average according to Formula (2.2):

$$D = \alpha \cdot D + (1 - \alpha) \cdot |RTT - M| \tag{2.2}$$

In this equation, α can be different from the value used in Formula (2.1). Eventually, the value of the timeout (T) is given by Equation (2.3):

$$T = RTT + 4 \cdot D \tag{2.3}$$

Although the value of the coefficient can be set to an arbitrary constant, using the value of 4 has two major advantages. Firstly, the multiplication by 4 can be easily implemented by shifting. Secondly, it reduces the number of unnecessary retransmissions and timeouts.

Another important concept in the context of computer networks is *BDP* (Bandwidth-Delay Product) which refers to the capacity of the network pipe from the sender to the receiver and back in bits or in bytes. It can be calculated as shown in Equation (2.4):

$$BDP = B \cdot RTT \tag{2.4}$$

Here, B denotes the bandwidth and RTT refers to the round-trip time. For instance, in the case of a network with bandwidth of 1 Gbps and an RTT of 1 ms, the value of BDP is calculated in Equation (2.5):

$$BDP = 10^9 \frac{b}{s} \cdot 10^{-3} s = 10^6 b = 125kB \tag{2.5}$$

Another fundamental characteristic of any network is its MTU (Maximum Transmission Unit) which determines the size of the largest packet or frame, specified in octets (eight-bit bytes), that can be transmitted over the given transmission medium. The term *packet* refers to the network layer PDU (Protocol Data Unit) whereas the data link layer PDU is known as *frame*. Furthermore, the transport layer PDU is called *segment*.

The definition of *MSS* (Maximum Segment Size) must be highlighted as well. The MSS is a parameter of TCP which specifies the largest amount of data, specified in octets, that a computer or communications device can receive in a single, unfragmented TCP segment. It does not include the size of the TCP header or the IP header.

I also introduce the concept of the *response function* of a congestion control protocol. The response function of TCP is the average throughput of a TCP connection in a function of the packet loss probability, the packet size (*MSS*) and the *RTT*. It can give much information about the protocol especially about its TCP friendliness, RTT fairness, convergence and scalability.

Finally, congestion control algorithms maintain a variable at the sender called *congestion window* which determines the number of segments that can be transmitted within an *RTT*.

In the following sections, some of the TCP versions in the first two groups will be analyzed including their properties focusing on their congestion control mechanisms. Among others, the advantages and the drawbacks of the applied algorithms will be detailed.

2.3 TCP Tahoe and TCP Reno

TCP Tahoe and TCP Reno were named after the operating systems, 4.3BSD-Tahoe and 4.3BSD-Reno releases [19], in which they were first introduced in 1988 and 1990, respectively. The name of TCP Reno refers to the city of *Reno* while TCP Tahoe was named after *Lake Tahoe* in Nevada, USA. TCP Reno is the improved and more efficient version of TCP Tahoe. In this section, firstly, the common characteristics of these two TCP versions are discussed. Afterwards, the differences are also explained.

When dealing with network congestion, two separate issues must be considered, which are the capacity of the receiver and the capacity of the network itself. These two issues are handled separately. Consequently, each sender uses two different windows. One of them is maintained by the receiver whereas the other, the *congestion window*, is under the control of the sender. Both variables are used in order to determine the number of bytes that can be sent by the sender. The number of bytes that can actually be sent will be the minimum of the two windows, and therefore the communicating endpoints make this decision together.

When a connection has been established *cwnd* (congestion window) is set to the maximum segment size allowed on that connection. Afterwards, a maximum-sized segment is sent. If that segment is acknowledged before the timer has expired, *cwnd* is increased by one segment (*MSS* bytes), and therefore the initial value of *cwnd* is doubled ($cwnd = cwnd + MSS$) allowing the sender to send two maximum-sized segments. If the value of *cwnd* is n segments and all the n acknowledgements are received, the value of *cwnd* is increased by the size of n segments. In practice, the value of *cwnd* is doubled for each acknowledged data burst. As a result, *cwnd* keeps growing exponentially until either a timeout occurs or the size of the receiver's window has been reached. This algorithm is called *Slow Start*, despite the fact that it is not slow at all, and all TCP implementations are required to support it.

In addition, the congestion control mechanism uses a third parameter, called *threshold*, that is initially set to 64 kilobytes. When a timeout occurs, the threshold is set to the half of the current size of *cwnd* and *cwnd* is reset to the size of a maximum-sized segment. Afterwards, the Slow Start algorithm is again used to determine the capacity of the network except that the exponential growth stops when the value of the *cwnd* reaches the size of the threshold. Henceforth, *cwnd* is linearly increased by each successful transmission (by one *MSS* for each burst instead of one *MSS* for each acknowledged segment). This phase of the algorithm is called *Congestion Avoidance*. The previously described algorithm, employed by TCP, is commonly referred to as *AIMD* (Additive Increase Multiplicative Decrease).

As an illustration of how the congestion control algorithm works, see Figure 2.1. The *MSS* here is 1024 bytes. The value of *cwnd* was initially set to 64 kilobytes but a timeout occurred. As a result, the threshold is set to 32 kilobytes and the current value of *cwnd* is 1 kilobyte for transmission 0. The congestion window then grows exponentially until it reaches the threshold that is currently 32 kilobytes. Subsequently, the growth is linear.

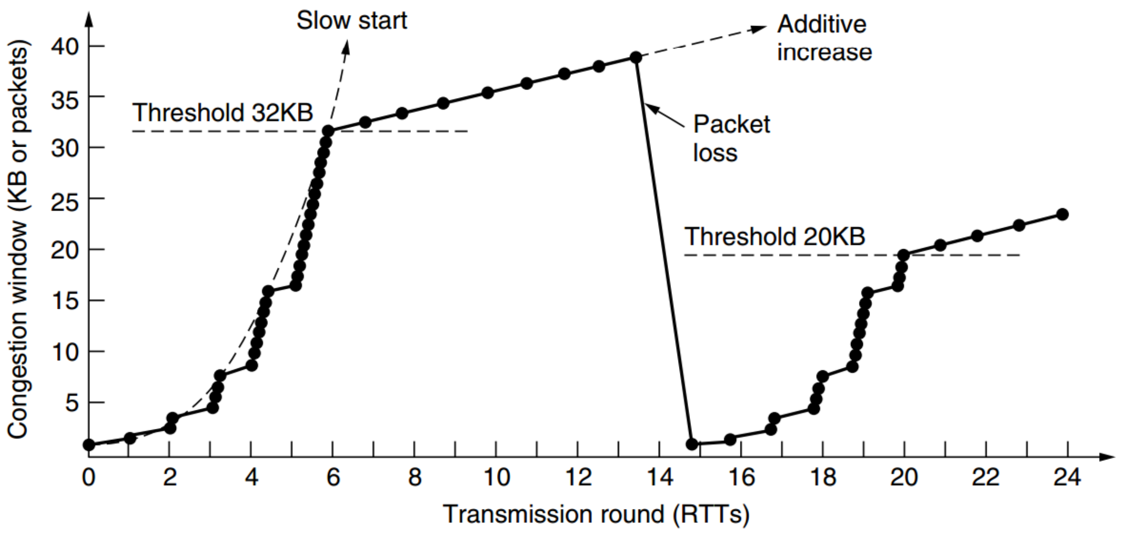


Figure 2.1. The congestion control algorithm of TCP Tahoe and TCP Reno

At transmission 13, a timeout occurs. This is detected when three duplicate acknowledgements arrive. At that time, the lost packet is retransmitted and the threshold is set to the half of the current congestion window (*cwnd* is 40 kilobytes now), which is 20 kilobytes, and the Slow Start algorithm is reinitiated. Restarting with a congestion window of one packet takes one round-trip time for all of the previously transmitted data to leave the network and be acknowledged including the retransmitted packet. The congestion window grows exponentially as it did in the Slow Start phase previously until it reaches the new threshold of 20 kilobytes. At that time, the growth becomes linear again. It will continue in this fashion until either another packet loss is detected via duplicate acknowledgements, a timeout occurs or the receiver's window becomes the limit. If *cwnd* reaches the size of the receiver's window, it stops growing and remains constant.

In the event of a packet loss, it is very likely that duplicate acknowledgements will arrive at the sender. The reason for this is that duplicate acknowledgements are generated by the receiver every time a segment is received out of order. So as to recover from such situations more effectively, a technique called *Fast Retransmit* was added to the original algorithm. It is an enhancement which reduces the time a sender waits before retransmitting a lost segment and works as follows. If a TCP sender receives a specified number of acknowledgements, which is usually set to three duplicate acknowledgements with the same acknowledgement number (a total of four acknowledgements with the same acknowledgement number), the sender can reasonably be confident that the segment with the next higher sequence number was dropped and will not arrive out of order. The sender will then retransmit the packet that was presumed dropped before waiting for its timeout.

In this case, TCP Tahoe and TCP Reno work differently. Triple duplicate acknowledgements are treated in the same way as a timeout by TCP Tahoe. It will perform Fast Retransmit, set the slow start threshold to the half of the current congestion window, reduce the congestion window to 1 *MSS* and reset to Slow Start state. However, in the case of TCP Reno, if three duplicate acknowledgements are received, it will halve the congestion window (instead of setting it to 1 *MSS* like Tahoe), set the slow start threshold equal to the new congestion window, perform a Fast Retransmit and enter a phase called *Fast Recovery*. If an acknowledgement times out, Slow Start is used as in the case of TCP Tahoe. The rationale behind this technique is that duplicate acknowledgements reflect that the subsequent packets arrived at the receiver. In this case, falling back to Slow Start would be unreasonable and inefficient. As a result of the Fast Recovery enhancement, which was introduced in TCP Reno, TCP Reno outperforms TCP Tahoe. In the later sections, for practical reasons, TCP Reno will be simply referred to as *TCP*.

2.4 TCP NewReno and TCP SACK

TCP NewReno [20] is a slight modification of TCP Reno. It is able to detect multiple packet losses, and consequently it is much more efficient than TCP Reno in the event of multiple packet losses. TCP NewReno also performs Fast Retransmit when it receives multiple duplicate packets, however, it differs from TCP Reno at this stage. Whenever a TCP NewReno sender receives duplicate acknowledgements, it assumes that the packet was lost. It enters the Fast Recovery phase and retransmits the lost packet. Upon reception of a partial acknowledgement, it does not leave the Fast Recovery phase until all the outstanding data transmitted in the same window has been successfully acknowledged. This behavior of TCP NewReno makes it different from TCP Reno. The reason for not exiting the Fast Recovery phase is the fact that partial acknowledgements (an acknowledgement for some but not all of the packets that were outstanding at the start of the Fast Recovery phase) do not acknowledge all the packets that were sent before entering the Fast Retransmission phase. Partial acknowledgements also indicate that more packets might have been lost within the same window and they need to be retransmitted immediately so that the expiry of the retransmission timer can be avoided. When the TCP NewReno sender has received all the partial acknowledgements for all the outstanding data, it receives a fresh acknowledgement. When this fresh acknowledgement has been received, the sender exits the Fast

Recovery phase and sets the value of the congestion window to the threshold performing Congestion Avoidance. In this way, it provides a very efficient mechanism to recover from multiple packet losses as the congestion window is not reduced multiple times, however, TCP NewReno still suffers from the fact that it takes one RTT to detect each packet loss.

Traditional implementations of TCP use an *acknowledgement number* field that contains a cumulative acknowledgement indicating that the receiver has received all the data up to the indicated byte. Nevertheless, TCP may experience poor performance when multiple packets are lost from one window of data. With the limited information available from cumulative acknowledgments, a TCP sender can only learn about a single lost packet per round-trip time. A $SACK$ (Selective Acknowledgement) [21] mechanism, combined with a selective repeat retransmission policy, may help to overcome these limitations. The receiver sends back SACK packets to the sender informing the sender of the data that has been received. The sender can then retransmit only the missing data segments. In short, the SACK option allows the receivers to report non-sequential data they have received.

Adding SACK extension to TCP does not change the basic underlying congestion control algorithm. The TCP SACK implementation preserves the properties of TCP Tahoe and TCP Reno of being robust in the presence of out-of-order packets and uses retransmission timeouts as the recovery method of last resort. The main difference between the TCP SACK implementation and the TCP Reno is in the behavior when multiple packets are dropped from one window of data. TCP SACK requires segments to be acknowledged selectively instead of being acknowledged cumulatively. Each acknowledgement has a block which describes which segments are being acknowledged. Consequently, the sender has a picture of which segments have already been acknowledged and which are still outstanding. Whenever the sender enters Fast Recovery, it initializes a variable called *pipe*, which is an estimation of how much data is outstanding in the network, and halves the congestion window. Every time an acknowledgement is received, the value of *pipe* is reduced by one and every time a segment is retransmitted, *pipe* is incremented by one. Whenever the value of *pipe* becomes smaller than the congestion window, TCP SACK checks which segments are unreceived and resends them. If there are no such segments outstanding, then it sends a new packet. As a result, more than one lost segment can be resent in one RTT. The sender exits Fast Recovery and enters Congestion Avoidance when all the outstanding packets have been acknowledged. This approach is very useful in Satellite Internet access [22]. However, the drawbacks of TCP SACK include a limitation on the number of SACK blocks that one acknowledgement can carry (due to the encoding of SACK information within the option field of the TCP header) and its reactive nature in responding to packet losses. The latter prevents TCP SACK from being able to proactively avoid congestion or infer the underlying cause of packet losses (For instance, is it a random radio noise or a traffic-based congestion event?). For this reason, network researchers and engineers have looked beyond TCP SACK for a more comprehensive solution to the problems of TCP over wireless networks.

2.5 CUBIC TCP

In this section, a widely used high-speed TCP variant, called TCP CUBIC [12], will be presented that is used by default in Linux kernels since version 2.6.19. TCP CUBIC is an enhanced version of BIC TCP (Binary Increase Congestion control) [11] which is the default congestion control algorithm in Linux kernels 2.6.8 through 2.6.18. TCP CUBIC simplifies the window control mechanism of BIC TCP and improves its TCP-friendliness and RTT-fairness. The window growth function of TCP CUBIC is a cubic function in terms of the elapsed time since the last loss event. Firstly, this cubic function provides good stability and scalability. Secondly, the real-time nature of the protocol keeps the window growth rate independent of RTT, and consequently its algorithm is TCP friendly on both short and long-RTT paths. Furthermore, its unique window growth function improves the stability and scalability of the protocol in both high-speed and long-distance networks.

A number of TCP variants have been developed in order to address the underutilization problem present in large bandwidth-delay product networks mostly due to the slow growth of the congestion window. Most of these protocols attempted to modify the window growth function of TCP, which led to fairness issues between the different TCP variants. These fairness issues include friendliness to existing TCP traffic (TCP-friendliness) and fair bandwidth sharing with other competing high-speed flows running with same or different round-trip delays (Inter/intra protocol fairness and RTT-fairness). TCP-friendliness defines whether a protocol is fair to TCP and it is critical to the safety of the protocol. When a new protocol is used, we need to make sure that its use does not affect the most common network flows (namely TCP) unfairly. The definition of TCP-friendliness is commonly interpreted as described in [9]. In brief, under high loss rate regions where TCP is well-behaved, the protocol must behave like TCP and, under low loss rate regions where TCP has a low-utilization problem, it can use more bandwidth than TCP. The main feature of TCP CUBIC is that its window growth function is defined in real-time, and therefore its growth rate will be independent of RTT. In the following part of this section, the window growth function of TCP CUBIC and its properties will be discussed.

The window growth function of BIC TCP can be too aggressive especially in the case of short RTT paths or low-speed networks. Furthermore, the several different phases of the window control mechanism add a lot of complexity in analyzing the behavior and performance of the protocol. The aim was to find a new window growth function while retaining most of strengths of BIC TCP, especially its stability and scalability, simplify the window control mechanism and enhance its TCP-friendliness. The congestion window of TCP CUBIC is determined by the cubic function in Equation (2.6):

$$W_{cubic} = C \cdot (t - K)^3 + W_{max} \tag{2.6}$$

Here, C is a scaling factor, t denotes the elapsed time since the last window reduction event and W_{max} is the window size just before the last window reduction. The parameter

K can be calculated as shown in Formula (2.7):

$$K = \sqrt[3]{W_{max} \cdot \frac{\beta}{C}} \tag{2.7}$$

In this formula, β is a constant multiplicative decrease factor applied for window reduction at the time of a loss event. More precisely, the window was reduced to $W_{max} \cdot \beta$ at the time of the last reduction.

Figure 2.2 shows the window growth function of TCP CUBIC. While most alternative algorithms of TCP employ convex growth functions, where after a loss event the window increment is always growing, TCP CUBIC uses both the concave and convex profiles of the cubic function to increase the window.

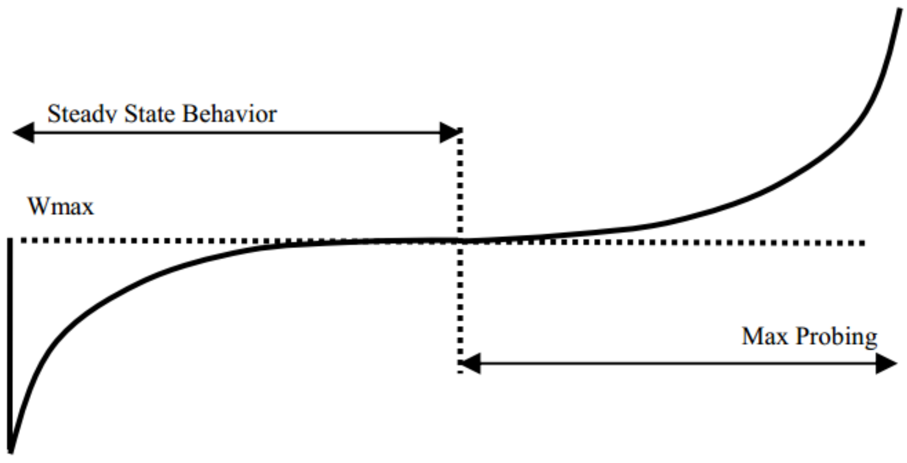


Figure 2.2. The window growth function of TCP CUBIC

After a window reduction following a loss event, it sets the window size to the value of W_{max} at the time of the loss event and decreases the congestion window multiplicatively as described above. In addition, it enters the regular Fast Recovery and Fast Retransmit phases of TCP. After having left Fast Recovery, it enters Congestion Avoidance and starts to increase the window using the concave profile of the cubic function as depicted in Figure 2.2. The cubic function is set to have its plateau at W_{max}, and therefore the concave growth continues until the window size reaches the value of W_{max}. It can be seen that the window grows very fast after a window reduction event, but as it gets closer to W_{max}, its growth slows down. Afterwards, the convex profile of the cubic function is used to increase the window. Around W_{max}, the window increment becomes almost zero. After this point, TCP CUBIC starts probing for more bandwidth. At this stage, initially, the window grows slowly accelerating its growth as it is moving away from W_{max}. This kind of window adjustment (concave and then convex) improves the behavior of the protocol and the stability of the network while maintaining high network utilization. Furthermore, the rapid growth after having reached the value of W_{max} ensures the scalability of the protocol. This is due to the fact that the window size remains almost constant forming a plateau around W_{max} where the network utilization is deemed to be the highest and, in steady-state, most window size samples of TCP CUBIC are close to W_{max} promoting high network utilization and protocol stability.

In order to further improve the fairness and stability of the protocol, the window increment is limited so as not to be more than S_{max} per second. This feature allows the window to grow linearly when it is far away from W_{max} making the growth function similar to the function used by BIC TCP which increases the window additively when the window increment per RTT becomes larger than a constant value.

The real-time increase of the window significantly enhances the TCP-friendliness of the protocol. In general, the window growth function of other RTT-dependent protocols (for example, TCP) grows proportionally faster (in real time) in shorter RTT networks whereas, in the case of TCP CUBIC, the growth is independent of RTT. Since TCP behaves more aggressively while the behavior of TCP CUBIC remains unchanged, in networks with short RTT, TCP CUBIC is more friendly than TCP. In short RTT networks, TCP CUBIC increases the window more slowly than TCP. In order to keep the growth rate at the same level as TCP, the window adjustment algorithm of TCP is emulated after a packet loss event. This phase of the algorithm is called *TCP Mode*. When receiving an acknowledgement in Congestion Avoidance phase, TCP CUBIC calculates the window growth rate during the next RTT period applying Equation (2.6). Afterwards, it sets $W(t + RTT)$ as the candidate target value of the congestion window. Assume that the current window size is *cwnd*. Depending on the value of *cwnd*, TCP CUBIC can run in three different modes. Firstly, if *cwnd* is less than the window size that Standard TCP would reach at time t after the last loss event, then TCP CUBIC is in the TCP friendly region. Otherwise, if *cwnd* is less than W_{max}, then TCP CUBIC is in the concave region. Finally, if *cwnd* is larger than W_{max}, TCP CUBIC is in the convex region.

If TCP mode is entered after a packet loss event, the window adjustment algorithm of TCP is emulated. Since TCP CUBIC reduces its window by a multiplicative factor of β after a loss event, the TCP-fair additive increment would be $3 \cdot \beta/(2 - \beta)$ per RTT. The reason for this is that the average sending rate of a protocol that uses an AIMD algorithm can be calculated as shown in Formula (2.8):

$$\frac{1}{RTT} \cdot \sqrt{\frac{\alpha}{2} \cdot \frac{2-\beta}{\beta} \cdot \frac{1}{p}} \tag{2.8}$$

In the above formula, α is the additive window increment and p is the packet loss rate. In the case of TCP, $\alpha = 1$ and $\beta = 1/2$, and consequently the average sending rate of TCP can be computed according to Formula (2.9):

$$\frac{1}{RTT} \cdot \sqrt{\frac{3}{2} \cdot \frac{1}{p}} \tag{2.9}$$

In order to achieve the same average sending rate as TCP for an arbitrary β, based on the two formulas shown above, α has to be set to $3 \cdot \beta/(2 - \beta)$. If the growth rate is α per RTT, the window size of the emulated algorithm of TCP at time t is given by

Equation (2.10):

$$W_{tcp}(t) = W_{max} \cdot (1 - \beta) + 3 \cdot \frac{\beta}{2 - \beta} \cdot \frac{t}{RTT} \tag{2.10}$$

If the current value of *cwnd* is less than $W_{tcp}(t)$, then the protocol is in the TCP friendly region and *cwnd* is set to $W_{tcp}(t)$ for each received acknowledgement.

To sum up, in the case of TCP CUBIC, it was important to improve the interoperability and fairness to the traditional TCP flows. Besides, the algorithm of TCP CUBIC provides stability and scalability while achieving high network utilization even in high BDP networks. Finally, this solution simplifies the algorithm employed by BIC TCP.

Chapter 3

Future Internet without Congestion Control

The evolution of transport protocols suggests that finding an optimal solution which meets all the challenges of the ever-changing Internet is almost hopeless. In this chapter, firstly, a promising concept is introduced and a new architecture for the future Internet without congestion control is proposed. Afterwards, a brief overview of the network subsystem of the Linux kernel is provided. Subsequently, the focus is on the basic principles of DFCP [7] including its operational phases, data encoding and decoding algorithms and parameters.

3.1 Basic concept and related work

In the early Internet, the paradigm of congestion control was introduced as a solution to avoid congestion collapse and performance degradation due to the overload of the network resources. Congestion control is mostly performed by TCP that is commonly used to transport data on the Internet these days. As the Internet is constantly and rapidly evolving, numerous more and more effective TCP variants have been developed [1], [2] in order to fit the requirements of the ever-changing network environments today, such as wireless local area networks, long-range satellite links, ad-hoc networks, high-speed backbone links, high-loss and high-latency networks. Although the currently used TCP variants work more efficiently in some given network environments, they all fail to provide an optimal and universal mechanism in today's heterogeneous environments and under rapidly-changing network conditions. It seems there is little hope that the alternative versions of the closed-loop congestion control mechanism, employed by TCP, could result in such an optimal solution in the future. Another important drawback of TCP is its buffer space requirement that is still a significant challenge in all-optical networks where only small buffers can be realized in the case of Internet routers due to both economic and technological constraints [4], [5], [6].

Considering the limitations of TCP, in the course of research, the original concept of congestion control has been rethought and redesigned, and consequently a number of new ideas have been suggested. According to an inventive idea, advocated by GENI [3], employing congestion control should be completely omitted. In order to recover the lost packets, efficient erasure codes are applied. However, no further refinement or realization of this idea was published. Regarding previous related research, a few other ideas have already been investigated before where congestion control was not applied at all. A decongestion controller was proposed by Raghavan and Snoeren [23] who studied its benefits. In addition, Bonald et al. [24] investigated the behavior of the network in the absence of congestion control. Their impressive results confuted the widely believed assumption according to which operating a network without congestion control leads to congestion collapse. López et al. [25] analyzed a fountain-based protocol through game theory. Their work showed that a Nash equilibrium can be obtained and, at this equilibrium, the performance of the network is similar to the performance obtained when all hosts comply with TCP. Furthermore, [26] introduced a new approach that employs rateless codes in a live unicast video streaming system including the experimental results. Botos et al. [27] presented a transport protocol based on rateless codes as an alternative to TCP for channels which experience high packet loss rates applying rateless codes. Kumar et al. [28] developed a transport protocol based on fountain codes, called FCT (Fountain Codes based Transport), that is able to achieve promising performance in an IEEE 802.11 WLAN cell. In their work, a detailed performance analysis study is carried out in order to provide an insight into the choice of various system parameters that can lead to optimal throughput performance. Additionally, they present a brief comparison between the performance of FCT and TCP by simulations.

3.2 Future Internet network architecture

In this paper, I introduce a new future Internet network architecture and a related novel transport protocol called Digital Fountain based Communication Protocol (DFCP) [7]. Henceforth, it is assumed that congestion control is not employed in the future Internet at all [3]. In this case, every entity in the network is allowed to send data at its maximum rate and the emerging huge, mostly bursty packet loss is compensated by applying effective *erasure codes*. Furthermore, so as to provide fairness among the competing flows, a *fair scheduling algorithm* is required to be employed in the case of all the network devices. This scheme has several benefits. Firstly, this network architecture provides an *efficient* solution since every network resource is fully utilized and all additional free capacity in the network will immediately be consumed. Moreover, the *simplicity* of the concept must be emphasized as the suggested coding scheme makes packet loss inconsequential, which can simplify network routers resulting in reduced buffer sizes, and consequently this proposal is able to provide support for all-optical networks where buffer sizing is a key question in the case of Internet routers [4], [5], [6]. Finally, the *scalability* and the *stability* of the new approach are important factors as well since the maximum-rate transmission results in more predictable traffic patterns. In addition, this would make traffic engineering a much easier task.

Omitting congestion control completely and applying maximum-rate data sending together might easily lead to enormous packet loss owing to the constant overload of the network devices. It must be highlighted that if no congestion is experienced during the maximum-rate sending, this concept would be the optimal solution as no other approaches could utilize the network resources better. In order to deal with packet loss due to the potentially heavy network congestion, the use of efficient *rateless codes* is proposed. In contrast to the traditional way of erasure coding, which transforms a message of k symbols into a longer message consisting of n symbols, rateless codes do not have a fixed code rate, which is defined as k/n. More precisely, rateless codes, which are also known as *Fountain codes*, are able to generate a potentially limitless sequence of encoded symbols from a given set of source symbols. As a result, the code rate of a fountain code tends to zero as n tends to infinity. The receiver is able to recover the sent data from any subset of the encoded stream which is only slightly longer than the original message. Consequently, the loss experienced in the network is inconsequential since the receivers only have to collect fragments of the encoded stream until they are able to decode the encoded data successfully. The *LT* (Luby Transform) codes [29] are the first practical realization of universal rateless codes which are *near-optimal* erasure correcting codes. However, LT codes cannot provide low-complexity encoding and decoding. *Raptor codes* [30] are a significant theoretical and practical improvement of LT codes, which are the first known class of fountain codes with linear time encoding and decoding. More precisely, Raptor codes require $\mathcal{O}(1)$ time to generate an encoded symbol while decoding a message of length k applying a belief propagation decoding algorithm requires $\mathcal{O}(k)$ time for the appropriate choice of inner/outer codes. As a result, in the case of the applied concept, the use of Raptor codes is proposed as a FEC (Forward Error Correction) scheme. For a message of k symbols and any real $\varepsilon > 0$ overhead parameter, a Raptor code is able to generate a potentially infinite-length encoded stream from which any subset of size $\lceil (1+\varepsilon) \cdot k \rceil$ is sufficient so that the receiver could restore the original message with high probability. The probability that a message can be restored increases with the number of symbols received. When the number of the received symbols is only slightly larger than k, the decoding probability is very close to 1. For instance, in the case of the latest generation of Raptor codes, the *RaptorQ* codes [31], the probability of decoding failure when k symbols have already been received is less than 1%. Furthermore, the chance of decoding failure when $k + 2$ symbols have been received is less than one in a million. A symbol can be any size, from a single byte to hundreds or thousands of bytes. An additional advantage of applying Raptor codes is that, both in the course of encoding and decoding, only simple operations are required, such as copying symbols and performing XOR (exclusive OR) operation on a few symbols.

Figure 3.1 depicts a network architecture based on fountain codes. In this scenario, congestion control is omitted and every entity in the network encodes its data applying Raptor codes and then sends the encoded data out at its maximum rate. The data transfer rate can only be limited by either the sending application or the capacity of the link.

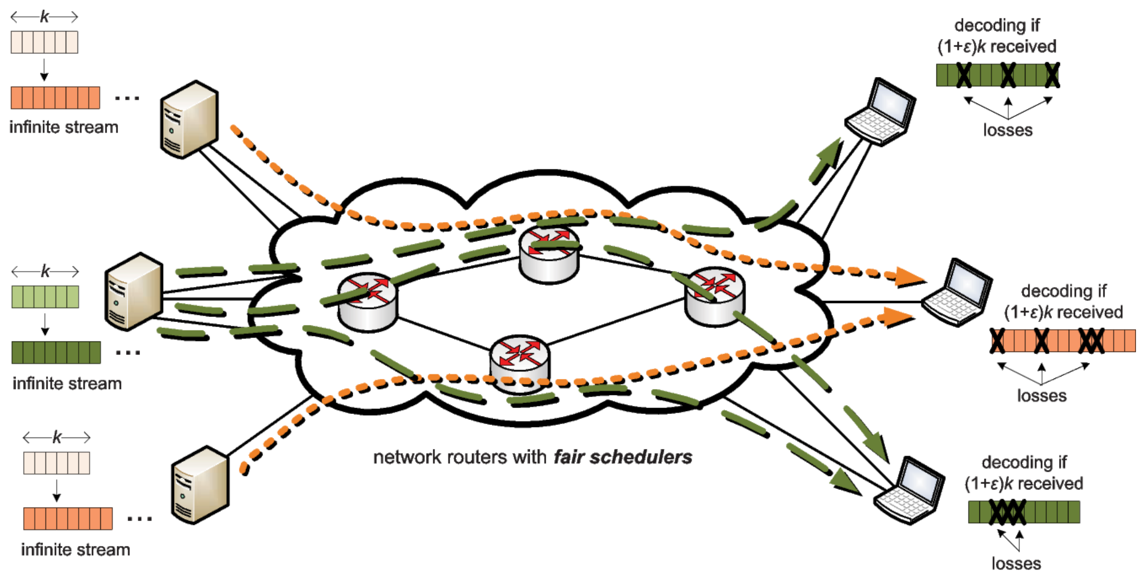

k
infinite stream
k
infinite stream
k
infinite stream
network routers with *fair schedulers*
decoding if (1+ε)k received
losses
decoding if (1+ε)k received
losses
decoding if (1+ε)k received
losses

Figure 3.1. Network architecture with N sender-receiver pairs

However, the maximum-rate sending may result in huge packet loss that can be effectively compensated by employing Raptor codes since the receiver is able to restore the original message of k symbols with high probability as soon as any subset of size $\lceil(1+\varepsilon)\cdot k\rceil$ of the encoded stream has been received. If the decoding fails, the receiver has to wait to receive more encoded symbols and make another attempt to decode the encoded data. The additional encoded symbols increase the probability of the successful decoding. These properties make the Raptor coding scheme a very efficient solution even under dynamically changing or bursty loss conditions.

Applying maximum-rate sending raises the question of fairness. The competing flows may have different access rates to a shared bottleneck link. In this case, the more aggressive senders might easily starve the less active ones. In order to solve this share allocation problem among flows sending at maximum rates, the use of *quasi-ideal fair queueing schedulers* is suggested in the network routers. Although implementing an ideal fair scheduler can be a difficult task in a normal environment, it is much simpler in a fountain code-based network transmission scheme where the packet loss is inconsequential.

In the next section, the basics of the network subsystem of the Linux kernel will be discussed which serve as a foundation for understanding the subsequent sections of this chapter.

3.3 The network subsystem of the Linux kernel

DFCP has been implemented in the Debian Linux kernel version 2.6.26-2. For testing purposes and to carry out the measurements, the Debian Linux version 5.0 (*Lenny*) was used with the modified kernel. The network subsystem of the Linux kernel follows a layered structure [32]. These layers are demonstrated in Figure 3.2.

First of all, the drivers of the various devices can be found at the lowest layer which provide low-level access to the hardware directly. The next layer consists of the network layer protocols, such as *IP* (Internet Protocol), *ICMP* (Internet Control Message Protocol), *IGMP* (Internet Group Management Protocol) and *RIP* (Routing Information Protocol). The next upper layer contains the transport layer protocols, for example *TCP* and *UDP* (User Datagram Protocol). Furthermore, the new protocol (*DFCP*) can also be found at this layer. The next layer, also known as the socket layer, consists of the different kinds of *network sockets* which provide a protocol-independent interface to the user applications. All the applications use the interface offered by the socket layer. A socket represents one endpoint of a two-way communication link between two programs running on the Internet network.

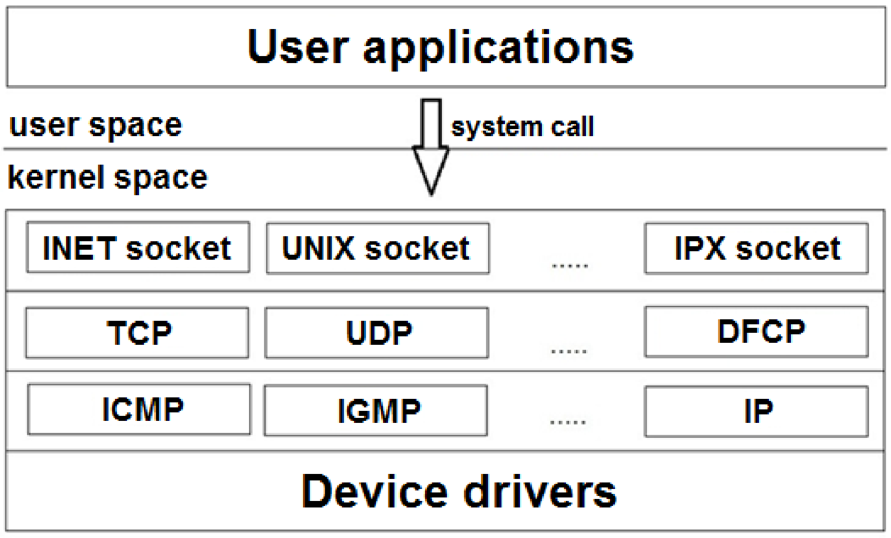


Figure 3.2. The structure of the network subsystem

A socket pair (a pair of connected sockets) represents the two ends of a communication channel. In the case of two communicating processes, both processes have their own sockets which are used to identify the connection on both sides. A number of different types of sockets are supported by Linux. These sockets are classified into *address families*, such as *AF_ UNIX*, *AF_ INET*, *AF_ AX25* and *AF_ IPX*. The connection-oriented protocols, such as TCP and DFCP, are supported by the *AF_ INET* address family.

The memory, used by an operating system, is separated into two parts which are called *kernel space* and *user space* [33]. The kernel space memory is reserved for the kernel and it contains the Linux kernel itself and its internal data structures. The kernel code is always executed in the privileged mode of the processor. The user space is used by the user applications. In the figure above, it can be seen that the user applications communicate with the kernel by using *system calls*. In the kernel, each system call is associated with a unique number and a function that is called when the system call is invoked. A list of all registered system calls is maintained in the *system call table*. The system call numbers are also listed in the file */usr/include/asm/unistd.h* for the user applications. Knowing the number of the system calls, the applications are able to identify the operation to be performed. For instance, in order to connect to a remote server, the *connect()* function is invoked by the user application. The *connect()* library function invokes a system call to request the kernel to perform the operation. The kernel then looks up the system call number in the system call table and invokes the appropriate system call handler, *sys_ connect()*, with the parameters specified by the user application. This process is similar in the case of data sending (*write()* function), data receiving (*read()* function), connection termination (*close()* function) and other operations.

3.4 The basic operation of the Digital Fountain based Communication Protocol

The DFCP applies the previously described concept, and therefore it does not employ any congestion control mechanism. In the first step, a connection needs to be established between the sender and the receiver. The connection establishment is very similar to the three-way handshake mechanism of TCP. When the connection has been established, the sender is allowed to begin sending its data. In the kernel, during the connection establishment, *queues* of the appropriate size, which is specified by a given parameter, are allocated that are used in the course of the data sending and receiving. Afterwards, the data sent by the application is stored in a free queue by the kernel. The stored data is then encoded and sent out. The encoding is always applied on a given amount of data which is referred to as a *block*. A queue stores all the data that belongs to a certain block. The DFCP uses a specific header format which is used to forward coding-specific information between the two interacting endpoints. In the course of the data sending, data retransmission is not employed at all. However, acknowledgements are used to indicate the successful decoding of a given block. The receiver waits until the sufficient amount of data, set by a certain parameter, has been received in order to successfully decode the encoded data and restore the original message with high probability. The given data block is then decoded and an acknowledgement is sent back to the sender. When the acknowledgement has been received by the sender, it is able to determine the next block to send. In this case, the memory used by the acknowledged block is freed on the sending side so that the receiver can store a new block in the queue. On the receiving side, the decoded data is passed to the appropriate application. Finally, if there is no more data to send, the connection has to be terminated, which also happens in a similar fashion to how TCP terminates a connection.

In the next section, the header structure of DFCP is discussed. Afterwards, the operational phases and the parameters of the protocol will be introduced.

3.5 DFCP header structure

The structure of the DFCP header is shown in Figure 3.3. The numbers in parentheses indicate the number of bits in each header field. The first two fields are the *port numbers* which identify the sending and the receiving applications (or processes), respectively. The port numbers are used to distinguish between the different network applications running on the same computer. The *Block ID* field is used to identify the block a given segment belongs to. The *S1*, *S2* and *S3* header fields contain 32-bit unsigned integer values and their exact meaning will be discussed in a later section about the encoding and decoding mechanisms. The *data offset* field specifies the length of the DFCP header in 32-bit words. Equivalently, it is also the offset from the start of the DFCP segment to the actual data.

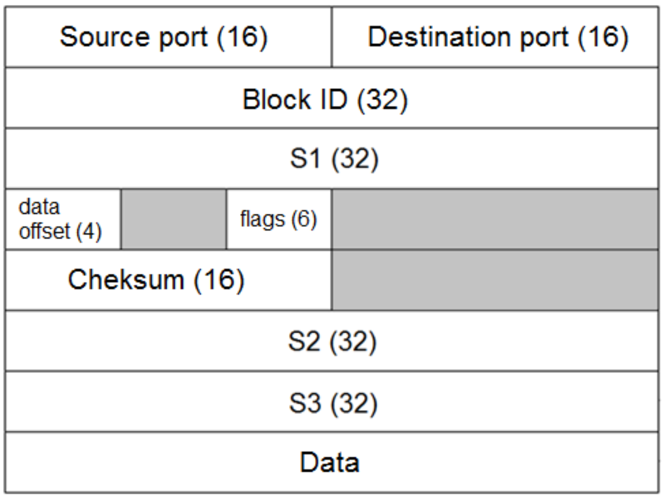


Figure 3.3. DFCP header format

In addition, the DFCP header contains several one-bit boolean fields, known as *flags*, which are used to control the flow of data across a connection. There are six DFCP control flags. Most of these, similarly to TCP, are used to control the establishment, maintenance and tear-down of a connection. For example, the *SYN* flag is used to establish a connection whereas the *FIN* flag is used to close a connection. The exact meaning of these flags will be explained in a later section about the operational phases of the protocol. Hereafter, when referring to a segment, the flags set will be indicated by putting their names before the word "segment". For instance, the phrase *SYNACK segment* refers to a segment with both the *SYN* and *ACK* flags set whereas all the other flags are unset. Finally, the 16-bit *checksum* field is used to check the integrity of the segment including both the header and the data.

3.6 DFCP connection establishment

In this section, the connection establishment process of DFCP will be presented which is very similar to the three-way handshake mechanism applied by TCP.

The connection establishment of DFCP can be divided into three steps. The first step is represented in Figure 3.4. During the connection establishment, the connection initiator sends a *SYN segment*. The header of this segment contains coding-related information that is required for the receiver to successfully decode the encoded data. The further details are given in a later section about the encoding and decoding processes of DFCP. In the course of the connection establishment, the appropriate state of the connection is maintained on both sides. After having sent the *SYN segment*, the initiator sets its state to *SYN_ SENT*. If the *SYN segment* is lost, it will be retransmitted by using a timer. The initial value of the timeout is set to one second. If there is no answer received from the other side until the timer expires, the *SYN segment* will be resent. In Figure 3.4, it is assumed that if the timer is not expired, the *SYN segment* has successfully been received by the other side, and consequently the second phase of the process can be entered. In this case, *s* refers to the sender. The value of the timeout is doubled every time a segment is resent. After five unsuccessful retransmission attempts, the connection establishment process is aborted and the resources are released at the sender.

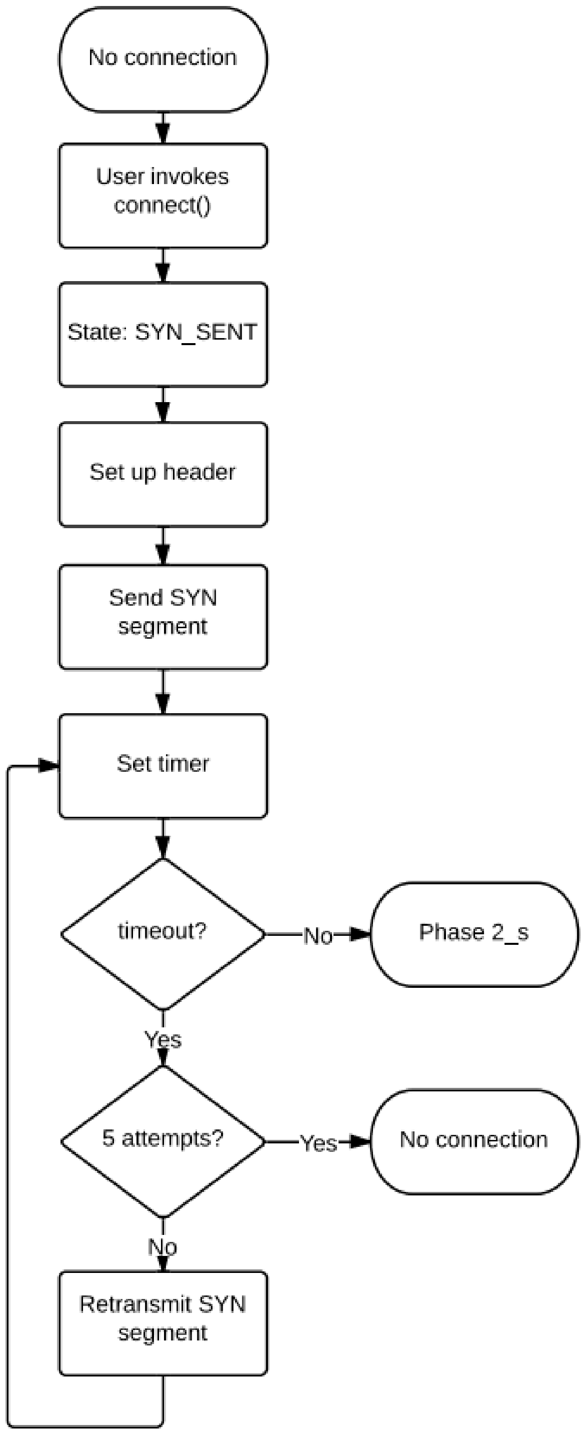


Figure 3.4. The first step of the connection establishment

In the course of the second step, depicted in Figure 3.5, the receiver processes the *SYN segment*, sets its state to *SYN_ RECV* and sends a *SYNACK segment* in response to the *SYN segment*.

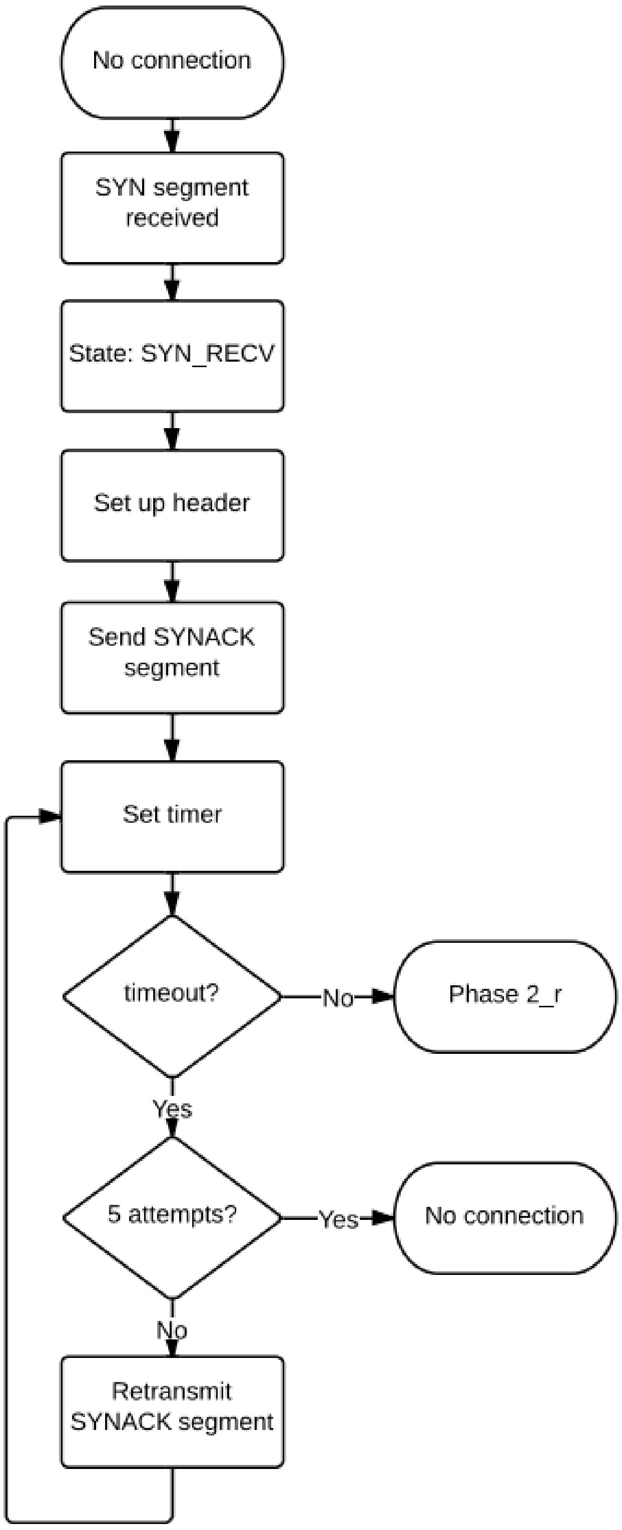


Figure 3.5. The second step of the connection establishment

The header of the *SYNACK segment* also contains coding-related information which can later be used by the other side in order to decode the encoded data. In the figure, r refers to the receiver. In the case of the *SYNACK segment*, similarly to the first step, timers are used to retransmit the lost segments. After five unsuccessful retransmissions, the connection establishment process is aborted and the resources are released.

Finally, the third step is demonstrated in Figure 3.6. The flowchart on the left side shows the actions taken by the connection initiator while on the right side, we can see the operations performed by the connection acceptor. In the course of this step, firstly, the initiator processes the *SYNACK segment*. Secondly, its state is set to *ESTABLISHED*. Finally, it sends an *ACK segment* to the other side. The other endpoint processes the *ACK segment* and sets its state to *ESTABLISHED* as well. In the case of the *ACK segment*, it is not necessary to use timers so as to retransmit the lost segments because if the segment is lost, the other side will resend the *SYNACK segment* due to the expiry of the *SYNACK timer*. If the number of the allowed retransmission attempts, usually set to five, has been reached, the endpoint that sent the *SYNACK segment* terminates the connection on its side. In this case, the other endpoint can be notified of the unsuccessful connection establishment by an *RST segment*, and consequently its resources can also be released.

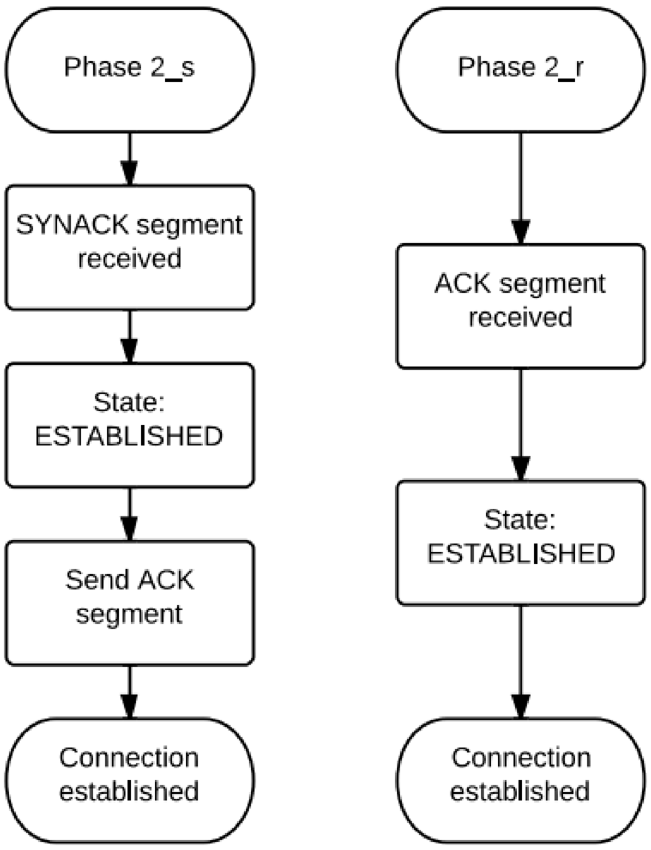


Figure 3.6. The third step of the connection establishment

3.7 The data encoding process of DFCP

In this part, the efficient erasure coding scheme employed by the protocol will be introduced. In general, *Raptor coding* is the concatenation of applying an *outer code*, also known as *pre-code*, and an *LT code*. In the case of DFCP, the applied Raptor coding can be divided into two separate stages. Firstly, an *LDPC code* (Low-Density Parity-Check) [34] is employed as the outer code and secondly, an *LT code* is used as the *inner code*. This section is split into three subsections. In the first subsection, a brief overview of the data encoding process is provided. In the next two subsections, the LDPC and the LT coding applied by the protocol will be discussed in detail, respectively.

3.7.1 An overview of the data encoding process

When a user application invokes the *write()* system call the data sent by the application is passed to the kernel. The kernel stores all the user data in a data structure called *send_ queue*. In the kernel, a list of *send_ queues* is pre-allocated during the connection establishment process. When the kernel has processed the user data, the data will be stored in an empty *send_ queue*. The applied coding scheme is illustrated in Figure 3.7.

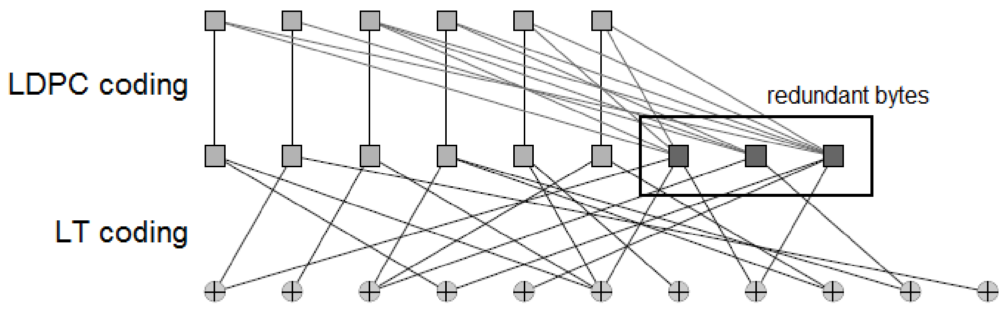
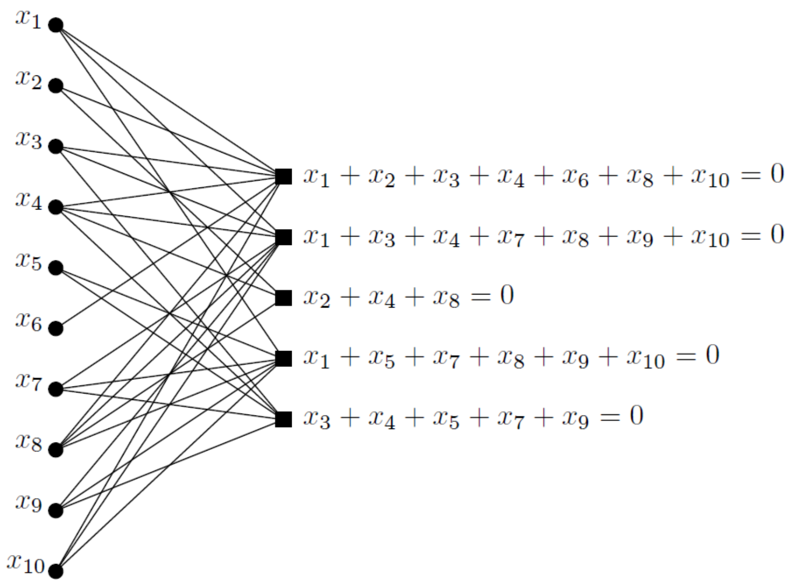


Figure 3.7. The applied Raptor coding scheme

In order to be able to guarantee the successful decoding of the encoded data with high probability after having received only slightly more encoded symbols than the original (uncoded) message, firstly, a traditional linear block coding is applied as pre-coding, called *LDPC coding*. In the course of both the encoding and the decoding, each symbol represents a byte (8 bits). As a result, henceforth, a symbol is equivalent to a byte. If the user application has sent k bytes of (uncoded) data, the kernel applies the LDPC coding and generates n bytes of LDPC encoded data since redundant bytes are appended to the original message of k bytes. The number of the redundant bytes can be calculated as $n - k$. In the case of the current implementation, the constant value of 2000 redundant bytes is used which, according to the experience gained so far, is sufficient to guarantee the successful decoding with relatively high probability. The generated n bytes constitute the output of the LDPC coding which, in turn, will be the input of the LT coding that, in theory, is able to produce an infinite-length stream of LT encoded bytes.

3.7.2 The LDPC encoding process

According to the definition of LDPC codes, they are linear codes obtained from sparse bipartite graphs. Suppose G is a bipartite graph with n left nodes and r right nodes. The left nodes are often referred to as *message* or *variable nodes* while the right nodes are referred to as *check nodes*. An example graph can be seen in Figure 3.8.



x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
x_9
x_{10}
$x_1 + x_2 + x_3 + x_4 + x_6 + x_8 + x_{10} = 0$
$x_1 + x_3 + x_4 + x_7 + x_8 + x_9 + x_{10} = 0$
$x_2 + x_4 + x_8 = 0$
$x_1 + x_5 + x_7 + x_8 + x_9 + x_{10} = 0$
$x_3 + x_4 + x_5 + x_7 + x_9 = 0$

Figure 3.8. An LDPC code

It can be seen that, for each check node, the sum (calculated by performing XOR operations) of its neighbors among the message nodes is zero. In the case of the current implementation, LDPC codes are generated based on a given probability distribution and the initial value of every check node is zero. The encoding process is illustrated in Figure 3.9. Firstly, for each message node, according to the given probability distribution, a random value, d, is generated that constitutes the degree (alternatively, the number of the neighbors) of the given message node. Afterwards, d check nodes are chosen according to the uniform distribution. These check nodes are chosen to be the neighbors of the given message node, and therefore (represented in the figure as “Set check node”) their values are XORed with the value of the current message node as shown in Equation (3.1).

$$cknode[rand] = cknode[rand] \oplus msg[i] \tag{3.1}$$

Here, *cknode[rand]* denotes the selected check node while *msg[i]* refers to the current message node. In the course of the LDPC encoding, the message nodes are associated with the bytes of the user data. As a result, each byte of the user data corresponds to a message node. In the case of the current implementation, r is set to the constant value of 2000, and therefore 2000 check nodes are produced during the LDPC encoding. Each data block contains $k = 63536$ (uncoded) message bytes. Since the generated check nodes are appended to the original message the value of n is 65536. These 65536 bytes constitute the input of the LT encoding.

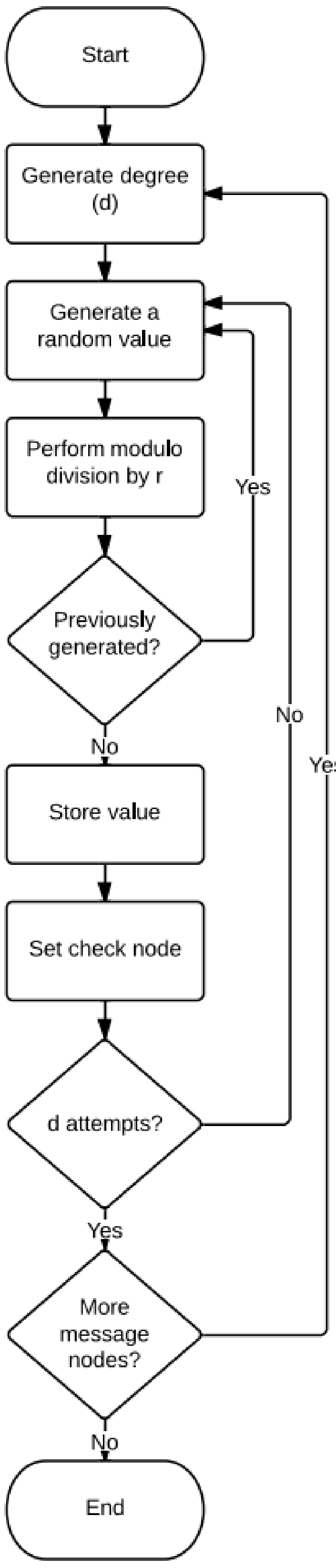


Figure 3.9. The applied LDPC encoding

3.7.3 The LT encoding process

Henceforth, n denotes the number of bytes which represent the input of the LT encoding. The LT encoded output stream is generated using the n input bytes. In order to generate the output stream of LT encoded bytes, an $\Omega_0, \Omega_1, \Omega_2, \ldots, \Omega_n$ probability distribution is defined on $\{0, 1, 2, \ldots, n\}$. In addition, associations are specified between the input bytes by performing XOR operations. For each output byte, its degree, denoted by D, determines how many input bytes have been XORed to generate the given output byte. This procedure is demonstrated in Figure 3.10. It can be seen in the figure that the first output symbol is associated with two source symbols, and therefore its degree is 2 whereas the last output symbol is associated with one source symbol, and consequently its degree is 1.

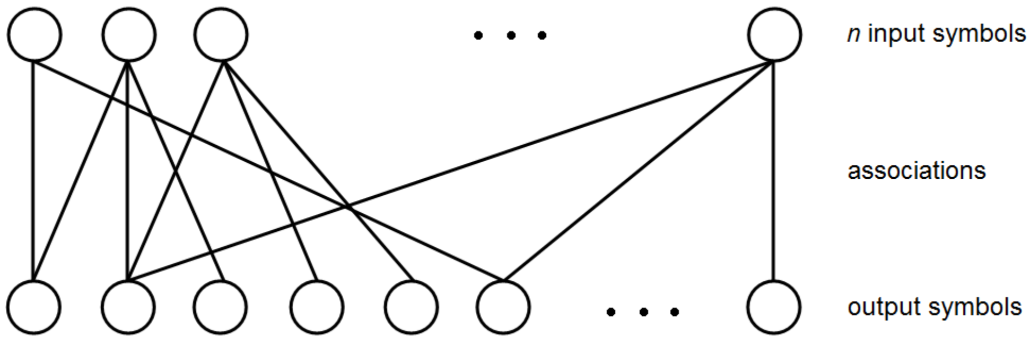


Figure 3.10. Associations are defined between the source symbols

In this case, for each i, $\Omega_i = P(D = i)$. These Ω_i values, which are used in the case of the current implementation of the protocol, are shown in the second column ($k = 65536$) of Table 3.1. In the table, n refers to the number of input symbols, which is currently $n = 65536$. All the additional Ω_i values, which are not given in the table, are assumed to be zero.

The LT encoding process itself is divided into three phases. In order to implement the first phase of the algorithm, the generation of random numbers in the interval [0,1] would have been required according to the uniform distribution. However, for performance-critical reasons, floating-point operations are not supported in the Linux kernel. So as to solve this issue, the $0 \leq \Omega_i \leq 1$ values were converted into integer numbers performing a simple transformation and the converted numbers were later used in the kernel in order to generate the required random numbers. The sum of all the Ω_i values is approximately 1. In the kernel, the use of 32-bit unsigned integer values was necessary, which are in the interval $[0, \ldots, 2^{32} - 1]$. The first step of the transformation was a multiplication by a constant factor c, which can be calculated as shown in Formula (3.2).

$$c = \frac{2^{32} - 1}{\sum_{i=0}^{n} \Omega_i} \tag{3.2}$$

Each Ω_i was multiplied by the above constant c and, in the second step of the transformation, the result was rounded to the nearest integer. In the case of Ω_2, this integer was decreased by one so as to exactly fit the interval $[0, \ldots, 2^{32}-1]$ with the minimal distortion of the original intervals. The exact Ω_i values before performing the described conversion can be seen in the second column of Table 3.1 while the third column shows the new values after executing the conversion.

Table 3.1. The integer conversion of the Ω_i values

Ω_i **values for** $n = 65536$	**Before the conversion**	**After the conversion**
Ω_1	0.007969	34226663
Ω_2	0.493570	2119871247
Ω_3	0.166220	713910892
Ω_4	0.072646	312012818
Ω_5	0.082558	354584619
Ω_8	0.056058	240767758
Ω_9	0.037229	159897657
Ω_{19}	0.055590	238757710
Ω_{65}	0.025023	107473182
Ω_{66}	0.003135	13464749

In order to generate the required 32-bit unsigned integer random numbers, a maximally equidistributed combined *Tausworthe generator* [35] is used, which is a fast multiplicative recursive generator and has good statistical properties. In addition, it is part of the Linux kernel. The state of the generator is stored in a data structure which contains three integer variables named *s1*, *s2* and *s3*.

In the first step of the LT encoding process, a random number is generated using the Tausworthe generator described above. All the random numbers are currently generated according to the uniform distribution, however, the performance of the coding might later be improved by changing the probability distribution. The generated random number is a 32-bit unsigned integer. After the generation, it is determined to which Ω_i interval the number belongs. In the end, the result of the first step is a d integer number indicating the degree of the output symbol according to the determined interval.

The second step of the LT encoding is represented in Figure 3.11. First of all, d bytes are chosen from the n input bytes according to the uniform distribution. So as to accomplish this task, it is necessary to generate d different random numbers. All of these numbers must be from the interval $[0, \ldots, n-1]$. The reason for this is that the sequence number of the first input byte is 0 while the sequence number of the last byte is $n-1$. In order to guarantee that all the random numbers are different, a modulo division by n is performed in the case of each random number. The resulting d numbers, which are all different, will indicate the indices of the bytes which are associated with each other so as to produce an LT encoded output byte. The difference of all the random numbers is ensured by storing all the previously generated values and comparing the currently generated number to the previous ones. If the currently generated value is not different from the others, it is dropped and a new random value is generated.

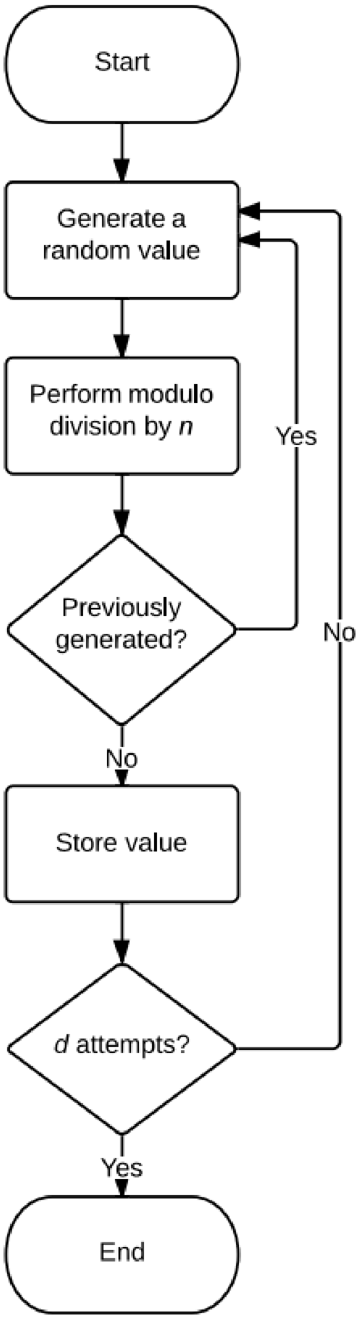


Figure 3.11. The second step of the LT encoding process

In the third step, we use the d random numbers generated during the previous step. The d chosen input bytes are XORed with each other. As a result, a Y output byte is produced. More precisely, we compute $Y = X_{s1} \oplus X_{s2} \oplus \ldots \oplus X_{sd}$ where X_{si} refers to the i^{th} chosen byte. In this case, bitwise XOR operations are performed on the bits of each input byte. Therefore, $\{u_1, u_2, \ldots, u_n\} \oplus \{v_1, v_2, \ldots, v_n\} = \{u_1 \oplus v_1, u_2 \oplus v_2, \ldots, u_n \oplus v_n\}$. The output of the third step is a Y encoded byte. To sum up, the input of the LT encoding process consists of n symbols while its output is one encoded byte. By repeating these three steps, in theory, it is possible to generate an infinite-length (or an arbitrary-length) stream of LT encoded bytes. After having finished encoding the user message, the resulting encoded stream will be sent out, which is discussed in the next section.

3.8 The data sending process of DFCP

After having described the data encoding process, in this section, the data sending process will be introduced. Firstly, the structure of a *send_ queue* is explained, which is used to store the data temporarily before it is sent. Afterwards, the data sending process is presented. Finally, the use of some specific DFCP header fields is discussed.

3.8.1 The structure of a *send_ queue*

In the case of each connection, during the connection establishment, a certain number of queues are allocated, specified by a parameter, which are later used to temporarily store the data sent and received. A queue that is used to store the data sent by the user is referred to as a *send_ queue* whereas a queue that stores the data the kernel received is called a *recv_ queue*. In short, the kernel stores the content of a block sent by the user in a *send_ queue* temporarily. The data is stored until the kernel receives an acknowledgement for the given block. Apart from the data itself, a *send_ queue* contains numerous other variables which are used in the course of the data sending process. The structure of a *send_ queue* is depicted in Figure 3.12.

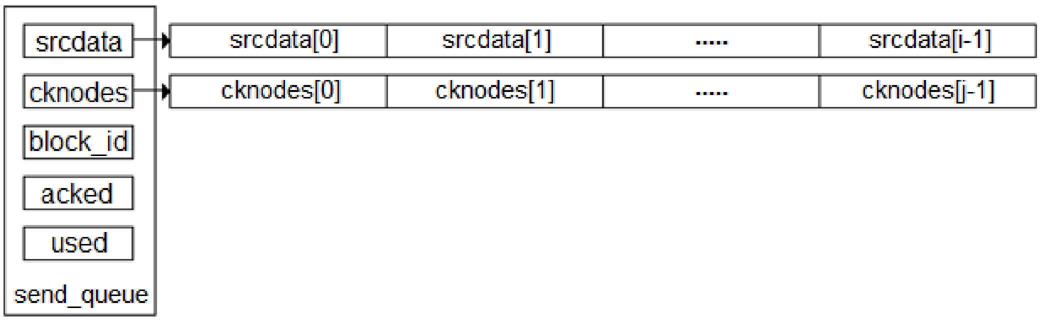


Figure 3.12. The structure of a *send_ queue*

Firstly, it stores the data sent by the user in the field *srcdata*. More precisely, this field contains a block of data which consists of 63536 bytes. If the user message is shorter than 63536 bytes, it will be padded with zero bytes. Secondly, the 2000 check nodes, which are generated in the course of the LDPC encoding, are stored in the field *cknodes*. In addition, each block has a unique identifier stored in the field *block_ id*. The block identifier is a non-negative integer. The first block always has the block identifier 0. For each of the following blocks, the identifier is increased by one. Furthermore, the field *acked* signifies whether the current block has already been acknowledged or not. Finally, the field *used* indicates if the given *send_ queue* is empty or not.

3.8.2 The data sending process

As described in the previous subsection, the data sent by the user application is temporarily stored in an empty *send_ queue* by the kernel. If there is no empty *send_ queue* available, the user process has to wait (sleep) until a *send_ queue* is released. After having stored the user data, the LDPC encoding process will be performed. The data sending process employs a *sliding window* mechanism, and therefore the sender is allowed to send a specified number of blocks, set by a certain parameter, without waiting for acknowledgements. The maximum size of the window determines how many *send_ queues* are allocated during the connection establishment. The window consists of the used (not empty) *send_ queues*. The data sending happens in a *round-robin* fashion. In the case of each used *send_ queue*, the user data is LT encoded and a DFCP segment containing 1420 bytes of LT encoded data is generated and sent out. When a segment has been sent out, the same process continues for the other used *send_ queues*. In theory, for each used *send_ queue*, an infinite number of segments are produced. In practice, a *send_ queue* is freed when an acknowledgement has been received for the given block allowing the user application to send additional data. This sending procedure guarantees that even if a large number of segments are lost, the receiver is able to restore the original message. As soon as the receiver has collected a given number of LT encoded bytes, it sends an acknowledgement for the block to the sender. If the acknowledgement has been lost, the receiver regenerates and resends it when additional segments are received for the same block.

The steps of the data sending process are presented in Figure 3.13. Firstly, it is determined which *send_ queue* will be used to store the data sent by the user and the block-related information. If all the queues are currently used, the user application has to wait. The waiting ends when the kernel has received an acknowledgement for a block that is currently stored in a *send_ queue*, and therefore the given *send_ queue* can be freed. The user data is then copied to the *send_ queue* and the LDPC encoding is performed. In the next step, a block is chosen to be sent according to the round-robin algorithm. The selected block can be different from the block currently sent by the user. Subsequently, a DFCP segment is allocated which is able to contain 1420 LT encoded bytes. In order to understand the size of the DFCP segments, we need to recall the definition of MTU. The MTU is the maximum amount of data that can be transmitted in a frame or in a packet by a given protocol. For example, the MTU of Ethernet is the largest number of bytes that can be carried by an Ethernet frame not including the header and the trailer. As a result, the MTU of Ethernet is 1500 bytes by default. By generating a DFCP segment which contains 1420 bytes of encoded data, the size of the IP packet that carries the segment will be very close to the MTU, and therefore the network resources will be used efficiently avoiding fragmentation. After allocating a segment, the DFCP header fields are set appropriately. This step is discussed in detail in the next subsection. Afterwards, the LT encoding process is executed. The three steps of the LT encoding are repeated until 1420 encoded bytes are produced. Finally, the DFCP segment is passed to the IP layer for further processing. The additional segments are generated in the same way until the kernel receives an acknowledgement for a given block. When an acknowledgement has been received, the kernel determines the *Block ID* of the acknowledged block and frees the associated *send_ queue*. Furthermore, the sending application will be notified of the successful transmission of the block by the kernel setting the return value of the *write()* function correctly.

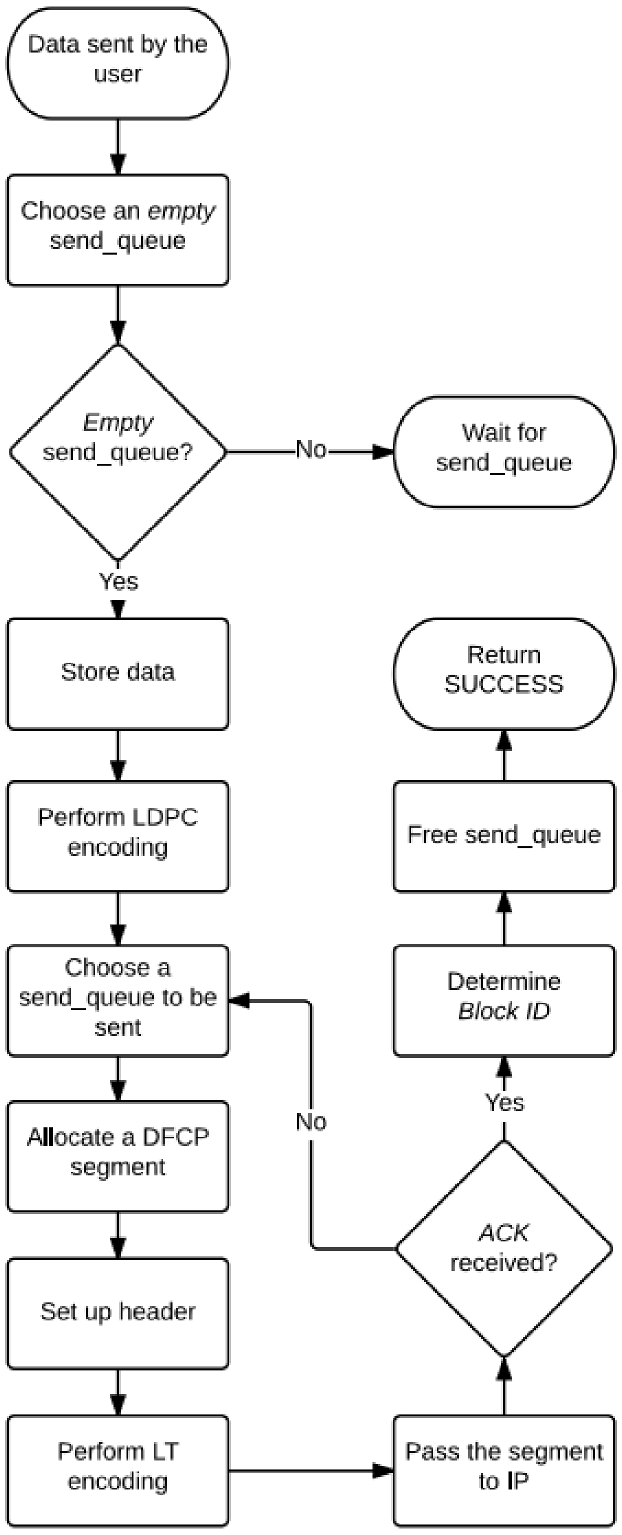


Figure 3.13. The data sending process

3.8.3 The use of the DFCP header fields

The random numbers produced by the sender so as to encode the original user message have to be regenerated at the receiver in order to be able to decode the encoded message successfully. Two different approaches are carried out to regenerate the random numbers used during the LDPC and LT encoding.

In the case of the LDPC encoding, in the course of the connection establishment, the header fields of the *SYN segment* contain three 32-bit unsigned integers (*s1*, *s2* and *s3*) which represent the initial state of the random number generator at the connection initiator. This state can be queried and set by invoking the appropriate kernel functions. When the other endpoint has received the *SYN segment*, it stores these three numbers and, in response to the *SYN segment*, it allocates a *SYNACK segment* and sets up the header fields querying the state of its own random number generator. Similarly, after having received the *SYNACK segment*, the initiator also stores the three 32-bit values which represent the initial state of the random number generator at the connection acceptor. During the LDPC encoding process, all blocks are encoded performing a transformation based on the three variables that describe the initial state of the generator. Since both sides have exchanged the initial state of their own random number generators, they will both be able to regenerate the random numbers used by the other endpoint in the course of the LDPC encoding process, and therefore they will be able to decode the encoded data.

More precisely, when performing the LDPC encoding process, a particular kernel function is invoked which sets the state of the generator based on the initial state and the *Block ID* of the block that is currently being encoded. After having set the new state, the check nodes are generated as described in the section about the LDPC encoding procedure. The other endpoint also knows the initial state of the generator, which is stored in kernel variables, the function, which implements the transformation, and is able to determine the *Block ID*. Knowing this information, the data receiver is able to set its generator to the same state as the sender, and consequently it is able to generate the same random numbers in the course of the LDPC decoding.

In the case of the LT encoding, we set up the header of the DFCP segment after having allocated it as depicted in Figure 3.13. According to the current implementation, for each DFCP segment, the state of the generator (*s1*, *s2* and *s3*) is queried before the actual LT encoding is performed. These three variables will later be copied to the appropriate header fields of the given DFCP segment and the segment will be sent out. Based on the information in the header fields, the receiver is able to set its random number generator to the same state as the sender and produce the same random numbers. Additionally, the *Block ID* is forwarded in the *Block ID* header field of the segment, and therefore the receiver is able to determine to which block the encoded bytes in the received segment belong.

3.9 The data receiving process of DFCP

In this section, the data receiving process is discussed in detail. Receiving the data and decoding the encoded data are two entirely separate functions in the kernel. The reason for this is that the data receiving procedure has to be incredibly fast, and therefore only very few operations can be performed for each received segment. The data decoding process and the related structures in the kernel are introduced in the next section.

When the user application invokes the *read()* system call, the kernel attempts to decode the current block. The *Block ID* of this block is stored in a kernel variable that is always incremented by one after having decoded a block. Consequently, the *Block ID* of the first block that will be decoded is 0. Afterwards, the variable is always incremented by one and the kernel will attempt to decode the next consecutive blocks (1, 2, . . .). This mechanism ensures the correct order of the blocks in the course of the data receiving process. If the kernel has not received the required amount of data for the current block yet, set by a certain parameter, the application has to wait until the kernel has received the necessary amount of data to perform the decoding process in the case of the given block.

The data receiving process is represented in Figure 3.14. Similarly to the data sending process, the block management information, which is used by the kernel in the course of the data decoding procedure, and the encoded data in the segments received by the kernel will be stored in a *recv_ queue* during the data receiving process. For instance, a *recv_ queue* contains the number of the encoded bytes which have been received by the kernel in the case of the given block. The structure of a *recv_ queue* will be discussed in the next section about the data decoding process.

When a segment has been received, firstly, it is determined to which block it belongs. This is performed by reading the value stored in the *Block ID* field of the DFCP header. In the next step, the kernel performs certain sanity checks on the *Block ID*. If the ID is found to be invalid, the segment is dropped. Afterwards, if a *recv_ queue* has already been associated with this block, the kernel will use the associated *recv_ queue* in the case of all the additional segments received for this block. Otherwise, two separate cases are possible. The kernel is either able to find a new empty *recv_ queue* for this block or there is no empty *recv_ queue* available at the moment. In the second case, the segment is dropped while in the first case, one of the empty *recv_ queues* will be associated with the new block. In the case of the associated *recv_ queue*, the variable that counts the number of the encoded bytes received for this block will be increased by 1420 as one DFCP segment consists of 1420 encoded bytes. The encoded bytes will be stored in a linked list and they will be processed in the course of the data decoding process. Afterwards, the kernel checks whether the required number of encoded bytes has been received or not. If the kernel has already received the necessary number of encoded bytes and the user process is currently sleeping (waiting for data), the kernel wakes up the process. Subsequently, the encoded bytes are decoded and finally, the user application will receive the decoded message.

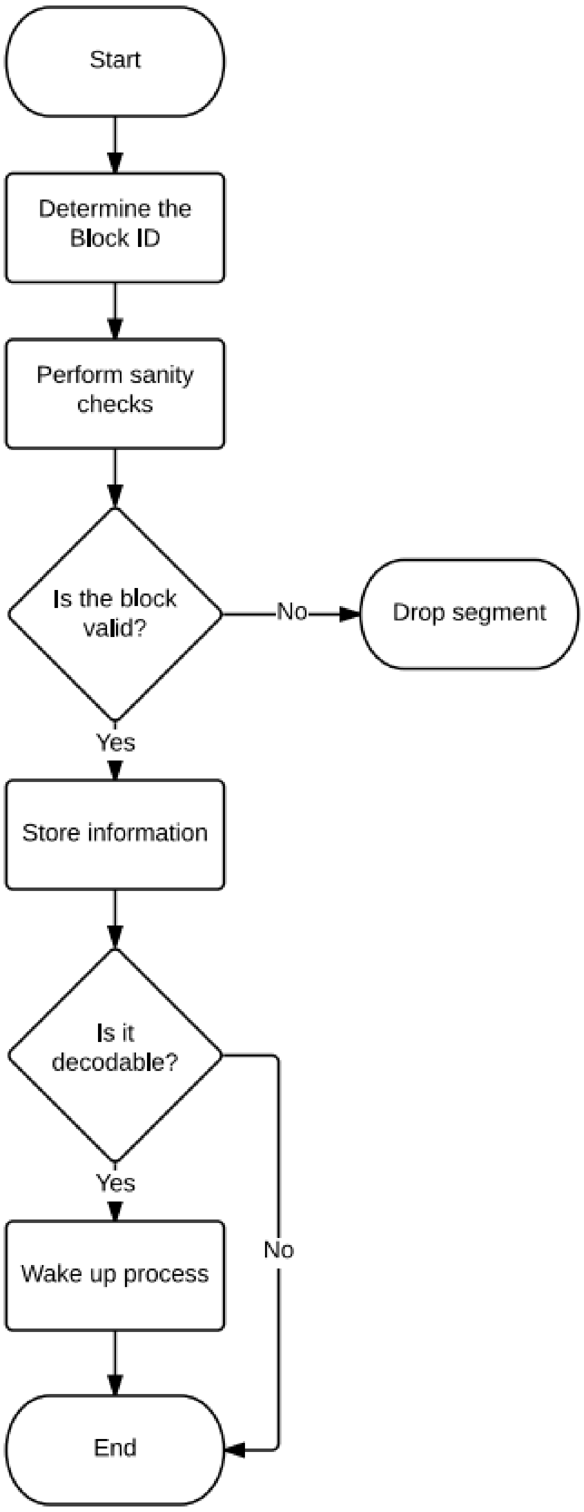


Figure 3.14. The data receiving process

3.10 The data decoding process of DFCP

In this section, firstly, the structure of a *recv_ queue* will be explained. Afterwards, the LT decoding and the LDPC decoding processes will be discussed in detail.

3.10.1 The structure of a *recv_ queue*

In the course of the connection establishment, the number of the *recv_ queues* allocated for the connection is specified by a certain parameter. In the case of the current implementation, the number of the *recv_ queues* cannot be changed later. When a segment has been received by the kernel, the block for which the segment was received is associated with an empty *recv_ queue*. This association is made when a segment is received with which there has not yet been a *recv_ queue* associated and there is at least one empty *recv_ queue* available so as to accumulate the content of the segment. A *recv_ queue* consists of several variables which are used in the course of the data decoding process. Moreover, the decoded data itself is also stored in the *recv_ queue* until it has been read by the user application.

The structure of a *recv_ queue* is depicted in Figure 3.15. The first three fields (*tnodes*, *failed*, *needed*) are used in the course of the LDPC decoding while the next three fields (*lists*, *udata*, *idx*) are used during the LT decoding. The purpose of these six fields is discussed in a later section about the LDPC and the LT decoding processes. The *count* field indicates the number of the encoded bytes which are currently stored in the given *recv_ queue*. Furthermore, the *num* field stores the number of bytes which have been successfully decoded. This field has an important role since more than one iteration might be required in the case of the decoding procedure until the entire user message has been restored. The *recv_ queue*, which was allocated for the given connection in the course of the connection establishment, is deallocated during the connection termination.

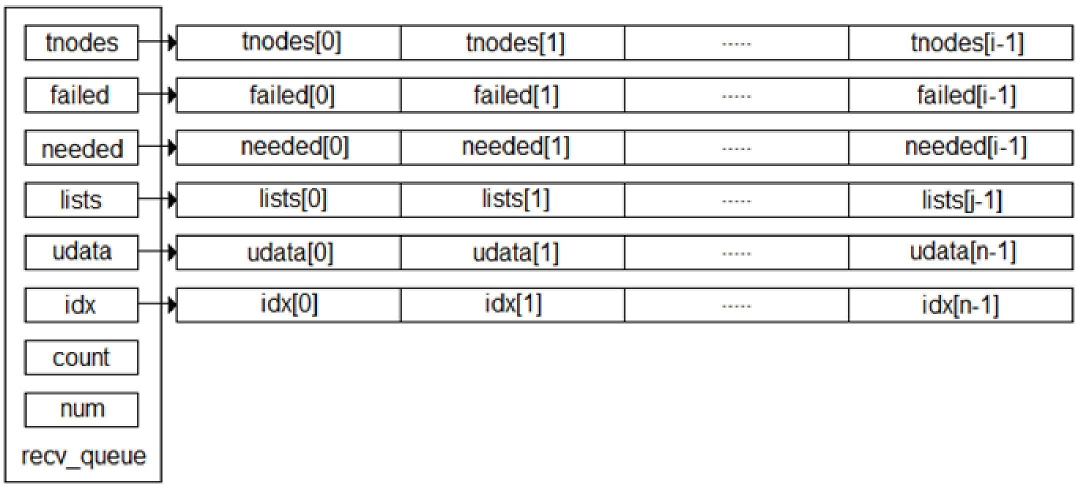


Figure 3.15. The structure of a *recv_ queue*

3.10.2 The LT decoding process

In the earlier section about the data receiving process, it was explained that the content of a received segment is stored in a linked list. Before the LT decoding procedure is performed, the linked list, which consists of encoded blocks of data, is processed. More precisely, the linked list is processed and the LT decoding is performed while the *read()* system call, invoked by the user application, is being executed.

During processing the linked list, for each stored segment, the encoded bytes are examined and every encoded byte is copied to another data structure (*lists*) which also consists of linked lists, however, it is organized to facilitate the data decoding procedure. A pointer to this data structure is stored in the *recv_ queue* which is associated with the given block. When the kernel has received the required number of bytes to decode the encoded data with high probability, the decoding process is performed. A certain kernel variable is used to specify the number of encoded bytes which are necessary to decode a given block with high probability. This parameter was determined in a heuristic manner and can be set in the course of the connection establishment. During the decoding process, firstly, the LT decoding is performed and afterwards, the LDPC decoding is executed. Consequently, during the LDPC decoding, both the restored bytes of the user message and the decoded check nodes can be used in order to recover the additional bytes of the user message. The LT and the LDPC decoding processes are executed in this order until neither of them is able to recover additional bytes of the user message. At this stage, two scenarios are possible. Firstly, all the bytes of the user message have been successfully restored. In this case, the decoding process ends with success. Secondly, there has been at least one encoded byte which cannot be decoded, in which case the decoding process ends with failure. In both cases, an acknowledgement is sent for the given block by the kernel so as to inform the other endpoint that the *send_ queue* associated with the block can be freed. The *Block ID* field of the acknowledgement contains the sequence number of the decoded block. After having sent the acknowledgement, the decoded user message is passed to the application that invoked the *read()* system call. Finally, the *recv_ queue* associated with the given block can be freed at the receiver as well. As a result, the empty *recv_ queue* can be used to receive additional blocks of data. If the decoding process has ended with failure, an acknowledgement is also sent for the block so that the associated *send_ queue* can be freed at the sender, however, the kernel returns a value that indicates a decoding failure to the receiving user application.

As depicted in Figure 3.15, the second three fields (*lists*, *udata*, *idx*) of the *recv_ queue* are used in the course of the LT decoding. The *lists* field contains a pointer to a complex data structure which consists of various lists. In the case of the current implementation, the value of j is set to 10, and consequently 10 lists are allocated. This allocation is done based on the number of bytes associated with each other so as to generate an LT encoded byte. In the case of $n = 65536$, the 10 possible values, which are 1, 2, 3, 4, 5, 8, 9, 19, 65 and 66, are represented in Table 3.1. To be more precise, for instance, the encoded bytes on the first list are associated with one message byte whereas the encoded bytes on the last list are associated with 66 message bytes.

In Figure 3.16, the structure of a list is presented. Each list consists of two pointers. The *head* pointer points to the first element on the list while the *tail* pointer points to the last element on the list. As a result, appending a new element to the list can be done without iterating through each item on the list since the new element will simply be appended after the element the *tail* pointer currently points to.

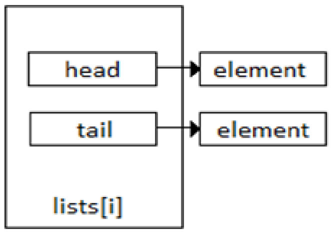


Figure 3.16. The structure of a list

Finally, at the lowest level of the hierarchy are the individual list elements. The structure of a list element is depicted in Figure 3.17.

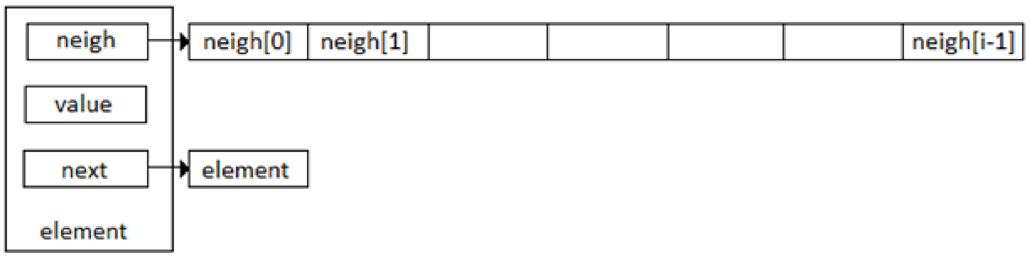


Figure 3.17. The structure of a list element

Each list element contains a pointer (*next*) which points to the next element on the list. The *value* field stores the value of one encoded byte out of the encoded bytes received in a segment. The *neigh* field contains the indices of the message bytes which have been associated with each other so as to generate the encoded byte denoted by *value*. The length of this array is different in the case of the certain lists, however, it is always the same in the case of the elements on a given list. As a result, for each element on the first list, there was only one message byte in the association in the course of the LT encoding. In the case of the second list, two message bytes were associated with each other in order to generate an encoded byte. Finally, in the case of the last list, 66 message bytes were associated with each other assuming that $n = 65536$. The list elements are generated in the following way. For each received segment, the header is processed and the random number generator at the receiver is set to the appropriate state based on the values in the previously mentioned header fields (*s1*, *s2* and *s3*). The sender queries the state of its generator before performing the LT encoding process, and consequently the receiver is able to generate exactly the same random numbers as the sender. Although certain segments can be received out of order, based on the values in the header fields of each segment, the original user message can be restored, and therefore the order of the segments at the receiver is inconsequential. To sum up, the number and the indices of the bytes which have been associated with each other in the course of the LT encoding procedure are determined based on the information in the header fields of the DFCP segments.

The *udata* field of the *recv_ queue* contains the successfully decoded bytes of the user message. Furthermore, the values of the restored check nodes are also stored in this field. Therefore, the value of *n*, as depicted in Figure 3.15, is 65536 since 2000 check nodes are generated in the course of the LDPC encoding process.

The *idx* field indicates whether the related bytes in the *udata* field have already been successfully decoded or not. The value of *idx[i]* is set to 0 if *udata[i]* has not yet been restored. Otherwise, its value is set to 1 and *udata[i]* contains the value of the restored message byte.

Finally, the current implementation of the LT decoding procedure is discussed. Firstly, it is obvious that the elements on the first list can be decoded. The reason for this is that, in the case of these elements, there is only one message byte in the association, and consequently the value of the message byte is trivially known since it was calculated as $Y = X_{s1}$ during the LT encoding process. In the case of the second list, two message bytes were associated with each other. The previously decoded bytes on the first and the second list can be used so as to facilitate the decoding process. Due to the properties of the XOR operation, the decoding process can be successfully performed if the Y encoded byte is known and there is only one byte in the association whose value is unknown. The reason for this is that, in the case of the second list, the encoded byte was calculated as $Y = X_{s1} \oplus X_{s2}$ during the LT encoding process. The Y encoded byte is known. If the first byte in the association is not known, however, the second byte is known, the first byte can be decoded as $X_{s1} = Y \oplus X_{s2}$. The second byte can be decoded similarly if the first byte is known. The decoding process happens in a similar fashion in the case of the third and the additional lists. In order to facilitate the decoding procedure, the previously decoded bytes on the current and the previous lists can be used. The LT decoding process is represented in Figure 3.18. Firstly, the lists are iterated in the order previously described. Finally, it is checked whether at least one new message byte has been successfully decoded or not. If there was no new message byte recovered successfully, the decoding process ends since it is certain that no new message bytes can be restored. Otherwise, the entire process is restarted and the lists are iterated once again. This step is required as such a message byte might be decoded on a given list which could be used so as to decode previous encoded bytes on the same list or additional encoded bytes on the previous lists. In the case of the low-level lists, there are more and more message bytes in the association, and consequently the basic principle is to attempt to decode as many encoded bytes on the high-level lists as possible so as to minimalize the number of undecoded bytes on the low-level lists. In the course of the LT decoding procedure, it is calculated how many message bytes have been successfully restored and finally, this value is returned by the LT decoding function.

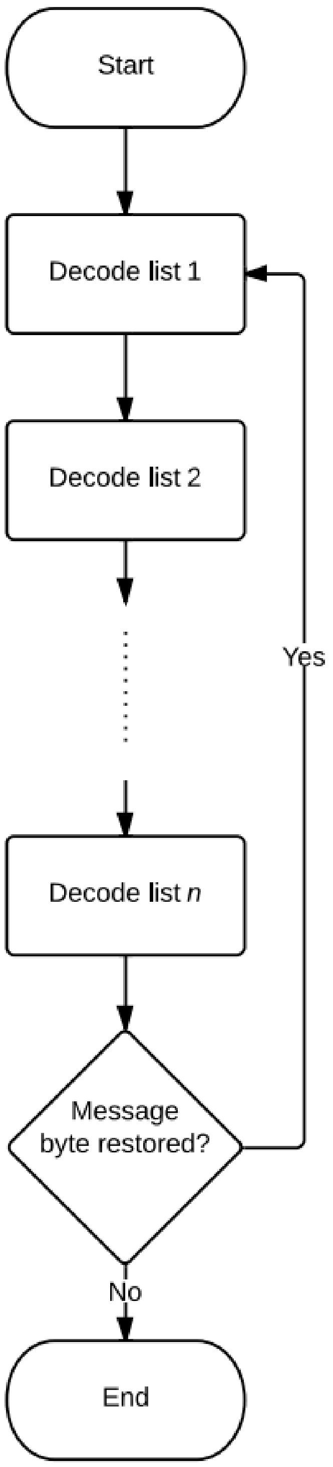


Figure 3.18. The LT decoding process

3.10.3 The LDPC decoding process

As illustrated in Figure 3.15, the first three fields (*tnodes*, *failed*, *needed*) of the *recv_ queue* are used in the course of the LDPC decoding. The function that implements the LDPC decoding always initializes these three fields whose values have local significance and are only used during the current iteration. The *tnodes[i]* field indicates the best known value of the i^{th} check node at the time. The value of *failed[i]* is set to 1 if the i^{th} check node cannot be restored. Otherwise, it is set to 0. The *needed[i]* field contains the index of the message byte which is required so that the given check node can be recovered. The LDPC decoding process can be divided into three steps.

The first step is the initialization. During this step, the value of each *tnodes[i]* and *failed[i]* is set to 0. Moreover, the initial value of each *needed[i]* is set to 65536 as this value cannot be interpreted as a valid index, and therefore it can be applied to indicate that the field is currently not used. Afterwards, the state of the random number generator is set to the state that was previously used by the sender in the case of the given block during the LDPC encoding process. The appropriate state is known since the two endpoints have exchanged the initial state of their random number generators in the course of the connection establishment and the initial state can be used to generate the particular state.

During the second step, the fields, which were initialized in the previous step, are set to their currently best known values. This step is exactly the same as it was depicted in Figure 3.9 in the case of the LDPC encoding procedure except the step “Set check node”. The reason for this is that, during the LDPC encoding, the value that is necessary to produce the value of the check node is known as it is a certain byte of the message sent by the user. However, in the case of the LDPC decoding process, it is possible that the given byte of the user message is unknown since it might not have been restored during the LT decoding procedure. The altered “Set check node” step is presented in Figure 3.19. If the value of the given byte of the user message has been successfully restored, and consequently its value is known, the particular message byte is XORed to the value of *tnodes[i]* as this message byte was one of the bytes in the association when the given check node was being generated. However, if the given byte of the user message has not been recovered, and therefore its value is unknown, two separate cases are possible. On the one hand, if all the message bytes in the association have been known so far in the case of the given check node (this check is denoted by “Failure” in the figure), the particular check node might be able to be used during the decoding process since the given check node itself may have been restored by the LT decoding process, and consequently its value might be known. As a result, *needed[i]* is set to the index of the only unknown message byte. On the other hand, if at least one message byte in the association is unknown, it is certain that the given check node cannot be used during the decoding process as there is more than one unknown byte in the association. In this case, *failed[i]* is set to 1 in order to indicate that the given check node cannot be used in the course of the third step.

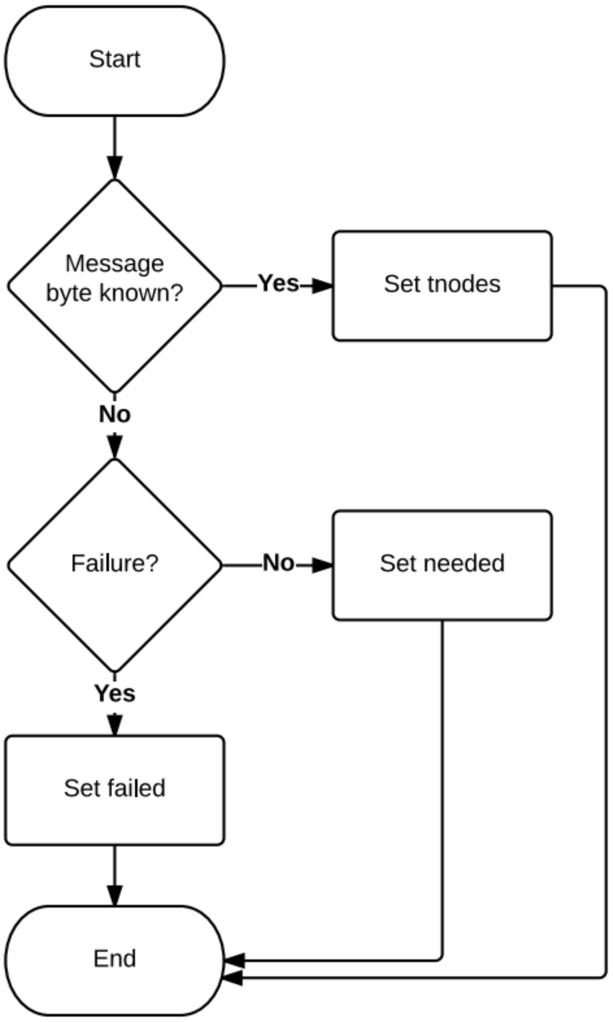


Figure 3.19. The modified “Set check node” step

During the third step, the actual decoding process is performed based on the values set in the course of the second step. The process is shown in Figure 3.20. The decoding steps are discussed for a given check node. The particular check node can only be decoded if *failed[i]* is not set for the given check node. Otherwise, it cannot be used during the LDPC decoding process. In the case of a given check node, if it can be restored, two cases are possible depending on whether the value of the check node is known or not. In the first case, the value of the check node is known, however, one of the message bytes in the association is not known. In this case, the given check node and *tnodes[i]* are XORed with each other in order to calculate the value of the unknown message byte. In the second case, the check node itself is not known, however, all the message bytes in the association are known. In this case, the value of the check node can be restored and its exact value is stored in the *tnodes[i]* variable.

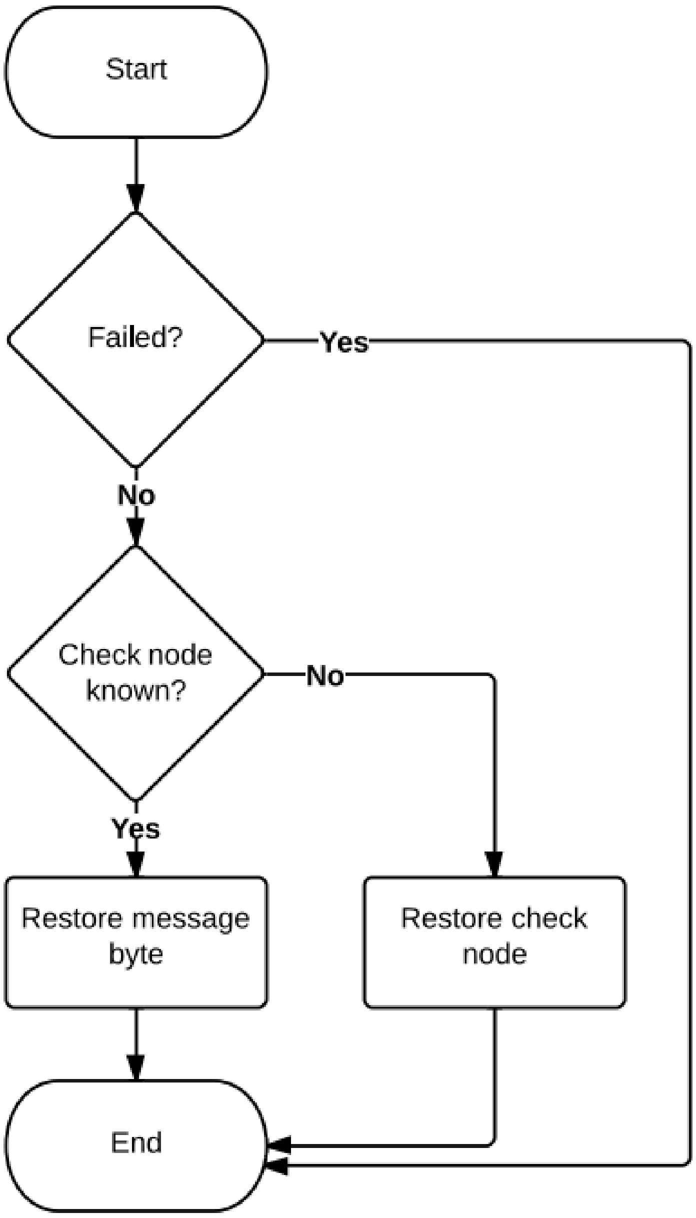


Figure 3.20. The third step in the case of a given check node

In the course of the third step of the LDPC decoding process, for each check node, the previously described steps are performed. Similarly to the LT decoding process, in the course of the LDPC decoding, it is calculated how many message bytes are restored successfully and finally, this value is returned by the LDPC decoding function.

3.11 DFCP connection termination

In this section, the connection termination procedure of DFCP will be discussed, which is very similar to the technique applied by TCP and its reliability is guaranteed by employing timers. The connection termination process of DFCP can be split into three phases.

The first phase of the connection termination is presented in Figure 3.21. During this phase, the connection termination initiator sends a *FIN segment* to the other endpoint.

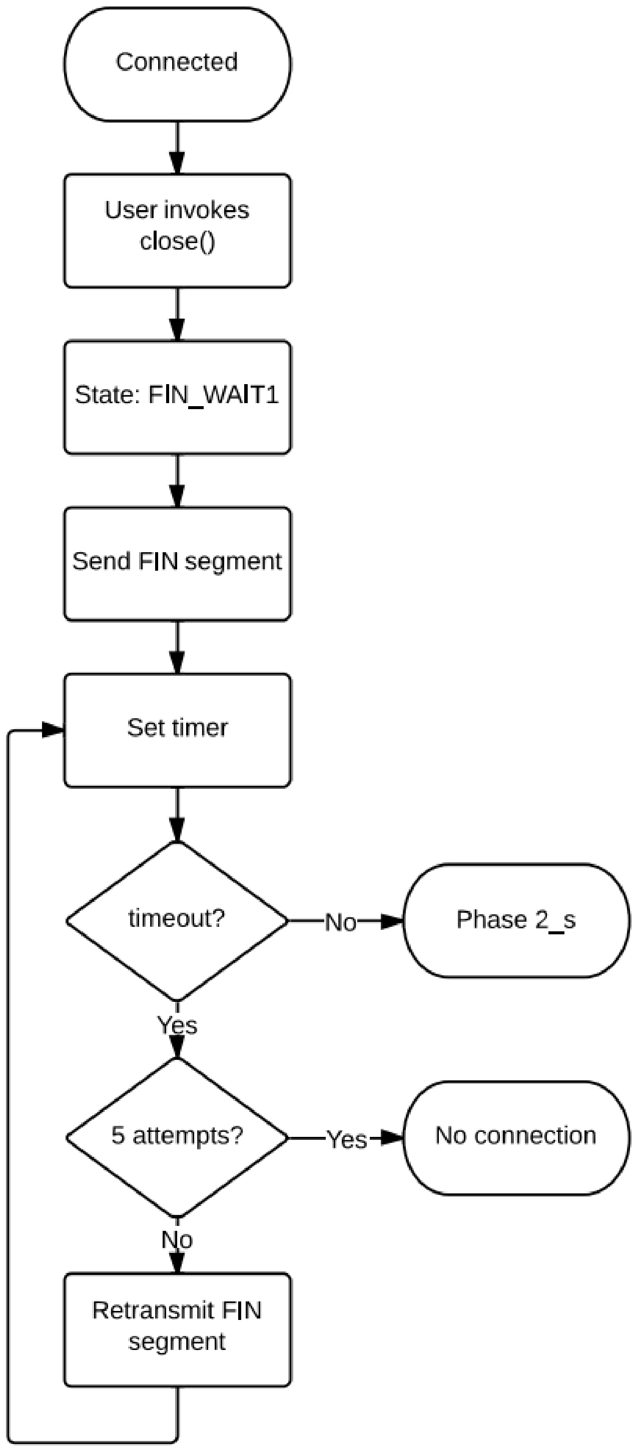


Figure 3.21. The first step of the connection termination

In the course of the first phase, the state of the initiator is set to *FIN_ WAIT1*. If the *FIN segment* is lost, it will be retransmitted by using a timer. By default, the maximum number of retransmissions is set to five. In Figure 3.21, it is assumed that if the timer is not expired, the *FIN segment* has successfully been received by the other side, and consequently the second phase of the process can be entered. After five unsuccessful retransmission attempts, the connection termination process is aborted and the resources are released at the initiator.

The second phase of the connection termination is depicted in Figure 3.22. During this phase, the endpoint that received the *FIN segment* processes the segment and sends an *ACK segment* in response to the *FIN segment*. In the case of the *ACK segment*, it is not necessary to use timers in order to retransmit the lost segments because if the segment is lost, the other side will resend the *FIN segment* due to the expiry of the FIN timer. Before having sent the *ACK segment*, the endpoint sets its state to *CLOSE_ WAIT*. The other side, which received the *ACK segment*, processes the segment and sets its state to *FIN_ WAIT2*. If the connection is not immediately closed by the other endpoint, it does not send a *FIN(ACK) segment*, however, it still acknowledges the *FIN segment* as presented in Figure 3.22.

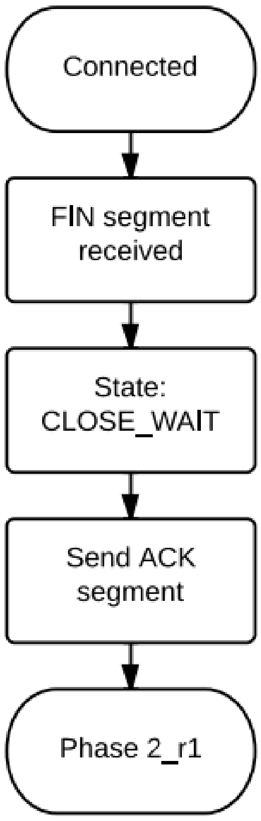


Figure 3.22. The step 2.a. of the connection termination

When the connection has also been closed by the other endpoint, it sends a *FINACK segment* to the other side and sets its state to *LAST_ACK*. Similarly to the *FIN segment*, if the *FINACK segment* is lost, it will be retransmitted by using a timer. The procedure is illustrated in Figure 3.23.

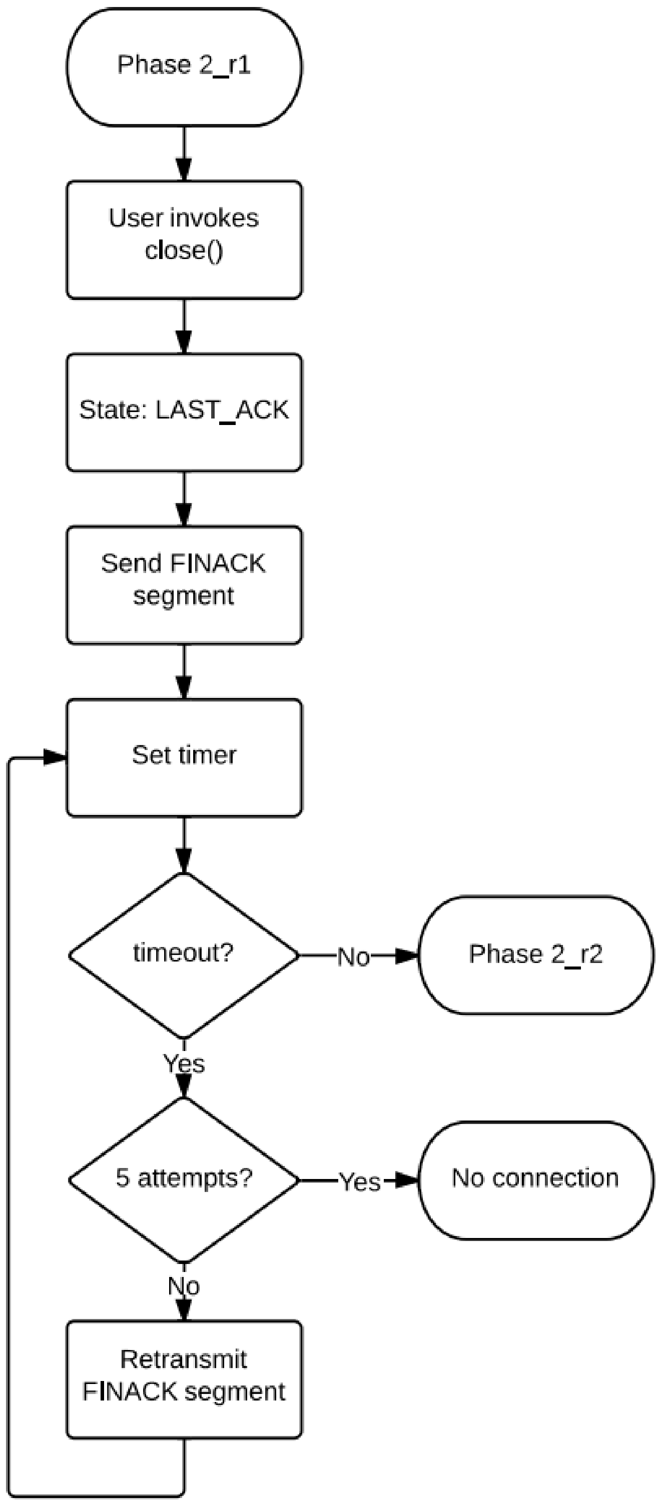


Figure 3.23. The step 2.b. of the connection termination

The third phase of the connection termination is represented in Figure 3.24. During this phase, the endpoint that received the *FINACK segment* sends an *ACK segment* in response to the *FINACK segment* without using timers and sets its state to *TIME_ WAIT*. This step is required in order to close the connection properly since the *ACK segment* might be lost. In this case, the other side gets stuck in *LAST_ ACK* state. However, as it resends the *FINACK segment*, it can be detected that the *ACK segment* was lost, and therefore it can be retransmitted.

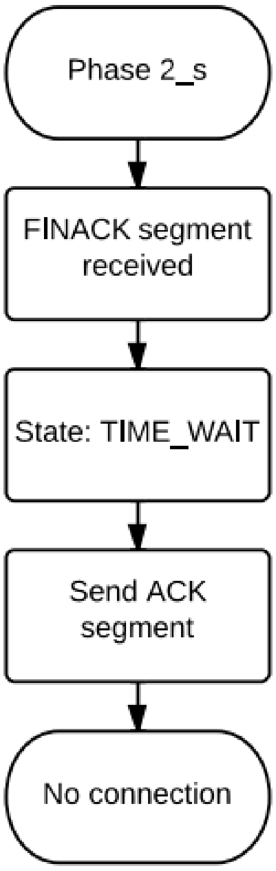


Figure 3.24. The third step of the connection termination

After waiting for a given amount of time in *TIME_ WAIT* state, the resources are released. The other endpoint that received the *ACK segment* processes the segment and sets its state to *CLOSE*. Finally, its resources are also released.

3.12 DFCP parameters

In this section, the parameters of DFCP are introduced. The value of these parameters can be set by the user application, and consequently it is able to alter the behavior of the protocol. If a particular parameter has been set to a given value, the kernel sets the related kernel variable to that value, and thereafter the protocol will operate according to the new value of the parameter. The scope of most parameters is limited to the connection for which the application has set the given parameters. As a result, the behavior of the protocol in the case of the new and already existing connections will be unaffected. In the case of the current implementation, there are a total of 13 parameters. Some of these parameters can be set whereas others can be queried by the user application. In addition, there are certain parameters on which both operations can be performed. The meaning and significance of the DFCP parameters are briefly presented in Table 3.2. The parameters can be divided into two separate categories. The first category consists of the parameters which can be set by the application. These parameters are used to alter the behavior of the protocol in the case of the given connection. The second category contains the parameters that can be queried by the user application. These parameters are used for testing and diagnostic purposes. Moreover, the significance of the parameters on which both operations can be performed is discussed in detail in a later subsection about the rate-limiting capability of the protocol.

Table 3.2. DFCP parameters

Name	**Function**	**Type**
Window size	It specifies the maximum size of the sliding window.	Set
Redundancy	It indicates the threshold used during the decoding procedure.	Set
Coding	It is used to enable (or disable) the offline encoding feature.	Set
Decoding	It is used to turn off (or on) the decoding procedure.	Set
Maxtokens	It sets the total number of tokens the kernel can be accumulate.	Both
Numtokens	It specifies the number of tokens available at the moment.	Both
Inctokens	It specifies the token increment per clock tick.	Both
Inttokens	It specifies the granularity of the rate limiter.	Both
Outgoing segments	It contains the total number of segments sent by the kernel.	Get
Incoming segments	It contains the total number of segments received by the kernel.	Get
Dropped segments	It contains the total number of segments dropped by the kernel.	Get
ACKs sent	It contains the total number of acknowledgements sent.	Get
ACKs received	It contains the total number of acknowledgements received.	Get

3.12.1 The primary parameters of DFCP

In this subsection, the first four parameters, shown in Table 3.2, are discussed. The *window size* parameter is used to set the maximum allowed size of the sliding window. This parameter also determines the number of *send_ queues* and *recv_ queues* and the amount of memory allocated for the given connection. The *redundancy* parameter is used to control the behavior of the receiver. After having received a certain number of encoded bytes, indicated by the *redundancy* parameter, for a given block, the receiver performs the decoding process on the block. The *coding* parameter is used to turn on or to turn off the encoding process in the case of the given connection. When the encoding procedure is turned off, the protocol applies the *offline encoding* capability, which is a slight modification of the original data sending procedure. In this case, the kernel only applies the actual data encoding process on the first data block. After having finished encoding the user message, the encoded bytes and the related variables (the states of the random number generator) used to build the headers of the DFCP segments are stored in a data structure (*tmp_ queue*) allocated for this particular purpose. When sending the additional data blocks, the kernel uses up the pre-encoded bytes and variables, and consequently it is not required to perform the actual encoding process again, which saves the time spent on encoding each data block. Henceforth, this data encoding method will be referred to as *offline encoding*. The steps of the process are presented in Figure 3.25.

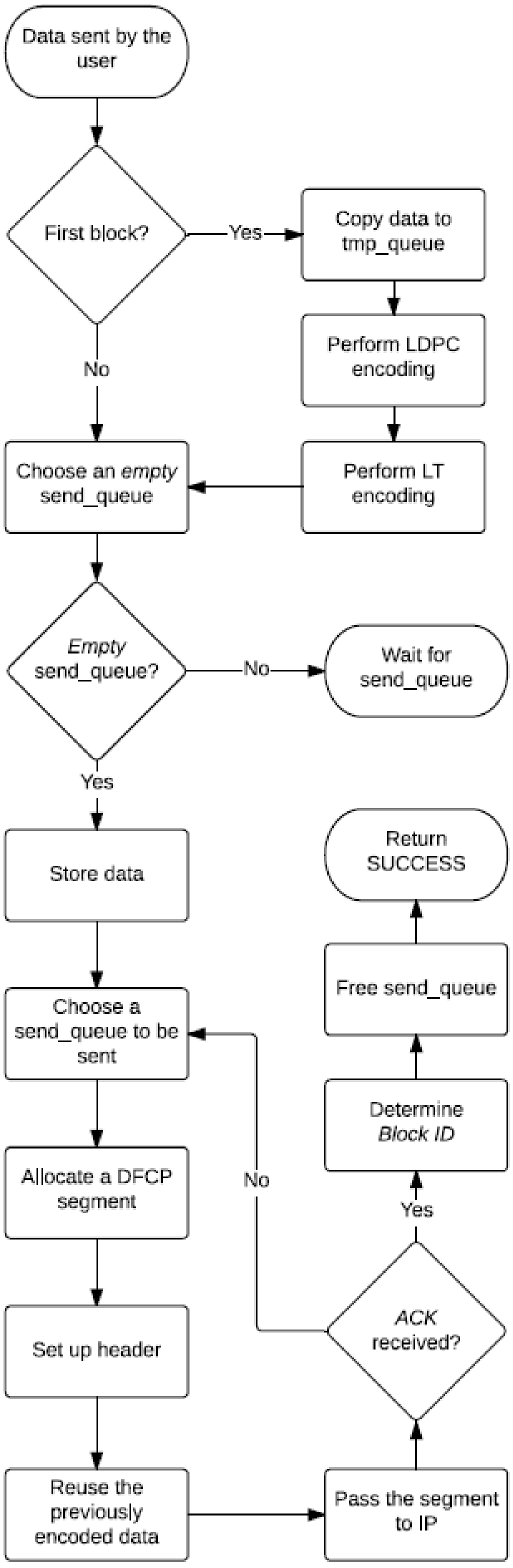


Figure 3.25. The *offline encoding* process

When data is sent by the user application, firstly, the kernel checks whether it is the first data block sent by the application or not. If it is the first block, the user message is copied to the *tmp_ queue*, which is a specific data structure in order to store the uncoded user message itself, the generated check nodes, the variables used to build the headers of the DFCP segments and the LT encoded bytes. Afterwards, the kernel performs the regular LDPC and LT encoding processes on the first block. In the course of the encoding processes, the states of the random number generator, the generated check nodes and the encoded bytes are stored in the appropriate fields of the *tmp_ queue*. Subsequently, similarly to the original data sending procedure, an empty *send_ queue* is chosen. If there is no empty *send_ queue* available at the time, the user application has to wait. The waiting ends when the kernel has received an acknowledgement for a block which is currently stored in a *send_ queue*, and consequently the given *send_ queue* can be freed. At this point, however, unlike the original data sending procedure, the user message is not copied to the chosen *send_ queue*. The *send_ queue* is solely used to store the block-related information so as to facilitate the operation of the sliding window mechanism. After storing the information in the given *send_ queue*, a block is chosen to be sent according to the round-robin algorithm without performing the regular LDPC encoding process. Afterwards, a DFCP segment is allocated. The DFCP header and data fields are built by the kernel based on the stored encoded bytes and additional information without executing the actual LT encoding procedure. Finally, the DFCP segment is passed to the IP layer for further processing. The additional segments are generated in the same way until the kernel receives an acknowledgement for a given block. When an acknowledgement has been received, the kernel determines the *Block ID* of the acknowledged block and frees the associated *send_ queue*. Furthermore, the sending application will be notified of the successful transmission of the block by the kernel setting the return value of the *write()* function correctly. When additional data blocks are sent by the application, the kernel does not copy the user message to the *tmp_ queue* again but it reuses the stored variables and the encoded bytes. Since the time-consuming data encoding processes are not performed, the altered data sending process is very fast. As a result, in the course of the network measurements, which are discussed in a later section, the significant characteristics of the new future Internet architecture can be focused on independently of the actual implementation of the applied coding scheme.

Finally, the *decoding* parameter is used to turn on or to turn off the decoding process entirely. When the decoding procedure is turned off, the receiver does not perform the decoding process on the received blocks but assumes that the blocks can be decoded after having received *redundancy* number of encoded bytes for each block. To be more precise, the data receiving process, depicted in Figure 3.14, is modified so that after having determined the *Block ID* and performed the sanity checks, the segment would immediately be dropped instead of storing its contents. The *recv_ queue* associated with the block is only used to count the number of encoded bytes received for the given block. When the required number of encoded bytes has been received, the receiving application is notified by the kernel. Afterwards, the kernel sends an acknowledgement for the given block and the *recv_ queue* is freed.

3.12.2 The rate-limiting capability of DFCP

In this subsection, the second four parameters, shown in Table 3.2, and the rate-limiting feature of the protocol are introduced. The second four parameters are used in order to control the data sending rate at the sender. As presented in Table 3.2, the value of these parameters can be both set and queried and their scope is limited to a certain connection. The rate-limiting enhancement enables the protocol to send data at a specified rate instead of sending at maximum rate.

In the case of the current implementation, a token bucket mechanism is applied in order to control the data sending rate. The basic operation of the mechanism is depicted in Figure 3.26.

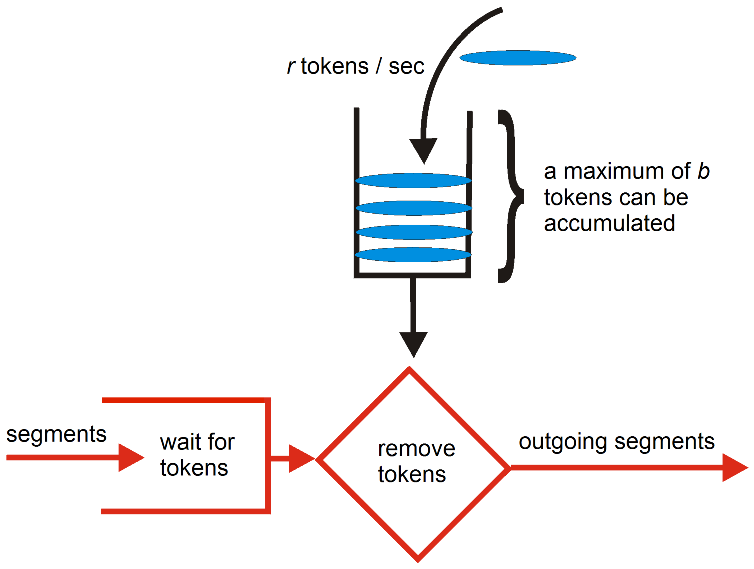


Figure 3.26. The operation of the token bucket algorithm

The algorithm maintains a fixed-capacity bucket into which tokens are added at a fixed rate, where each token represents an encoded byte. Before sending a segment, it is checked whether the bucket contains the sufficient number of tokens at the time. If at least 1420 tokens, which is equivalent to the number of encoded bytes in a DFCP segment, are available, the appropriate number of tokens is removed from the bucket. Afterwards, the segment is passed to the IP layer for further processing. However, if there are insufficient tokens in the bucket, the contents of the bucket remain unchanged. In this case, the kernel waits until the necessary number of tokens is available. More precisely, the user application is able to specify the data sending rate by invoking the appropriate system call. In other words, it is determined how many tokens need to be generated in a unit of time in order to achieve the requested data sending rate. The kernel uses timers so as to implement this task. The maximum size of the bucket can also be specified by the application, which determines the degree of burstiness.

The user application is responsible for setting the four parameters of the rate limiter properly. After having set the necessary parameters, the tokens are generated at the given rate by the kernel, and therefore the segments are sent at the specified rate. As regards the meaning of the required parameters, the variable *maxtokens* refers to the maximum size of the bucket. If the number of tokens in the bucket has reached the value of *maxtokens*, no additional tokens are generated. A segment can only be sent if there are at least 1420 tokens in the bucket. The segments are sent at maximum rate until there are enough tokens available. If there are less than 1420 tokens in the bucket, the kernel waits until the necessary number of tokens is available. The parameter *inctokens* determines the number of tokens generated in a unit of time. In the kernel, a unit of time refers to the frequency at which a certain timer interrupt is invoked. In the case of the current implementation, this interrupt is called 250 times in a second, and therefore the number of tokens is increased once in 4 milliseconds. The parameter *inctokens* can be calculated as shown in Formula (3.3):

$$inctokens = \left\lfloor \frac{bandwidth \cdot 1024 \cdot 1024}{8 \cdot 250} \right\rfloor \tag{3.3}$$

Here, *bandwidth* refers to the requested data sending rate specified by the application in Mbps. The resulting integer number is used by the Linux kernel in order to increase the number of tokens periodically. The parameter *inttokens* determines the frequency at which the timer interrupt is invoked. By default, the value of this parameter is set to 1, which provides the best granularity and the most accurate operation. In this case, the number of tokens is increased once in 4 milliseconds. However, for instance, if the value of this parameter were set to 10, the number of tokens would be increased once in 40 milliseconds. Finally, the parameter *numtokens* is used by the user application so as to manually set the number of tokens in the bucket.

3.12.3 The testing and diagnostic parameters of DFCP

In this subsection, the last five parameters, shown in Table 3.2, are presented, which are used for testing and diagnostic purposes. These parameters are global to each connection. The parameter *outgoing segments* counts and stores the total number of DFCP segments sent while the parameter *incoming segments* counts and contains the total number of DFCP segments received by the kernel. The variable *dropped segments* counts the total number of DFCP segments dropped by the kernel. Finally, the parameter *ACKs sent* counts and stores the total number of acknowledgements sent whereas the parameter *ACKs received* counts and contains the total number of acknowledgements received by the kernel.

To sum up, the behavior and operation of the protocol can be investigated and the functions of the protocol (such as the encoding, decoding and rate-limiting capability) can be temporarily (or permanently) disabled (or enabled) by setting the values of certain parameters. Consequently, applying parameters facilitates testing of the protocol. Furthermore, the behavior of the protocol on complex network topologies can be more easily analyzed.

Chapter 4

Performance Evaluation of DFCP

This part focuses on the measurements carried out in order to validate the proper operation of DFCP, analyze the behavior of the protocol and compare the performance of DFCP to widely applied TCP variants. The chapter is organized in the following way. It consists of three sections. In the first section, the network topologies and the traffic scenarios are presented. In addition, the software tools used for the measurements and the related hardware equipment are introduced and their configurations are discussed. In the second section, the operation of DFCP is investigated and validated. Finally, in the third section, a comprehensive performance analysis is conducted on multiple platforms so as to analyze the performance of DFCP. The measurement results are provided, evaluated and compared to the performance of significant TCP variants. In addition, the necessary conclusions are drawn.

In order to examine and validate the operation of DFCP, simulations and testbed measurements have been carried out [36]. The main risk of solely relying on simulation results is the fact that simulation measurements might be unrealistic in several cases. As a result, significant real-network factors can be easily neglected [37]. However, only performing testbed measurements can lead to the loss of generality since the special hardware components of the host computers on which the measurements are conducted can hugely affect the results. In addition, building a network testbed is a time-consuming process. Since DFCP is based on a new approach, it is crucial to ensure that the measurement results are reliable and all the conclusions drawn are valid. In order to fit these requirements, a validation analysis has been carried out on various platforms including a laboratory testbed built in the department, the Emulab network environment [38] and the ns-2 network simulator [39].

4.1 Network topologies and traffic scenarios

In this section, the network topologies and the traffic scenarios are presented. Furthermore, the software tools used for the measurements are introduced and their configurations are briefly described.

The behavior and performance of DFCP have been evaluated on different network topologies including a simple *dumbbell topology* and a more complex *parking lot topol-*

ogy. The dumbbell topology consisted of N sender-receiver pairs as depicted in Figure 4.1.

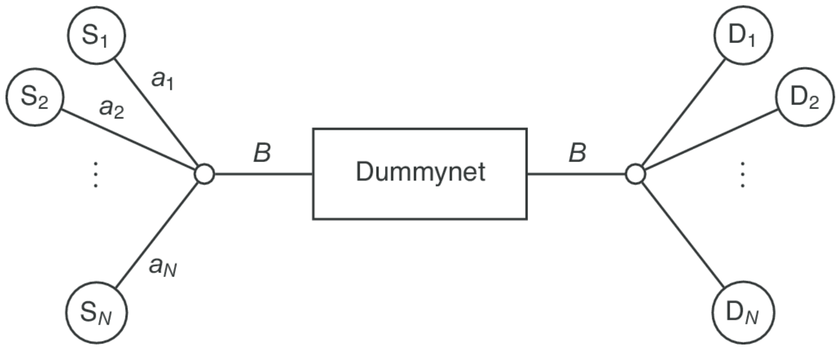


Figure 4.1. Dumbbell topology with N sender-receiver pairs

Firstly, experiments have been conducted with a single flow (one sender-receiver pair) to demonstrate the ability of DFCP to resist variable delay and packet loss characteristics in the network. In this case, the capacity of the bottleneck link (C_B) was set to 1 Gbps. Furthermore, the fairness properties of DFCP have been examined using two source and destination nodes (two sender-receiver pairs). The basic purpose of these measurements was to analyze the behavior of DFCP under the condition when two concurrent flows compete for the available bandwidth of the bottleneck link. In this scenario, the capacities of both access links (denoted by a_1 and a_2) and the bottleneck link (denoted by B) were set to 1 Gbps. As regards scalability, the performance and fairness properties of DFCP have been investigated when increasing the number of flows ($N = 10, 20, ..., 100$) and in the case of specifying various bottleneck link capacities ($C_B = 0.1, 1, 10$ Gbps).

The scenarios described above have made it possible to explore the fundamental features of DFCP and examine its scalability. In addition, the performance and behavior of DFCP have been analyzed in a more realistic environment. The parking lot topology, which was built for this series of experiments, is depicted in Figure 4.2.

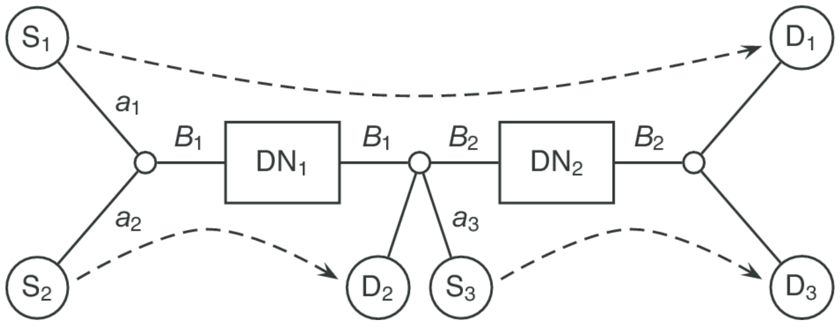


Figure 4.2. Parking lot topology with three sender-receiver pairs

As it can be seen, the topology included three sender-receiver pairs and two bottleneck links. In a real network, multiple bottlenecks are common, and therefore it is essential to evaluate how a transport protocol would perform under such conditions. In the case of these experiments, the capacity of each access link (a_1, a_2 and a_3) was set to 1 Gbps and the capacities of the bottleneck links (C_{B_1} and C_{B_2}) were set to various values as discussed in the following section. Except if otherwise stated, the measurements lasted 60 seconds and the results were obtained by excluding the first 15 seconds from the length of each measurement in order to ignore the impact of the transient behavior of the investigated transport protocols.

In the case of the measurements with multiple flows, the individual flows were started at the same time and the WFQ (Weighted Fair Queueing) scheduling algorithm was applied with equal weights by default. However, additional experiments have also been carried out with other fair schedulers, such as the SFQ (Stochastic Fair Queueing) and DRR (Deficit Round Robin) algorithms. In addition, some measurements have been conducted using the DropTail queue management scheme as well, which is the simplest queue management mechanism available in today's network routers.

In order to evaluate the performance of DFCP and validate its operation, the test scenarios were executed on three different platforms independently, which were a laboratory testbed built in the department, the Emulab network environment and the ns-2 network simulator. The *laboratory testbed* consisted of senders, receivers and the Dummynet network emulator [40], which was used to simulate various network parameters, such as buffer size, bandwidth, delay and packet loss rate. Each host computer was equipped with the same hardware components [36]. The second platform, *Emulab*, is a network testbed providing researchers with a wide range of environments in which to develop, debug and evaluate their systems [38]. For each test scenario, the measurement setup was identical to the one used in the laboratory testbed. According to the notational system of Emulab, the type of the sender and receiver nodes was *pc3000* while *d710* nodes were applied as network emulators. Similarly to the testbed measurements, the modified Linux kernel including the current implementation of DFCP has been booted on all the test computers. The third tool, the *ns-2 network simulator*, was used to validate the operation of DFCP. Since the first prototype of DFCP has been implemented in the Linux kernel, it was required to find a solution to integrate the new protocol into the ns-2 simulation environment. In fact, there are several tools available for this purpose, but very few of them are able to provide reasonable accuracy, efficiency and support for a wide range of operating systems and kernel versions. In order to meet all the necessary requirements, the NSC (Network Simulation Cradle) [41] has been chosen, which executes the code of real-world operating systems' network stacks in a wrapper that allows the protocols to be used in the ns-2 network simulator. This provides real-world code in a simulation context allowing accurate simulation at little extra cost. NSC supports the simulation of a number of network stacks, such as FreeBSD, OpenBSD, lwIP and Linux. NSC has been validated by comparing situations using a test network with the same situations in the simulator and it has been shown that NSC is able to produce extremely accurate results. Moreover, NSC has been ported to several network simulators including both ns-2 and ns-3. Although NSC is an excellent tool to simulate different TCP implementations and new TCP-like transport protocols, it was a challenging job to integrate the current implementation of DFCP into the NSC environment since DFCP is based on a completely different paradigm than the principles of TCP.

4.2 Validation measurements

In this section, the validation analysis is documented which has been carried out to examine the behavior of the protocol and validate its operation. The results of the validation measurements, which were performed on the three previously described platforms, are discussed in detail.

The performance of DFCP was measured in terms of *goodput* which is a well-known and widely-accepted performance metric in networking and it refers to the number of useful information bytes delivered per second to the application layer protocol. Henceforth, in the figures, the average goodput is calculated for the entire length of the measurements with the exception of the transient phase of the investigated protocols which is excluded from the calculations. In the next two figures, the basic characteristics of DFCP are illustrated showing its adaptability to changing network conditions, such as variable packet loss rate and round-trip time. These measurements have been carried out on the dumbbell topology with one sender-receiver pair as depicted in Figure 4.1. The impact of packet loss on the performance of DFCP is presented in Figure 4.3.

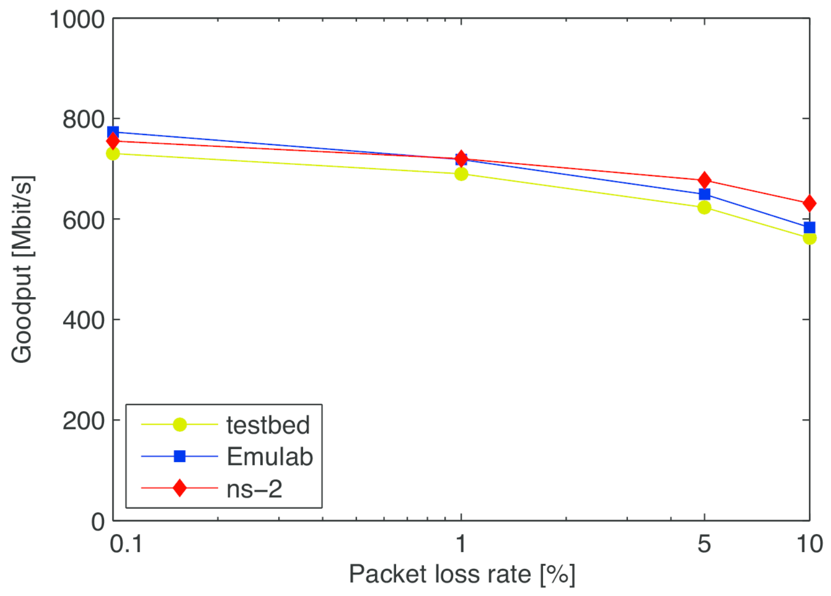


Figure 4.3. The impact of packet loss on the performance of DFCP

It is important to analyze this scenario since TCP is very sensitive to packet loss resulting in significant performance degradation in the case of increasing packet loss rate. The figure clearly shows that DFCP is even able to operate efficiently in environments with high packet loss rates by using *optimal redundancy*. More precisely, for a given packet loss rate, optimal redundancy refers to the minimum coding overhead that is required for the receiver to be able to decode the encoded data successfully. The impact of round-trip time on the performance of DFCP is illustrated in Figure 4.4.

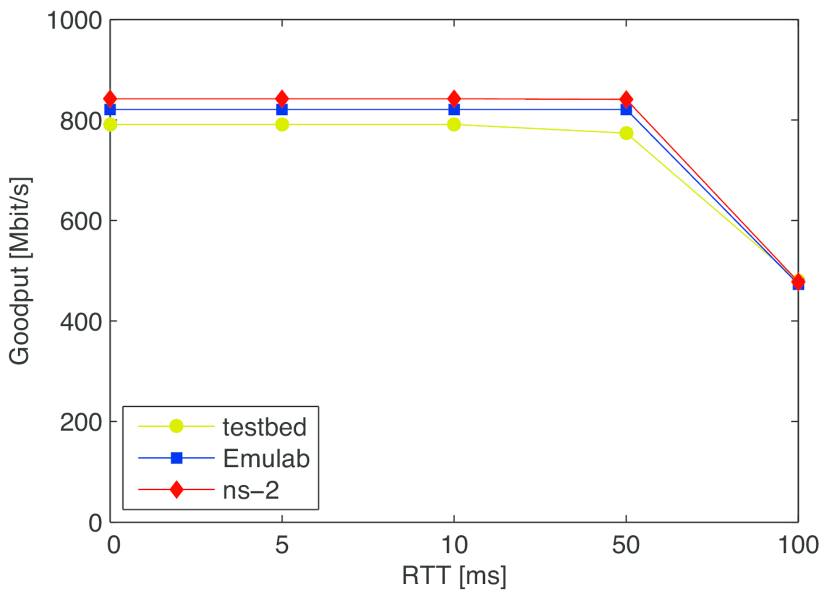


Figure 4.4. The impact of round-trip time on the performance of DFCP

It can be seen that DFCP achieves outstanding performance in high-delay networks since its goodput decreases slowly as the round-trip time increases. As shown in the previous two figures, the curves have very similar characteristics in the case of all the three platforms. As a result, it can be stated that these advantageous features of DFCP have been validated.

In Figure 4.5, the behavior of DFCP is illustrated when two competing flows with different round-trip times share the available bandwidth. This is a common situation in real networks often referred to as RTT-fairness problem [42].

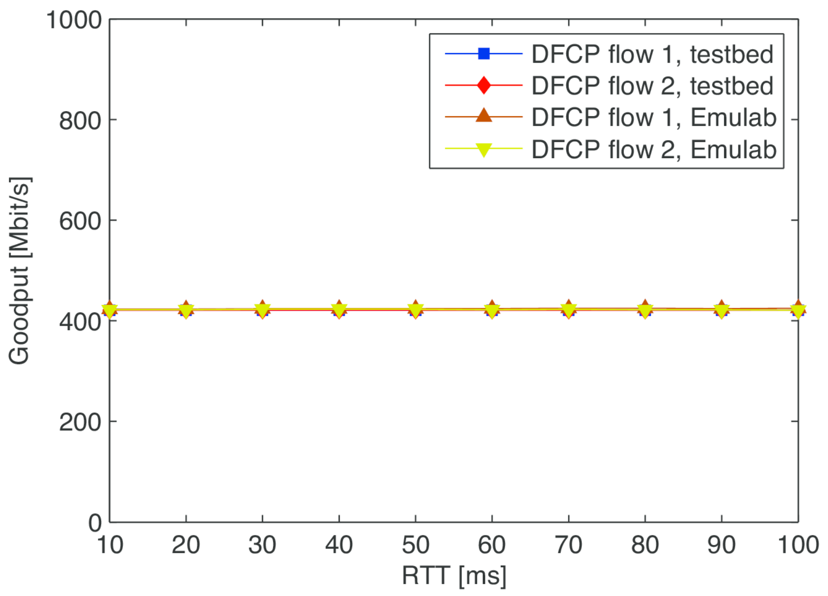


Figure 4.5. Bandwidth sharing between two competing DFCP flows with different RTTs

RTT-fairness is a critical issue since traditional TCP is unfair. More precisely, in the case of competing flows with different RTTs, the flow having smaller RTT is able to gain more bandwidth than the others since the flows having smaller RTTs grow their windows faster, which leads to unfair bandwidth allocation. In this particular scenario, the first flow had a fixed RTT of 10 ms whereas the RTT of the second flow was increased from 10 ms to 100 ms. As presented in the figure, DFCP is able to share the available bandwidth fairly on the different platforms since both flows gain an equal share of the bandwidth of the bottleneck link independently of their actual RTTs.

The following measurement, see Figure 4.6, was conducted on the parking lot topology illustrated in Figure 4.2.

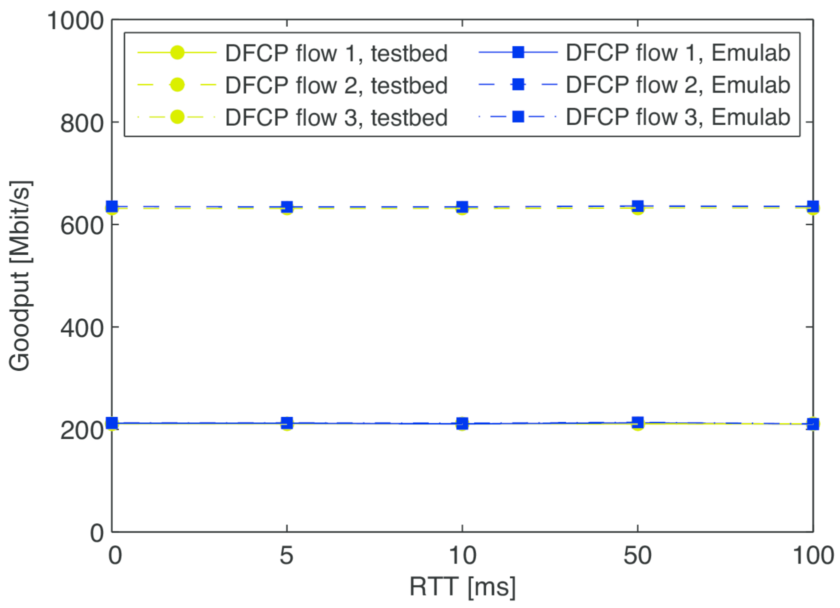


Figure 4.6. The performance of DFCP in a network with multiple bottleneck links

This particular scenario was designed in order to study the behavior of DFCP in a multi-bottleneck environment. The capacity of the first bottleneck link (B_1) was set to 1 Gbps whereas the second bottleneck link (B_2) had a capacity of 500 Mbps. The goodput values of the three DFCP flows are depicted in the figure as a function of the RTT of B_2 while B_1 had a fixed RTT of 10 ms. It can be observed that the first flow and the third flow gain an equal share of the bandwidth available on B_2. Since the rate of the first flow is limited by the capacity of B_2, the second flow gains more bandwidth than the first one utilizing the available bandwidth of B_1. As a result, each bottleneck link becomes fully utilized and is fairly shared by the DFCP flows.

4.3 Comparative analysis

In this section, a comprehensive performance analysis is presented, which was carried out on the three testing platforms by comparing the performance of DFCP to widely used TCP versions, which, in this particular case, are TCP CUBIC and TCP NewReno with SACK option enabled.

4.3.1 Loss and delay performance

One of the major beneficial properties of DFCP is illustrated in Figure 4.7 and Figure 4.8 which demonstrate that DFCP is much more resistant to packet loss than TCP CUBIC and TCP NewReno if the optimal redundancy is used.

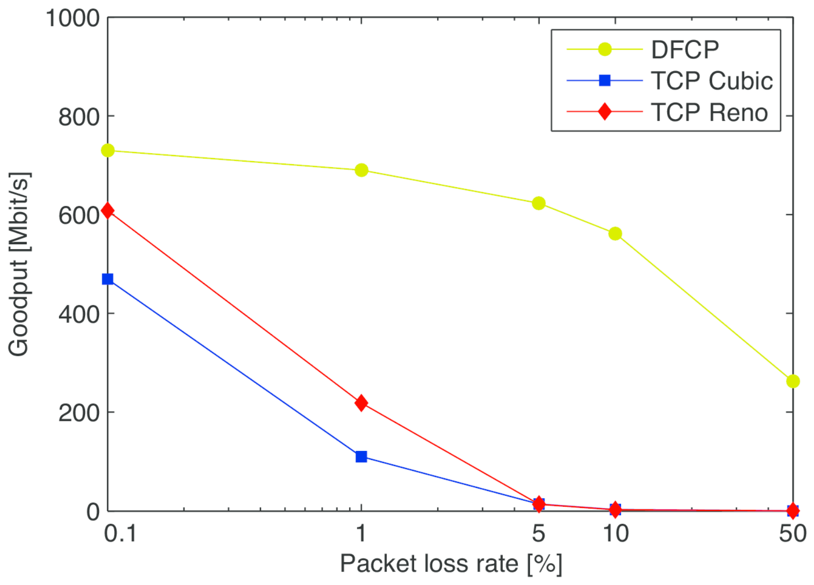


Figure 4.7. The performance of DFCP and the investigated TCP variants in a lossy environment (Testbed)

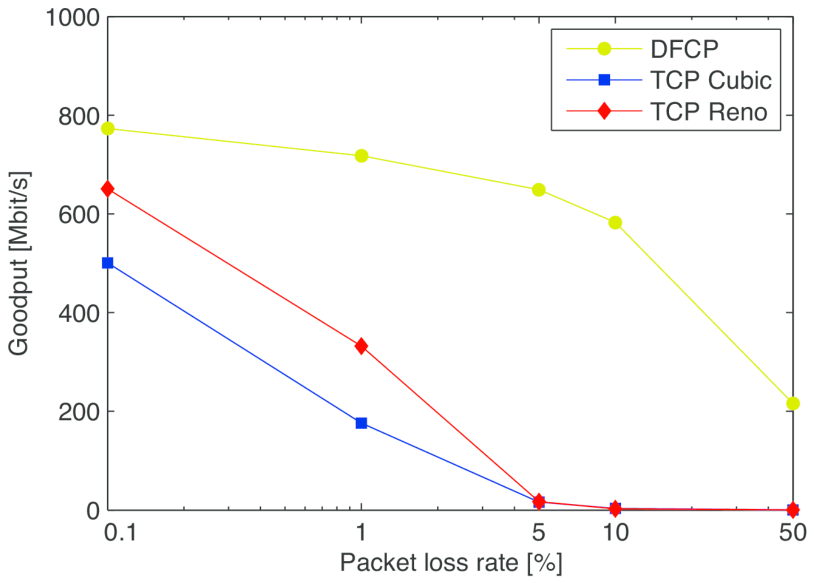


Figure 4.8. The performance of DFCP and the investigated TCP variants in a lossy environment (Emulab)

The goodput difference has already been noticeable at 0.1% packet loss rate, however, if the packet loss rate increases, DFCP highly outperforms both TCP variants. For instance, at 1% packet loss rate, the ratio of the goodput values achieved by DFCP and TCP NewReno is approximately 3. In addition, this ratio is 6 in the case of TCP CUBIC.

When the packet loss rate reaches the value of 10%, DFCP becomes more than 250 times faster than the investigated TCP versions and it still operates efficiently in the case of extremely high packet loss rates (50%). However, it can be observed that both TCP variants suffer from significant performance degradation even in the case of low packet loss rates. Furthermore, it is also noticeable that the performance characteristics of the investigated transport protocols seem to be very similar in the laboratory testbed and in the Emulab environment, which can be considered a validation of the results.

The results of the performance comparison of DFCP and the two TCP variants in the case of variable round-trip time are presented in Figure 4.9 and Figure 4.10.

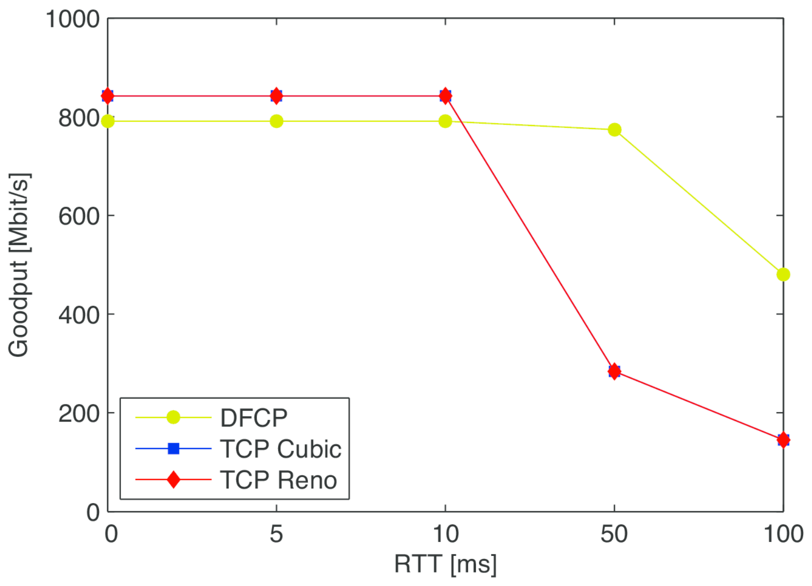


Figure 4.9. The performance of DFCP and the investigated TCP variants in a variable RTT environment (Testbed)

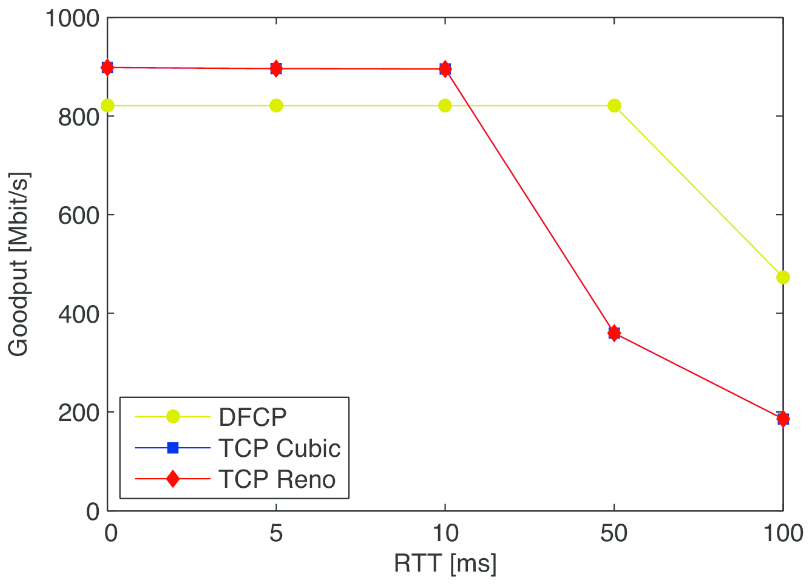


Figure 4.10. The performance of DFCP and the investigated TCP variants in a variable RTT environment (Emulab)

As illustrated in the figures, in the RTT interval 0 ms - 10 ms, the two TCP variants perform better than DFCP in terms of goodput, however, the difference is negligible and is caused by the coding overhead. Furthermore, for the RTT values greater than 10 ms, DFCP achieves significantly higher transfer rates than TCP CUBIC and TCP NewReno. Since the typical value of the round-trip time exceeds 10 ms in real networks [43], DFCP operates more efficiently under such conditions than TCP.

4.3.2 Buffer size requirement

It is a well-known fact that the buffer size requirement of TCP is at least of square-root order in the number of competing flows [5]. This requirement imposes a significant challenge on all-optical networks where only small buffers can be realized due to both economic and technological constraints [4], [6]. In Figure 4.11, it is demonstrated how the performance of DFCP and the two TCP variants is affected by the buffer size.

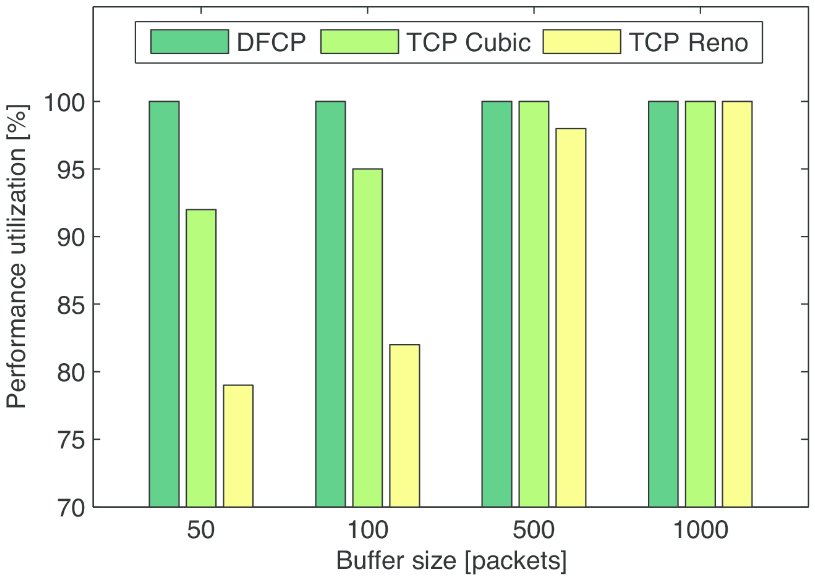


Figure 4.11. The impact of buffer size on the performance of DFCP and the two TCP variants

In this scenario, the round-trip time was set to 10 ms and no packet loss was simulated. The buffer size is given in packets and the vertical axis represents the *performance utilization* of the investigated transport protocols. The performance utilization is the ratio (specified in percentage) of the goodput that can be obtained with a particular buffer size and the maximum goodput that can be achieved when the buffer size is set to a sufficiently high value at which all the limiting factors can be neglected. It can be seen that, if the buffer size is set to 1000 packets, each protocol is able to achieve maximum performance utilization. However, if the buffer size decreases, the performance of the TCP variants drops significantly. For instance, if the buffer size is set to 50 packets, TCP CUBIC and TCP NewReno can only operate at reduced transfer rates which are 92% and 79% of the ideal cases, respectively. At the same time, DFCP is able to reach the maximum performance independently of the buffer size. This property of the transport mechanism of DFCP favors all-optical networking.

4.3.3 Bandwidth sharing using various buffer management disciplines

Another important aspect that need to be focused on and investigated is how a transport protocol shares the available bandwidth of a bottleneck link among the competing flows, which is often called the *fairness property*. In the experiments conducted, *Jain's fairness index* was used as the fairness metric, which is one of the most popular and widely accepted fairness indices in the literature [44]. Jain's index is computed as shown in Formula (4.1):

$$JI = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} x_i^2} \tag{4.1}$$

Here, x_i denotes the normalized throughput (or goodput) of flow i while n refers to the number of flows. The value of JI is between 0 and 1 and the higher the value, the better the fairness. It is widely known that the standard TCP flows cannot share the bandwidth of a bottleneck link equally in the case of competing flows with different round-trip times due to the properties of the AIMD mechanism [42].

The achieved goodput in the case of two competing DFCP and TCP CUBIC flows is presented in Figure 4.12 and Figure 4.13. The first flow had a fixed delay of 10 ms whereas the delay of the second flow was increased from 10 ms to 100 ms. Since the results of TCP NewReno were almost the same as in the case of TCP CUBIC, only the latter is shown in the figures.

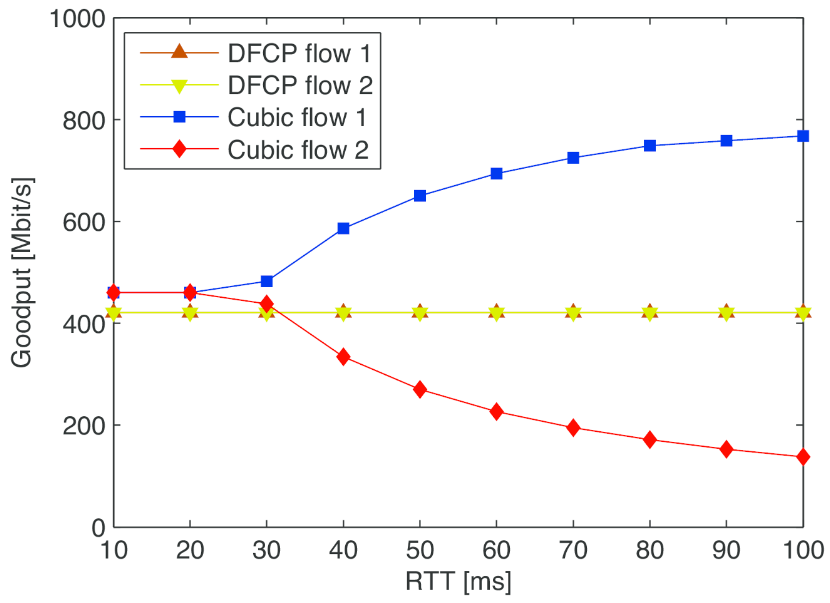


Figure 4.12. The performance of DFCP and TCP CUBIC in the case of two competing flows (Testbed)

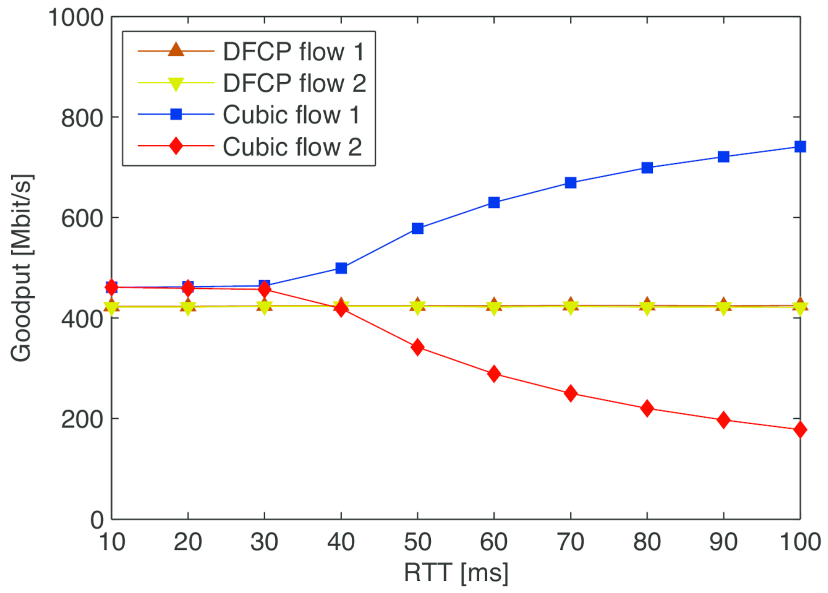


Figure 4.13. The performance of DFCP and TCP CUBIC in the case of two competing flows (Emulab)

It can be seen that, according to the testbed measurements (see Figure 4.12), the bottleneck link capacity is equally shared by the two TCP CUBIC flows for the RTT values less than 20 ms. However, for the RTT values greater than 20 ms, the goodput of the second flow starts to decrease, and consequently the flow with the smaller RTT can gain a greater portion of the available bandwidth of the bottleneck link indicating the unfair behavior of TCP. In contrast, the DFCP flows achieve perfect fairness as they share the available bandwidth equally and they are much less sensitive to the round-trip time compared to TCP. It is emphasized that the goodput difference between the DFCP and TCP flows for the RTT values less than 20 ms is due to the coding overhead.

Comparing the results obtained on the different platforms, it can be observed that the behavior of DFCP in the Emulab environment is the same as in the laboratory testbed whereas TCP CUBIC achieves slightly better fairness in the Emulab environment.

In our proposed future network architecture, the best solution to achieve fairness is to use *fair schedulers* as it was mentioned in Chapter 3. In fact, the data transmission mechanism of DFCP is not able to guarantee fairness on the host side. Therefore, the only solution is if this task is performed by the network routers. However, in this context, there are some open questions to be answered. On the one hand, a number of fair scheduling algorithms have been developed in the last two decades but few of them are available in today's routers. Consequently, it is a difficult issue to choose between them. On the other hand, in the case of most routers, the DropTail queue management policy is applied by default since it is the simplest algorithm, however, it is not eligible to provide fairness. It has to be analyzed how DFCP operates under such conditions. In order to find answers to these issues, the fairness analysis was extended to also investigate additional queueing mechanisms available in ns-2. The fairness index for different schedulers is illustrated in Figure 4.14.

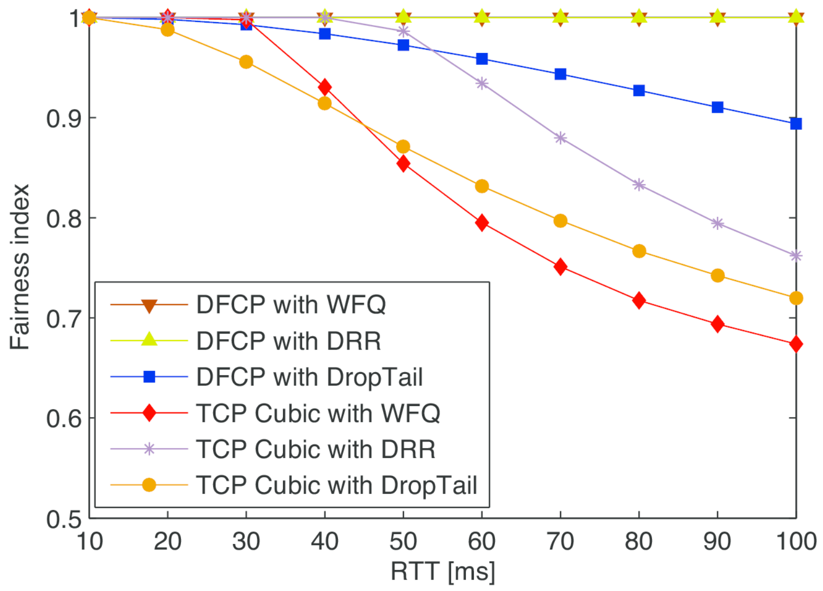


Figure 4.14. Intra-protocol fairness with WFQ, DRR and DropTail queueing

The results clearly show that, if fair schedulers are used, DFCP is able to guarantee perfect fairness in the case of two competing flows independently of their actual RTTs. Moreover, DFCP achieves better fairness than TCP even with the much simpler DropTail algorithm. To sum up, it can be observed that, in a more realistic environment with typical network parameters, DFCP is able to provide a higher degree of fairness compared to TCP for each queueing discipline.

4.3.4 Performance in a multi-bottleneck environment

The results of the performance comparison of DFCP and TCP CUBIC are presented in Figure 4.15 and Figure 4.16. These experiments were conducted on the parking lot topology (see Figure 4.2) with three concurrent flows started at the same time. In this measurement, the capacities of the bottleneck links, denoted by B_1 and B_2, were set to 1 Gbps. The bottleneck link B_1 had a fixed RTT of 10 ms while the RTT of B_2 was increased from 0 ms to 100 ms. Based on the figures, the following conclusions can be drawn. Until the round-trip time of B_2 reaches the value of 10 ms, both the DFCP and TCP CUBIC flows share the bandwidth of B_1 and B_2 in a fair way. However, for the higher RTT values, TCP CUBIC becomes unfair gradually due to the fact that TCP is sensitive to the round-trip time. As the goodput achieved by the first and the third flow decreases for increasing RTT (since they go through *B2*), the second flow with lower RTT gains more and more bandwidth. Consequently, TCP CUBIC does not provide fairness between the first and the second flow that have different RTTs. In addition, in this case, the available bandwidth of B_2 is also shared unequally, and therefore the first and the third flow achieve different goodput performance. As it was mentioned earlier, this behavior is highly undesirable and the results show that DFCP is able to solve this issue by providing perfect fairness for each flow independently of their actual RTTs owing to its robustness to changing network conditions.

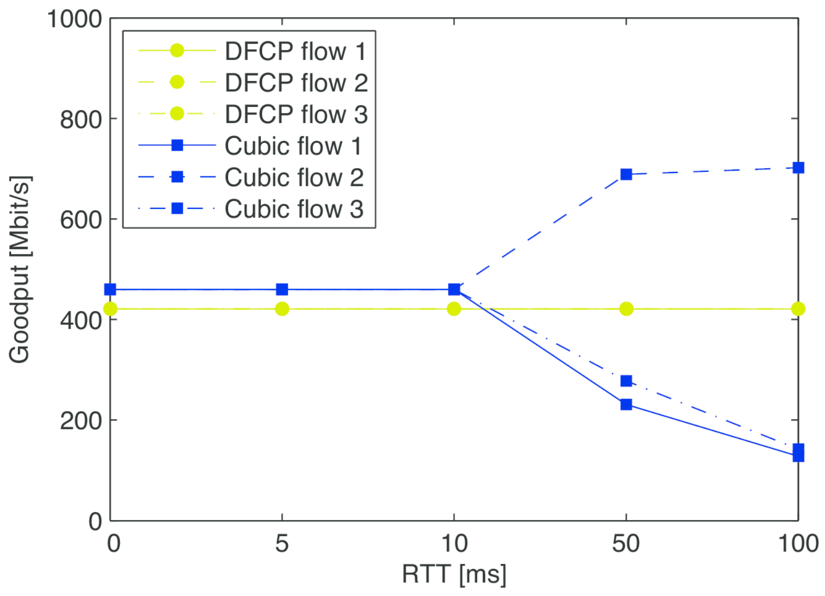


Figure 4.15. The behavior of DFCP and TCP CUBIC in a multi-bottleneck environment with variable delay (Testbed)

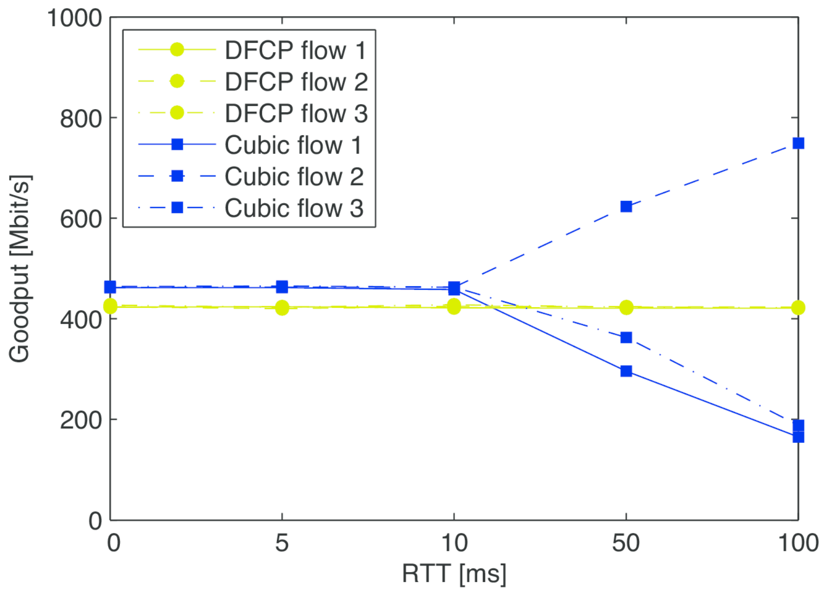


Figure 4.16. The behavior of DFCP and TCP CUBIC in a multi-bottleneck environment with variable delay (Emulab)

The results of a similar measurement performed on the parking lot topology for variable packet loss rate are depicted in Figure 4.17. In this case, the capacities of the bottleneck links were set to 1 Gbps and 500 Mbps, respectively. The bottleneck link B_1 had a fixed packet loss rate of 0.01% whereas it was increased from 0.01% to 5% on B_2. The round-trip time was set to 10 ms on both links. It can be seen that DFCP provides fairness for the flows competing for the available bandwidth of B_2 and their goodput values decrease very slowly as the packet loss rate increases resulting in excellent utilization of both bottleneck links. In contrast, in the case of the first and the third flow, TCP CUBIC only ensures fairness if the packet loss rate is greater than 1% at which value both flows are almost unable to transfer data. The goodput of the first flow starts at a lower value than the goodput of third flow because the first flow goes through both B_1 and B_2, and therefore it is affected by a higher packet loss rate. The link utilization achieved by the two TCP variants is relatively low, since TCP is highly sensitive to packet loss.

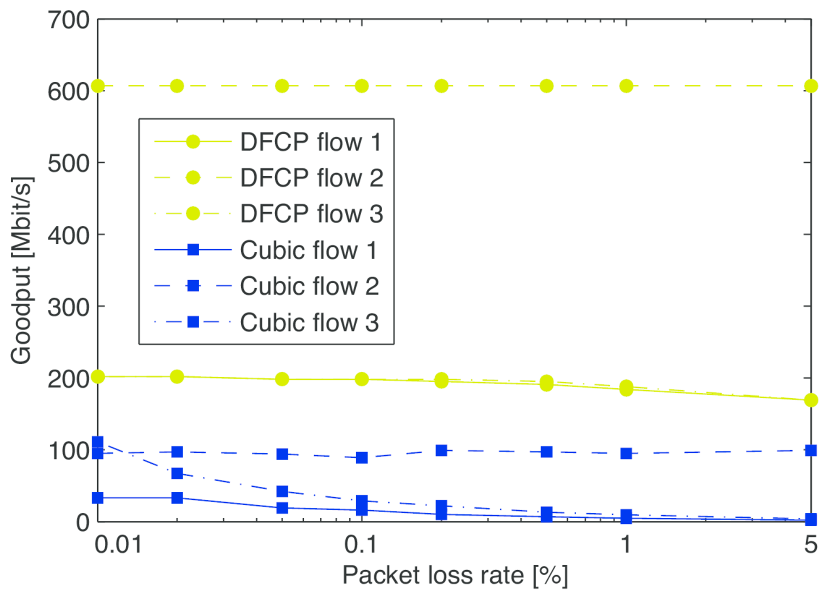


Figure 4.17. The behavior of DFCP and TCP CUBIC in a multi-bottleneck environment with variable packet loss

4.3.5 Scalability

Typically, on a bottleneck link, hundreds of flows compete for the available bandwidth and the capacities of these links are continuously increasing due to the development of communication technologies. *Scalability* is an important requirement for transport protocols, which means that they have to provide similar performance and fairness if the number of flows and the link capacities increase. The following simulations compare the scalability of two fundamentally different data transfer paradigms, TCP CUBIC with DropTail queue management, which is currently used on the Internet, and DFCP with DRR scheduling, which is the new concept. In this case, the measurements were carried out on the topology depicted in Figure 4.1 and they lasted 200 seconds. The buffer size was set to 0.1 BDP and each flow had an RTT of 100 ms.

The performance scalability of the investigated transport protocols for various number of flows and link capacities is shown in Table 4.1.

Table 4.1. Performance scalability

Bandwidth	**Normalized aggregate goodput [%]**		
	10 flows	*50 flows*	*100 flows*
0.1 Gbps	98 / 100	100 / 100	100 / 100
1 Gbps	96 / 100	98 / 100	99 / 100
10 Gbps	22 / 100	95 / 100	96 / 100

The normalized aggregate goodput was computed as the ratio of the aggregate goodput of the concurrent flows to the maximum goodput that can be achieved by a single flow. The normalized values are specified in percentage and are given for TCP CUBIC and DFCP, respectively, separated by a slash mark. It can be observed that DFCP is able to achieve the maximum performance independently of the number of flows and the bottleneck bandwidth. In contrast, in the case of TCP CUBIC, the normalized aggregate goodput increases with the number of flows but decreases with the link capacity. For instance, in the case of a 100 Mbps link, the maximum performance can be achieved by 50 competing flows, however, an increase in the link capacity by two orders of magnitude leads to 5% performance degradation. Moreover, high capacity links cannot be fully utilized by a small number of flows since the round-trip time limits the transmission rate of the individual flows. In this particular case, an RTT of 100 ms results in a goodput reduced to approximately 200 Mbps for each flow (see Figure 4.9 and Figure 4.10), and consequently it leads to the underutilization of the 10 Gbps link in the case of 10 flows.

The fairness scalability of DFCP and TCP CUBIC as a function of time for an increasing number of flows is demonstrated in Figure 4.18 and Figure 4.19.

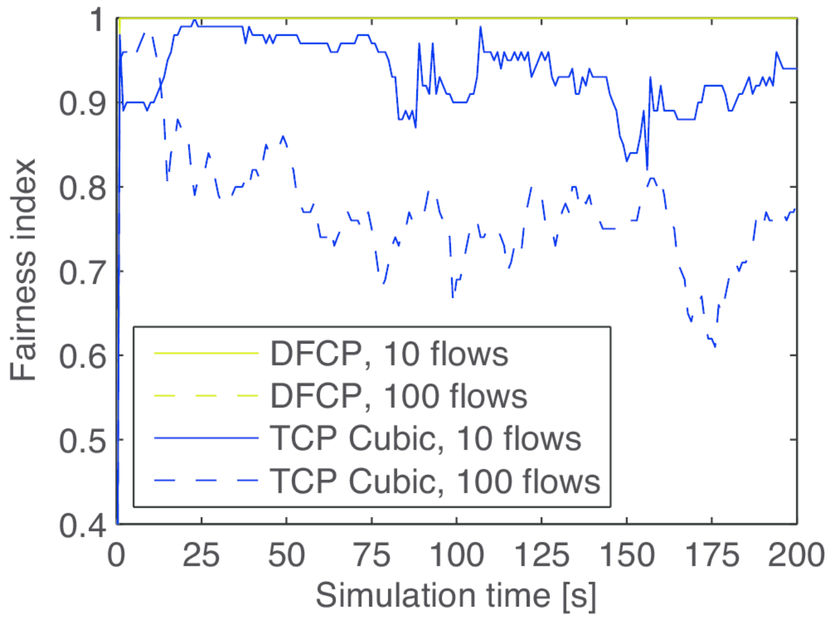


Figure 4.18. Fairness stability in the course of time

The fairness stability of the investigated transport mechanisms for different number of flows is depicted in Figure 4.18. It is clearly shown that the Jain's fairness index of TCP CUBIC fluctuates significantly in the course of time. In addition, an increase in the number of competing flows results in a higher degree of instability and lower mean fairness. The fairness index for an increasing number of flows is illustrated in Figure 4.19.

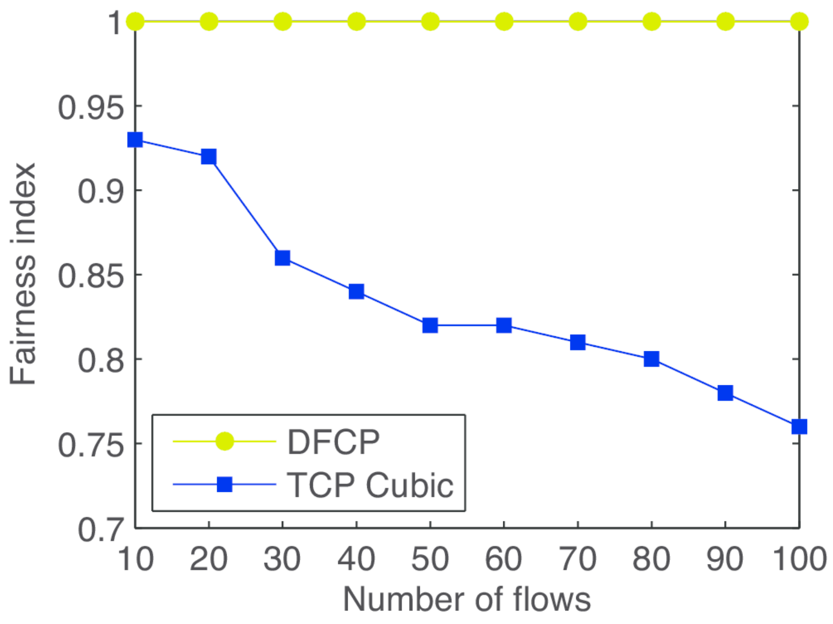


Figure 4.19. The fairness index for an increasing number of competing flows

It has to be emphasized that, in this scenario, each flow had the same delay in order to avoid the phenomenon of RTT unfairness. In the case of TCP CUBIC, the tendency is obvious, the larger the number of the concurrent flows, the smaller the fairness index. However, in contrast to all of these results, DFCP is able to achieve fair bandwidth sharing on various time scales without suffering from stability issues and independently of the number of the competing flows.

In this chapter, two alternative data transport mechanisms for future networks were investigated, the digital fountain code based DFCP and the congestion control based TCP. In order to draw solid conclusions, the operation of DFCP was analyzed on various network topologies (dumbbell and parking lot topologies) and on multiple platforms including the laboratory testbed, the Emulab network environment and the ns-2 network simulator. A comparative performance analysis of DFCP and TCP was also carried out on the previously mentioned platforms. It has been shown that the goodput performance of DFCP is significantly better than the investigated TCP variants for a wide range of packet loss rates and round-trip times. It has also been observed that DFCP is able to achieve maximum performance even in the case of small buffers, which could make it attractive for all-optical networks. Furthermore, DFCP provides fair bandwidth sharing among competing flows independently of their RTTs. Although perfect fairness can only be achieved when fair schedulers (for instance, DRR) are employed, DFCP can ensure better fairness than

TCP even without fair schedulers or if the simple DropTail algorithm is applied. Finally, the digital fountain code based transport also guarantees good scalability and stability in terms of both performance and fairness for an increasing number of flows and for increasing link capacities. The measurement results justify that the promising approach, proposed by GENI, has several advantageous properties and a broad spectrum of possible applications.

Chapter 5

Conclusion and Future Work

TCP, which is the most widely used transport protocol today, is not able to work efficiently and fully utilize the network resources. In order to solve these issues, a number of new TCP variants have been developed by modifying the conventional congestion control algorithm of TCP. However, applying different congestion control mechanisms led to interoperability, efficiency and complexity issues between the TCP variants. Therefore, the commonly used TCP versions do not give universal and optimal solutions to the challenges of today's ever-changing, heterogeneous network environments.

In order to find answers to the performance and interoperability problems of TCP, innovative concepts and ideas have been worked out in the course of research. In this work, a new transport protocol, which is called DFCP, was presented that is based on a promising idea, suggested by GENI, according to which applying congestion control should be completely avoided. In this case, every entity in the network is allowed to send data at its maximum rate and the emerging huge, mostly bursty packet loss is compensated by applying efficient erasure codes.

I have implemented DFCP in the Linux kernel and the protocol is currently under research and development. I have also analyzed the operation and behavior of the protocol on various network topologies and on multiple platforms under a wide range of network conditions by carrying out both testbed measurements and simulations. The results have confirmed that the protocol has several favorable properties and is able to achieve significantly better performance in several network scenarios compared to different TCP variants. It has also been shown that DFCP has a broad spectrum of possible applications highlighting its reduced buffer space requirements which could make it a promising solution for all-optical networks.

After the introduction, Chapter 2 provided an overview of the basic concepts in computer networking. Afterwards, the most widely used high-speed TCP versions were investigated highlighting their congestion control algorithms, as well as the advantages and drawbacks of the applied algorithms. In Chapter 3, the DFCP and its underlying concept were introduced concentrating on the operational phases and basic functions of the protocol. Furthermore, the employed coding scheme and the parameters of the protocol were also detailed. In Chapter 4, the focus was on the testbed measurements and simulations. In this chapter, on the one hand, a validation analysis was performed in order to examine and validate the operation of the protocol. On the other hand, a comparative performance evaluation of DFCP was carried out in order to compare the performance of the protocol to widely applied TCP variants in various network scenarios. Finally, this chapter, Chapter 5, concludes the paper and details the suggested future work and improvements.

Regarding the future work, DFCP is currently under active research and development. In the future, it is crucial to test and analyze the properties of the current implementation of DFCP in order to find its possible drawbacks. Afterwards, all the possible weaknesses need to be understood and eliminated. Furthermore, it is also important to conduct additional testbed measurements and simulations on various, more complex network topologies. In addition, the effects of the parameters of the protocol on its behavior and performance need to be thoroughly investigated. Finally, it is also essential to carry out a more detailed analysis when small buffers are used in the network in order to provide an efficient solution which meets the requirements of future networks.

Bibliography

[1] Alexander Afanasyev, Neil Tilley, Peter Reiher, and Leonard Kleinrock. Host-to-Host Congestion Control for TCP. *IEEE Communications Surveys & Tutorials*, 12(3):304–342, July 2010.

[2] Sándor Molnár, Balázs Sonkoly, and Trinh Tuan Anh. A Comprehensive TCP Fairness Analysis in High Speed Networks. *Computer Communications*, 32:1460–1484, August 2009.

[3] David Clark, Scott Shenker, and Aaron Falk. GENI Research Plan. Version 4.5, Global Environment for Network Innovations, April 23 2007. `http://groups.geni.net/geni/attachment/wiki/OldGPGDesignDocuments/GDD-06-28.pdf`; accessed November 6, 2011.

[4] Neda Beheshti, Yashar Ganjali, Ramesh Rajaduray, Daniel Blumenthal, and Nick McKeown. Buffer sizing in all-optical packet switches. *Optical Fiber Communication Conference*, 2006.

[5] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing Router Buffers. *ACM SIGCOMM Computer Communication Review*, 34(4):281–292, August 2004.

[6] Hyundai Park, Emily E. Burmeister, Staffan Bjorlin, and John E. Bowers. 40-Gb/s Optical Buffer Design and Simulation. In *Proceedings of the 4th International Conference on Numerical Simulation of Optoelectronic Devices*, pages 19–20, Santa Barbara, CA, USA, 2004.

[7] Sándor Molnár, Zoltán Móczár, András Temesváry, Balázs Sonkoly, Szilárd Solymos, and Tamás Csicsics. Data Transfer Paradigms for Future Networks: Fountain Coding or Congestion Control? *IFIP Networking 2013*, pages 1–9, 2013.

[8] Van Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM 1988*, pages 314–329, Stanford, CA, USA, August 16–18 1988.

[9] Sally Floyd. Highspeed TCP for Large Congestion Windows. RFC 3649, Internet Engineering Task Force, December 2003. `http://www.ietf.org/rfc/rfc3649.txt`; accessed November 1, 2011.

[10] Tom Kelly. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review*, 33(2):83–91, April 2003.

[11] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *Proceedings of IEEE Infocom 2004*, volume 4, pages 2514–2524, Hong Kong, China, March 7–11 2004.

[12] Injong Rhee and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. In *Proceedings of Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2005)*, Lyon, France, February 3–4 2005.

[13] Lawrence S. Brakmo and Larry L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on selected Areas in communications*, 13:1465–1480, 1995.

[14] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN)*, 14(6):1246–1259, 2006.

[15] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound TCP approach for highspeed and long distance networks. In *Proceedings of IEEE Infocom 2006*, Barcelona, Spain, April 23–29 2006.

[16] Ren Wang, Massimo Valla, M. Y. Sanadidi, and Mario Gerla. Adaptive bandwidth share estimation in TCP Westwood. *GLOBECOM*, pages 2604–2608, 2002.

[17] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of ACM SIGCOMM 2002*, Pittsburgh, PA, USA, August 19–23 2002.

[18] Andrew S. Tanenbaum. *Számítógép hálózatok, Második, bővített, átdolgozott kiadás.* Panem, Budapest, 2004. ISBN: 963–545–384–1.

[19] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, 26(3), July 1996.

[20] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, Internet Engineering Task Force, April 2012. `http://tools.ietf.org/html/rfc6582`; accessed July 11, 2014.

[21] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgment Options. RFC 2018, Internet Engineering Task Force, October 1996. `http://tools.ietf.org/html/rfc2018`; accessed July 12, 2014.

[22] Mark Allman, Dan Glover, and Luis Sanchez. Enhancing TCP Over Satellite Channels using Standard Mechanisms. RFC 2488, Internet Engineering Task Force, January 1999. `http://tools.ietf.org/html/rfc2488`; accessed July 12, 2014.

[23] Barath Raghavan and Alex C. Snoeren. Decongestion Control. In *Proceedings of the 5th ACM Workshop on Hot Topics in Networks (HotNets-V)*, Irvine, CA, USA, November 2006.

[24] Thomas Bonald, Mathieu Feuillet, and Alexandre Proutiére. Is the "Law of the Jungle" Sustainable for the Internet? In *Proceedings of INFOCOM 2009*, Rio de Janeiro, Brazil, 2009.

[25] Luis López, Antonio Fernández, and Vicent Cholvi. A game theoretic comparison of TCP and digital fountain based protocols. *Computer Networks*, 51(12):3413–3426, 2007.

[26] Shakeel Ahmad, Raouf Hamzaoui, and Marwan Al-akaidi. Robust Live Unicast Video Streaming with Rateless Codes. In *Proceedings of 16th International Workshop on Packet Video, PV 2007*, pages 78–84, Lausanne, Switzerland, November 2007.

[27] Anghel Botos, Zsolt A. Polgar, and Vasile Bota. Analysis of a transport protocol based on rateless erasure correcting codes. *IEEE International Conference on Intelligent Computer Communication and Processing*, 1:465–471, 2010.

[28] Dinesh Kumar, Tijani Chahed, and Eitan Altman. Analysis of a Fountain Codes Based Transport in an 802.11 WLAN Cell. *International Teletraffic Congress*, pages 1–8, 2009.

[29] Michael Luby. LT Codes. In *Proceedings of 43rd Symposium on Foundations of Computer Science (FOCS 2002)*, pages 271–280, 2002.

[30] Amin Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6), June 2006.

[31] Michael Luby, Amin Shokrollahi, Mark Watson, Thomas Stockhammer, and Lorenz Minder. RaptorQ Forward Error Correction Scheme for Object Delivery. RFC 6330, Internet Engineering Task Force, August 2011. `http://tools.ietf.org/html/rfc6330`; accessed July 21, 2014.

[32] Sameer Seth and M. Ajaykumar Venkatesulu. *TCP/IP Architecture, Design and Implementation in Linux.* Wiley-IEEE Computer Society Pr, 2008. ISBN: 9780470147733.

[33] Thomas Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems.* Charles River Media, Inc., Rockland, MA, USA, 2004. ISBN: 1584502843.

[34] Amin Shokrollahi. LDPC codes: An Introduction. *Digital Fountain, Inc., Tech. Rep*, April 2 2003.

[35] Robert C. Tausworthe. Random Numbers Generated by Linear Recurrence Modulo Two. *Mathematics of Computation*, 19:201–209, 1965.

[36] Zoltán Móczár, Sándor Molnár, and Balázs Sonkoly. Multi-Platform Performance Evaluation of Digital Fountain Based Transport. *SAI 2014*, pages 690–697, 2014.

[37] Martin Bateman and Saleem N. Bhatti. TCP Testing: How Well Does ns2 Match Reality? *The IEEE 24th International Conference on Advanced Information Networking and Applications*, 0:276–284, 2010.

[38] Emulab - Network Emulation Testbed Home. `http://www.emulab.net/`; accessed September 9, 2014.

[39] The Network Simulator - ns-2. `http://nsnam.isi.edu/nsnam/index.php/Main_Page`; accessed November 25, 2012.

[40] Marta Carbone and Luigi Rizzo. Dummynet. `http://info.iet.unipi.it/~luigi/dummynet/`; accessed April 22, 2012.

[41] Sam Jansen and Tony McGregor. Network Simulation Cradle. `http://research.wand.net.nz/software/nsc.php`; accessed November 25, 2012.

[42] Eric Gavaletz and Jasleen Kaur. Decomposing RTT-Unfairness in Transport Protocols. *IEEE Workshop on Local and Metropolitan Area Networks*, pages 1–6, May 2010.

[43] Sebastian Kaune, Konstantin Pussep, Christof Leng, Aleksandra Kovacevic, Gareth Tyson, and Ralf Steinmetz. Modelling the Internet Delay Space Based on Geographical Locations. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 301–310, Weimar, Germany, 2009. IEEE Computer Society.

[44] Dah-Ming Chiu and Raj Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17(1):1–14, June 10 1989.

List of Figures

List of Tables