Visualizing the limiting factors for the speed of TCP in realtime

Bachelor Thesis (269950)

Author: L.H.X. Hobert

Supervisors: Dr.ir. Aiko Pras
Dr.ir. Pieter-Tjerk de Boer
M.Sc. Tiago Fioreze

Design and Analysis of Communication Systems
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

June 28, 2006

Acknowledgments

This work is the result of a 9 months lasting bachelor assignment. It is the final step in receiving my bachelor degree in telematics at the University of Twente, Enschede.

Taking this last step was not easy. During the meeting with Aiko Pras and Tiago Fioreze, two of my main supervisors, I have learnt a lot in the area of measurement and in planning and approaching a project like this one. They made it also possible for me to perform my hobby rowing. In addition, I would like to thank Tiago for helping and pushing me in writing this work. Without your effort I could have never achieved this. Finally, I would thank my supervisors, including Pieter-Tjerk de Boer, for reviewing this work and giving me tips to improve my work.

Besides my supervisors, I would thank my family in supporting me during my study. They support me in every aspect of my study and without their support I would have never received my bachelor degree.

Finally, I would like to thank my rowing team “ouderejaars licht” and my coaches for a fantastic year. Sometimes the stress of my assignment made me grumpy but many times the training relieved me for several hours from that stress.

"Freedom is not worth having if it does not include the freedom to make mistakes".

Mahatma Gandhi
political and spiritual leader of India (1869 - 1948)

Abstract

This work presents methods that detect the limiting factors for the speed of TCP in realtime. The limiting factors under consideration are the receive window, network and application limitation. The similarities in functionality between these methods are grouped in a framework in this work. In addition, this work shows the implementation of the realtime detection methods in order to show in realtime the limiting factors for the speed of TCP streams using a graphical interface.

Contents

Chapter 1

Introduction

1.1 Context

The monitoring of network activity is an important task in the network management area. Network managers may be able to plan improvements in his or her managed networks and even detect malicious activity by characterizing traffic that pass through the managed network.

Currently, the monitoring of traffic flows can be performed by using two major approaches:

- **Active monitoring**: data are injected into the network in order to perform measurements. An important drawback of this type of monitoring is that it will interfere with data flowing through the network.
- **Passive monitoring**: data are captured from the network without interfering the network activity.

By using these approaches, network managers may draw a health status of the managed network in order to detect congestions and evaluate performance.

1.2 Motivation

The Transport Control Protocol (TCP) is one of the most used transport protocol on the Internet, therefore there is a great interest in the performance [PFTK00, LM97] and the limiting factors for the speed of TCP [ZBPS02]. Mark Timmer, a graduated bachelor, has identified four speed limiting factors of a TCP stream: the receive window, sender buffer, network and application limiting factors. In addition, he developed an offline method to identify the limiting factors for the speed of TCP streams by using passive measurement [Tim05]. He also developed during his bachelor assignment, a tool to make offline analysis of TCP streams out of a repository.

System and network administrators may be interested in finding out what factors are limiting the speed of TCP streams in networks managed under their supervision in realtime. Using the Mark Timmer's tool, they are only able to perform an offline measurement of the limiting factors for the speed of TCP streams. The offline method requires that packets are first captured and stored

in a repository before they can be analyzed. As a consequence, there might be events that could limit the transmission capacity of some TCP connections, but they would be detected only when the repository was analyzed. This might not allow network administrators to quickly respond to events that are disturbing the TCP connections, for instance.

In order to solve these problems, this work presents a method that split TCP connection into parts to determine a part's limiting factor. A connection can be split up in, for instance, in equal periods of time, i.e., the duration of each part will be the same, and also in the number of packets, i.e., the number of packets in each part will be the same. For each part of a TCP connection, the methods presented in this paper will identify its limiting factor in realtime.

1.3 Research question

Based on the motivation presented, the main research question of this work is "How to show, in realtime, the receive window, network and application speed limiting factors of a TCP stream?" This research question can be subdivided into multiple questions:

- How to identify in realtime those limiting factors for the speed of a TCP flow?
- How to implement a program capable of showing those speed limiting factors in realtime?

1.4 Research approach

In order to identify in realtime the speed limiting factors, this work presents new methods based on Mark Timmer's methods. Unlike Mark Timmer's method, we assume that TCP streams may be limited by multiple of the speed limiting factors, identified in [Tim05], over its lifetime. Therefore, we propose methods that determine the limiting factor for the speed of a connection. In order for these methods to perform measurements they do not need to process the full connection first as is done in the methods of Mark Timmer. In addition to these methods, a new approach is used to detect a network limitation is presented in this work. This method keeps track of the size of the congestion window for certain major TCP implementations. In order to do that, the method presented in this work need to know what the cause is for a packet to be out of sequence. To determine this cause in realtime, this work proposes an adaption of the method proposed in [JID+02], which is described in further details in Section 3.2.
In addition, this work also presents a graphical tool that enables network managers to visualize the limiting factors for the speed of TCP streams flowing through their network. This tool is designed by using the principles of a finite state machine in order to identify the speed limiting factors in realtime. Based on the theoretical design of the methods to detect the speed limiting factors, the finite state machines are designed using UML state diagrams. For each speed limiting factor, the percentage of all connections limited by a certain factor is determined. This percentage is then visualized using a pie-diagram. Further details about this tool is presented in Section 4.1

1.5 Contribution

This work has focused on the improvement of Mark Timmer's methods in order to measure in realtime the limiting factors for the speed of a TCP stream [Tim05]. Based on these methods, a tool has been designed and implemented in order to visualize in realtime the limiting factors of TCP streams.

1.6 Intended audience

This work is intended for two groups of audience:

- The first group are students who intend to extend the tool by adding new measurement methods, analyze methods or output interfaces.
- The second group are system and network administrators.

1.7 Structure of paper

Chapter 2 gives an overview of the state of the art in the area of identifying the limiting factors for the speed of TCP. Then Chapter 3 presents the development of methods to identify in realtime these speed limiting factors. Chapter 4 describes the tool developed in this work to measure the speed limiting factors in realtime. Chapter 5 then provides a manual for users interested in using the tool. Finally, the conclusions of this work are presented in Chapter 6.

Chapter 2

State of the art

Research on the identification of the limiting factors for the speed of TCP has been developed [ZBPS02, Tim05]. The most relevant research to be presented in this chapter is that one developed by Mark Timmer. The main reason for that is that his methods are used in this work as a base to develop a realtime method for identifying limiting factors for the speed of a TCP flow.

One of the research questions that is answered in [Tim05] is "What factors can limit a TCP flow?". Mark Timmer states in his work that four types of speed limiting factors can be identified:

- Receive window limitation
- Sender buffer limitation
- Network limitation
- Application limitation

In this work only the receive window, network and application limitation factors are considered. The sender buffer limiting factor, which is based on keeping track of the maximum number of outstanding bytes, is not considered as [TdBP06] show this limitation occurs rarely.

This chapter therefore gives a brief overview on what kind of situations the receive window, network and application limitations do occur and how these situations can be determined using the methods described in [Tim05]. For a more detailed explanation about Mark Timmer's methods, see [Tim05].

2.1 Receive window limiting factor

In order to cope with the difference in the speed of receiving correct data from a sending entity and reading this data by an application program, a receiving entity will use a buffer. There may be a situation where the application program does not read the data in the buffer fast enough to empty it. In a situation like that, the sender can continuously send more data and, as a result, the buffer can get overloaded. In order to prevent such a situation, the receiver entity has the ability to limit the speed at which a sender is sending data by advertising the space left in its buffer: the TCP receive window. The sender keeps a buffer

of unacknowledged data in order to perform retransmissions when necessary. The amount of unacknowledged data in the sender's buffer may not be greater than the receive window as it is described in the specification of TCP [Pos81]. So when the size of the sender buffer is equal to the receive window the sender may not send more data until data in the sender's buffer is acknowledged.

To detect the situation where a receive window limitation does occur, two quantities need to be known:

- **The number of unacknowledged bytes at the sender entity, i.e., the outstanding bytes**: the difference between the sequence number of a packet and the largest acknowledgment number seen so far produces a lower limit on the number of outstanding bytes at the sender.
- **The number of bytes allowed to send by the sender as advertised by the TCP receive window**: this quantity sets a maximum limit on the number of outstanding bytes in the sender.

Mark Timmer describes in his work two faulty methods to determine the receive window known by the sender: the last measured receive window and the highest measured receive window. According to Mark Timmer, a wrong decision can be made if the number of outstanding bytes are compared with the last or highest receive window, as described in his work.

A solution proposed in [Tim05] is to keep track of the maximum next sequence number (`maxNextSeqNr`), i.e. the maximum value of a DATA packet's sequence number plus its payload length over all data packets received at a certain moment, and the maximum allowed next sequence number (`maxAllowedNextSeqNr`), i.e., the maximum value of a ACK packet's acknowledgement number plus its advertising receive window over all ACK packets received at a certain moment. At first sight, a receive window limitation is indicated by the equality of the quantities `maxNextSeqNr` and `maxAllowedNextSeqNr`. Besides this criterium, other criteria may also indicate a receive window limitation. If the sender is reluctant to send non-full packets or it transmits data in blocks, other criteria may apply [Tim05]. A stream is considered receive window limited if more than 50 percent of the receiver window limitation checks indicate a receiver window limitation of the stream

2.2 Network limiting factor

Mark Timmer's method to analyze the network limiting factor compares the average bandwidth against the average achievable bandwidth of a TCP stream using the TCP friendly formula (see Formula 2.1). The TCP friendly formula is normally used as a congestion algorithm for applications not using TCP as a transport layer protocol [Kla05].

The actual bandwidth is calculated at the end of each flow by dividing the amount of data send (excluding the retransmitted data) by the lifetime of the connection. This gives an average bandwidth for the stream.

$$BW = \frac{MSS}{RTT} \cdot \frac{c}{\sqrt{p}} \tag{2.1}$$

In order to calculate the average achievable bandwidth (BW) several quantities need to be known. First the maximum segment size (MSS) needs to be

measured. This quantity can be measured by keeping track of the maximum payload length of all DATA packets processed. Second the round-trip time needs to be measured (RTT). The round-trip times is a value that indicates the time elapsed for a message to travel to an end host and back. Third a constant (c) has to be chosen. In [Tim05] a value of $\sqrt{\frac{3}{4}}$ is used. This value should be used when delayed acknowledgements are used by the TCP connection [PFTK00]. The last quantity to be measured is the loss rate (p). This quantity is determined by calculating the ratio between the number of loss events [Tim05] and the number of packets.

For example, a common and default value for the maximum segment size of TCP packets is 536 bytes [Pos81]. In combination with a value of 0.05 seconds for the round-trip time, $\sqrt{\frac{3}{4}}$ as constant and a loss rate of $\frac{1}{1000}$, i.e., one out of a thousand packets get lost. With these values the average achievable bandwidth, according to Formula 2.1, is:

$$\frac{536}{0.05} \cdot \frac{\sqrt{\frac{3}{4}}}{\frac{1}{1000}} \approx 9.3 Mbit per second$$

According to Mark Timmer, a stream is considered network limited if the average bandwidth is 50 percent of the average achievable bandwidth.

2.3 Application limiting factor

The last limiting factor this work addresses is the application limiting factor. This limiting factor is subdivided into two situations:

- **Lack of data**: a TCP stream may be limited in its throughput in the situation where the application program has no data to send. An application program such as an instant messenger sends data via TCP when a user has typed a message. This situation can be measured by looking at the number of outstanding bytes: zero outstanding bytes indicate the situation where a TCP stream is limited by an application limiting factor. Only in the situation where the TCP stream is also receive window limited, this criteria does not apply. A TCP streams with an application program capable of providing enough data has almost the whole time multiple bytes of outstanding data.

  According to Mark Timmer, a connection can be considered limited by a lack of data if at least two percent of the time the connection has zero byte of outstanding data. When this limitation is detected other limiting factors do not apply anymore to the connection as this limiting factor is dominant above all others [Tim05].

- **Application layer acknowledgements or requests**: the protocol used at application level may contain a feature to control the flow of data. For instance, it can contain the feature to control retransmissions or the throughput. As the methods in [Tim05] do not measure at application level due to the number of protocols available, we cannot measure these features. Since the methods are designed to process only half-duplex flows,

acknowledgment packets with a payload of even a few bytes gives an indication that these features are present.

A connection is considered limited by application layer acknowledgements or requests if at least ten percent of the number of ACK packets contain a payload. As the detection of a payload in ACK packets does not indicate for a hundred percent that connection is limited by this factor, this method applies only when other limiting factors are not detected.

So far, in this chapter, we have looked at a summary of techniques used to detect a receive window, network and application limitation of a TCP stream, developed in [Tim05]. The next chapter will describes how these methods can be modified in order to identify limiting factors for the speed of a TCP stream in realtime.

Chapter 3

Realtime detection methods

In order to develop a realtime alternative for the methods described in [Tim05], some additional requirements need to be met by the detection methods described in this work. Before start describing the additional requirements for each speed limiting factor, it is important to know a common property that is used in all detection methods. These common properties are present in the methods of Mark Timmer, but are not explicitly addressed by him in his work.

Every detection method that is considered in this work, can be split up into two parts (see Figure 3.1). The detection method is split up in parts in order to assign specific requirements to a single part.

The first part, called the measurement part, is responsible for processing TCP packets. Moreover, it contains criteria in order to decide whether the TCP stream was limited by a particular speed limiting factor over a certain period of time. The second part, called the interpretation part, combines multiple measurements, taken by the measurement part, in order to decide whether a TCP stream is limited over a certain period of time.

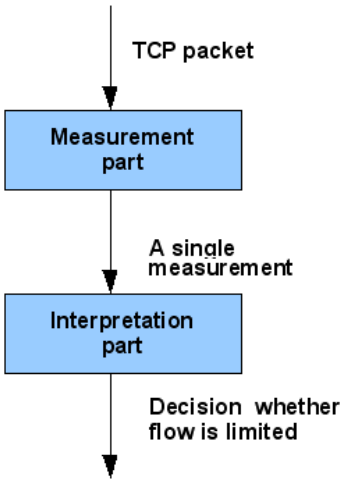

Figure 3.1: Split up detection method into parts

A common requirement for all speed limiting factors is that the criteria perform their measurements based on the last received packet without the need to reprocess packets received before. Resources like memory and CPU cycles are scarce so storing packets for reprocessing can put a great burden on these resources. This burden can eventually even become a threat if not enough resources are available to process a TCP packet in time.

3.1 Realtime method for receive window limitation detection

The criteria developed in [Tim05] enables the methods in the measurement part to perform the measurements efficiently; it processes packets only once and in the order how they are received. Therefore we can use these criteria to measure a single realtime receive window – ReRwnd – limitation without any modifications.

The major problem we have to deal with in the development of a realtime method for receive window limitation detection is: how can we determine of a part of a TCP connection if this part is limited by a receive window limitation? The method used in [Tim05] cannot be used as it determines for a complete connection whether it is limited. In order to make this decision it has to process the whole connection first. In order to deal with this problem, this paper provides two solutions:

Time-based ReRwnd solution

The time-based solution works by making a decision whether a connection was ReRwnd limited over a certain period of time. In order to do that, this solution will keep track of two counters for each connection (See Algorithm 1 for pseudo code):

- **numberOfLimitations**: this counter is increased by one every time the measurement part measures a ReRwnd limitation.
- **numberOfMeasurements**: this counter is increased by one every time the measurement part take a measurement in order to determine if the connection is ReRwnd limited at that moment.

```
1 numberOfMeasurements := 0;
2 numberOfReRwndLimitations := 0;
3 for each RwRwnd measurement performed do
4 |   numberOfMeasurements++;
5 |   if measurement was ReRwnd limited then
6 |   |   numberOfReRwndLimitations++;
7 |   end
8 end
```

Algorithm 1: Pseudo code Keeping track of the measurements

Algorithm 2 provides pseudo code for this solution to make a decision whether the flow was limited over a part of its lifetime. This algorithm performs in parallel with Algorithm 1 and they share the variables `numberOfMeasurements` and `numberOfReRwndLimitations`. The parameter to this algorithm will be the number n. This number determines the time interval between two decisions by the interpretation part. If n is equal to the lifetime of a connection, then this method will behave the same as the solution described in [Tim05].

```
   input: n; The number of seconds between two decisions
1  for every n seconds do
2  |  if numberOfReRwndLimitations > 0.5 * numberOfMeasurements
   |  then
3  |  |  signal RERWNDLIM;
4  |  else
5  |  |  signal not RERWNDLIM;
6  |  end
7  |  numberOfReRwndLimitations := 0;
8  |  numberOfMeasurements := 0;
9  end
```

Algorithm 2: Pseudo code Time-based ReRwnd solution

Check-based ReRwnd solution

The time-based solution proposes an algorithm to decide after a specified time whether or not a TCP connection was ReRwnd limited over the last period. The check-based solution proposes an algorithm that will decide after a specified number of checks by the measurement part whether or not a TCP connection was ReRwnd limited. Algorithm 3 shows pseudo code of the check-based solution.

```
   input: m; The number of measurements between two decisions
1  numberOfMeasurements := 0;
2  numberOfReRwndLimitations := 0;
3  for each measurement performed do
4  |  numberOfMeasurements++;
5  |  if measurement was ReRwnd limited then
6  |  |  numberOfReRwndLimitations++;
7  |  end
8  |  if numberOfMeasurements == m then
9  |  |  if numberOfReRwndLimitations > 0.5 * numberOfMeasurements
   |  |  then
10 |  |  |  signal RERWNDLIM;
11 |  |  else
12 |  |  |  signal not RERWNDLIM;
13 |  |  end
14 |  |  numberOfReRwndLimitations := 0;
15 |  |  numberOfMeasurements := 0;
16 |  end
17 end
```

Algorithm 3: Pseudo code Check-based ReRwnd solution

In each solution, the decision whether a streams was ReRWnd limited is made if more than 50 percent of the detections were RwRwnd limited. This threshold is copied from the work of Mark Timmer.

In the implementation of our tool, this work explains which method is used in the implementation of the interpretation part.

3.2 Realtime method for network limitation detection

Each implementation of the TCP protocol keeps track of a variable called congestion window (`cwnd`). This variable stores the maximum number of outstanding bytes in order to avoid congestion on the network. In the case where the number of outstanding bytes is limited due to the congestion window, the TCP flow is network limited.

A TCP stream may also be network limited without the congestion window imposing that limitation. This situation is described in [TdBP06]. In this work we will not describe how this situation can be measured as this article was discovered in one of the last phases of this work.

The most straightforward method to detect this situation is to measure this `cwnd` variable and compare it to number of outstanding bytes. Although, this method is the most straightforward, it is not the easiest method as this variable is never advertised by the sender (as it is done for the receive window variable).

The method described by Mark Timmer indirectly measures the congestion window; the achieved bandwidth is directly related to the congestion window of a TCP stream (in the case the network is the only limiting factor of a TCP stream). This achieved bandwidth is compared to the maximum achievable bandwidth in order to determine a network limitation. This method can be used without a problem in offline detection. Meanwhile, this method causes some problems during realtime analysis: during a peak in available bandwidth, the achieved bandwidth can be greater than the average achievable bandwidth. Using a network limitation detection based on the average achievable bandwidth may indicate a limitation of the TCP stream while this is not the case. Figure 3.2 illustrates this situation.

In order to determine the network limiting factor, multiple types of methods can be used. One of the types makes use of the congestion avoidance principle used in TCP Vegas, called delay-based congestion control. Briefly it detects a congestion in the network if the round-trip time increases due to full router buffers. However, an increase of the round-trip time does not directly indicates a network congestion as described in [PJD04].

Another technique, used in this work, is described in the next subsection.

Inferring technique

This method, as described in [Jai05], tries to keep track of the congestion window at the sender. In order to do so, first out-of-sequence packets are classified as “retransmission”, “reordering”, “duplicate” or “unneeded retransmission”. This process of classifying out of sequence packets is described in Subsection **Out of sequence classification**. These classifications are used to simulate the congestion window of three major TCP implementations (Tahoe, Reno and NewReno). The process of simulation of these TCP implementations is described in Subsection **Simulation**. Finally, one TCP simulation is picked as most likely being the sender’s TCP implementation. In order to determine a

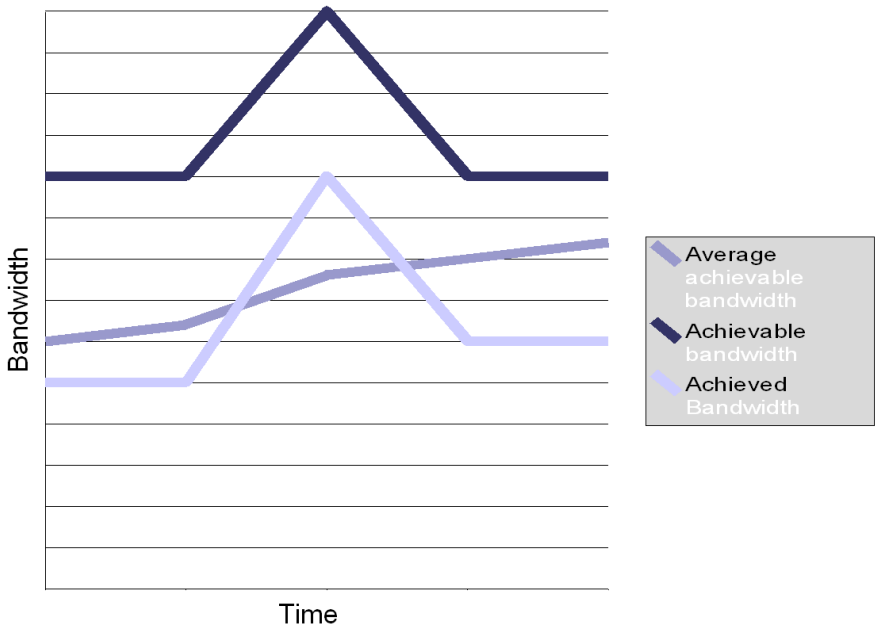

Figure 3.2: Illustration of situation where bandwidth is greater than the average achievable bandwidth, but not network limited.

network limitation, the congestion window of this picked simulation is compared to the number of outstanding bytes. As also the case in detecting the receive window limitation, multiple criteria do apply for the detection of a network limitation. These criteria are discussed in Subsection **Detection criteria**.

Out of sequence classification

The method used to classify out of sequence packets is described in [JID+02]. This methods keeps track of the tuple (`W`, `x`, `t`) for each captured DATA packet. The value `W` is the identifier in the IP packet, `x` is the TCP sequence number and `t` is the time when the packet was captured. When an out of sequence packet is received (a DATA packet with a sequence number `x` smaller or equal to the highest sequence number capture so far), this packet is classified by the decision process shown in Figure 3.3. Jaiswal describes this process in more detail in [JID+02].

Looking at the decision process, it shows us a possible problem: the second decision in the process determines if a packet is captured before. At first sight, this requires to store the tuples of all packets captured. However, this is not necessary and might even cause problems: the wrapping of sequence numbers by TCP may indicate that a packet is received before; while actually the sequence number is reused. As acknowledged out-of-sequence packets are classified as "unneeded retransmission", an implementation of the decision process should only store unacknowledged tuples. In addition, this solves the prob-

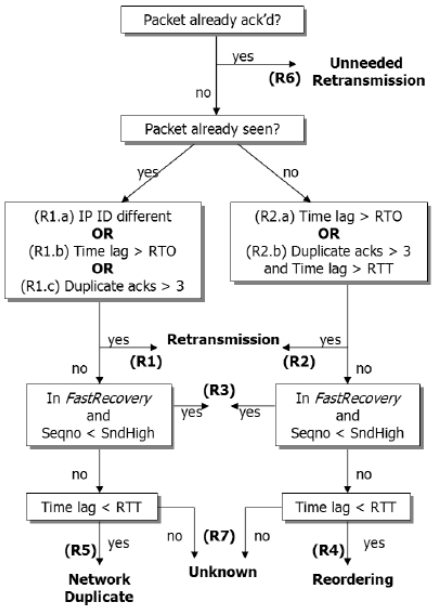

Figure 3.3: Classifying out-of-sequence DATA packets

lem of wrapping since sequence numbers cannot be reused as long as they are unacknowledged.

Simulation

The simulations of the congestion window of the three major TCP implementations are simulated using an FSM. For each implement an FSM models the behavior of the sender. In order to perform the measurement of the congestion window in realtime the methods should perform its measurements based on the current processed packet. Since the original methods satisfy this requirement it needs no adaptations.

The three TCP implementations have in common that they all have the "slow start" and the "congestion avoidance" state. The implementations of the algorithms in these states is actually required by [Bra89]. The behavior is also common in each TCP implementation, exception for the reception of duplicate acknowledgement packets in TCP Reno and NewReno.

In addition to the "slow start" and "congestion avoidance" states, TCP Reno and NewReno have also the "fast retransmit" and "fast recovery" phases. The two extra phases implement the "fast retransmit algorithm" and "fast recovery algorithm", respectively, as described in [Ste97].

The difference between TCP Reno and NewReno is the behavior of the "fast recovery algorithm". TCP Reno change to the "congestion avoidance" state on the reception of a non-duplicate acknowledgement packet. TCP NewReno make this change only in the case where this non-duplicate acknowledgement packet acknowledges all bytes send before entering the "fast retransmit" phase.

The simulation of the congestion window has some uncertainties [Jai05]. The congestion window is under-estimation if some of the three duplicate acknowledgement packets gets lost between the measurement and the sender. Since we are measuring at the sender side, this scenario is not very likely. Meanwhile, the over-estimation (see [Jai05]) of the congestion may occur more likely in our situation; The distance between the measure point and the receiver is larger than in the case where the measurement is performed in the middle. Therefore, the possibility that acknowledgement packets are lost is more likely.

Jaiswal concludes in [Jai05] the description of the method that the simulated congestion window of 90 percent of the connections is within $0.5 \cdot MSS$ of the actual congestion window. This conclusion is used in the detection criteria in order to detect a network limitation.

Detection criteria

In order to detect a network limitation, one of the three simulating TCP FSMs is chosen to be most likely the TCP implementation of the sender. In order to choose the best FSM the number of outstanding bytes is checked against the size of the receive window and congestion window of the simulation FSM. Since we assume that the number of outstanding bytes does not limit the stream, the FSM violates the TCP specification `cwnd` if the number of outstanding bytes exceed the congestion window(`cwnd`). The FSM with the least number of violations is chosen to be the best simulation FSM.

Finally, when the best FSM is chosen, its value of `cwnd` is compared to the number of outstanding bytes in order to determine the network limitation. As we saw in detection the receive window limiting factor, the sender can have multiple sending schemes. This also results in multiple criteria to detect a network limitation; for each sending schema one criteria.

The first, and most straightforward, criteria is the complete window utilization. Algorithm 4 shows the algorithm of this criteria.

```
1 if cwnd-0.5*MSS < outstandingBytes cwnd+0.5*MSS then
2 |   return COMPLETECONGESTIONWINDOWUTILIZATION;
3 end
```

Algorithm 4: Criteria complete congestion window utilization

This criteria checks for the situation where the number of outstanding bytes is equal to the congestion window (`cwnd`), taking into account the uncertainties in tracking the congestion window. This criteria is used in the situation where the sender uses a complete window sending scheme.

Besides the complete window sending schema, the sender may use the maximum segment size schema to send data. In this case, the sender only transmits packets when its has enough data to fill a complete packet. The pseudo code of this criteria is showed in Algorithm 5.

Last but not least, the sender may use the block sending schema. In that situation the sender sends a burst of data equal to a block size, for instance, 4096 bytes. This burst is usually send over multiple packets. Algorithm 6 shows the pseudo code of this criteria.

```
1 if cwnd - MSS - 0.5*MSS < outstandingBytes < cwnd + 0.5*MSS&&
  sizePreviousPacket == MSS then
2 |  return MSSCONGESTIONWINDOWUTILIZATION;
3 end
```

Algorithm 5: Criteria MSS utilization

```
1 if this flow is block based then
2 |  if cwnd - blockSize - 0.5*MSS < outstandingBytes < cwnd +
  |  0.5*MSS&& sizePreviousPacket == endPacketLength then
3 |  |  return BLOCKCONGESTIONWINDOWUTILIZATION;
4 |  end
5 end
```

Algorithm 6: Criteria Block size utilization

The variable `endPacketLength` is explained in the work of Mark Timmer. His work presents an algorithm to determine if a flow is send by a sender using a block-based sending schema.

The major drawback of this detection method is that it cannot track the congestion window of TCP implementations using a loss-based congestion control such as TCP Vegas.

Until now, we have described the measurement part of the network limitation detection. As with other detection methods, this detection method also consist of a measurement part. In the next section this work describes two algorithms in order to determine in realtime if a stream is network limited – ReNwLim – over a part in its lifetime.

Time and check-based ReNwLim solution

The Time and Check-based ReNwLim solutions are almost the same as the time and check-based ReRwnd solutions, respectively. The only difference is that the Time and Check-based ReNwLim solutions use the variable `numberOfReNwLimitations` instead of `numberOfReRwndLimitations`. Therefore, this part is not repeated here. For a description of these solutions see Section 3.1.

3.3 Realtime method for application limitation detection

A TCP stream can be limited at the application layer in two scenarios: due to the lack of data by the application to send and due to acknowledgments or request at application level. The development of a realtime method to detect these scenarios are based on the methods developed by Mark Timmer and are described in the next two subsections, respectively.

Lack of data

The detection of this scenario – ReAppLack – is based on the detection of the number of outstanding bytes. In the detection method the period is measured where a sender has zero outstanding bytes. The criteria described in [Tim05] perform its measurement based on the current packet without the need to re-process packets, therefore it needs no adaption in order to measure in realtime.

As with the ReRwnd limitation detection, the major problem is to determine in realtime if a connection was limited over a part of its lifetime. In order to solve these problems two solution are provided: a "Time-based ReAppLack" and a "Check-based ReAppLack solution". These solutions have many similarities with the solutions provided in Subsection 3.1; Both time-based solutions determines their limitation after a period in time whereas the check-based solution determines this limitation after a certain number of limitation checks by the measurement part.

In the next two subsections these two solutions are described.

Time-Based ReAppLack method

To determ each n seconds if a flow is ReAppLack limited, the periods of zero outstanding bytes have to be counted. Pseudo code in Algorithm 7 presents how this period is tracked in variable `zeroOutStandingBytesPeriod`.

```
1 zeroOutStandingBytesPeriod := 0;
2 for each zeroOutStandingBytes period do
3 |   zeroOutStandingBytesPeriod = zeroOutStandingBytesPeriod +
  |   measuredPeriod;
4 end
```

Algorithm 7: Pseudo code keeping track of periods with zero outstanding bytes

In the time-based method to determine if a connection is ReAppLack limited, the time-based algorithm makes that decision every n seconds. This n is a parameter passed to the algorithm (See Algorithm 8 for pseudo code). This solution runs in parallel with Algorithm 7 and shares the variable `zeroOutStandingBytesPeriod`. Every n seconds the methods determines if the variable `zeroOutStandingBytesPeriod` was more than 2 percent of n seconds. In that case the connection was ReAppLack limited. Algorithm 8 provides the pseudo code of this algorithm.

```
  input: n; The number of seconds between two decisions
1 for every n seconds do
2 |   if zeroOutStandingBytesPeriod > 0.02 * n then
3 |   |   signal REAPPLACKLIM;
4 |   else
5 |   |   signal not REAPPLACKLIM;
6 |   end
7 |   zeroOutStandingBytesPeriod := 0;
8 end
```

Algorithm 8: Pseudo code Time Base ReAppLack solution

Check-based ReAppLack method

The check-based solution determines after m periods of zero outstanding bytes at the sender if the connection was ReAppLack limited since the beginning of those m periods. In order to determine this limitation, the period of zero outstanding bytes is compared to the total period of measurement since the beginning of the m periods. As the period between two checks by this solution are not fixed, the current time is stored in the variable `startMeasurement` by calling the method `currentTime` at the start of each check. Pseudo code for this algorithm is provided in Algorithm 9

```
   input: m; The number of measurements between two decisions
 1 numberOfMeasurements := 0;
 2 zeroOutStandingBytesPeriod := 0;
 3 startMeasurement := currentTime();
 4 for each zeroOutStandingBytes period do
 5 |   numberOfMeasurements++;
 6 |   zeroOutStandingBytesPeriod = zeroOutStandingBytesPeriod +
   |   measuredPeriod;
 7 |   if numberOfMeasurements == m then
 8 |   |   if zeroOutStandingBytesPeriod > 0.02 * (startMeasurement -
   |   |   currentTime()) then
 9 |   |   |   signal REAPPLACKLIM;
10 |   |   else
11 |   |   |   signal not REAPPLACKLIM;
12 |   |   end
13 |   |   startMeasurement := currentTime();
14 |   |   numberOfMeasurements := 0;
15 |   |   zeroOutStandingBytesPeriod : 0;
16 |   end
17 end
```

Algorithm 9: Pseudo code Time Based ReAppLack solution

The above two solutions determine whether a stream is ReAppLack limited if more than 2 percent of total time the sender had zero outstanding bytes. This threshold is copied from the work of Mark Timmer since this threshold, presented in his work, produced proper results.

The decision which method is used in the implementation of our tool is explained in Chapter 4.

Application layer acknowledgements or requests

The detection of this scenario – ReAppAck- is just based on the detection of acknowledge packets containing a payload. As the detection methods in [Tim05] only consider asymmetric TCP streams, acknowledgement packets containing a payload are a good indication of this scenario.

The detection method developed by Mark Timmer base its measurement only on the current packet received. The method counts the number of packet containing a payload and the number of packets without a payload. As the

method base their measurement on the current packet without to reprocess packets, this method can be used in realtime without any modifications.

As with the detection of the realtime receive window limitation, the major problem is in deciding wether a stream is limited over a part in its lifetime; the methods of Mark Timmer assume that a connection is limited by one limiting factor of its lifetime. In order to decide whether a connection is limited over a part of its lifetime, two solution are presented in the next to subsections. These two methods have similarities with the methods described in Subsection 3.1.

Time-based ReAppAck detection

The time-based ReAppAck detection methods works by keeping track of the number of packets with and without a payload over the last n seconds (This variable n is an input parameter to the solution). To keep track of these counters, they are stored in the variables *numberOfPacketsWithPayload* and *numberOfPacketsWithPayload*, respectively. Algorithm 10 shows pseudo code of the algorithm to keep track of these variables.

```
1 numberOfPacketsWithoutPayload := 0;
2 numberOfPacketsWithPayload := 0;
3 for each AppAck measurement performed do
4 |   if packet in measurement has a payload then
5 |   |   numberOfPacketsWithPayload++;
6 |   else
7 |   |   numberOfPacketsWithoutPayload++;
8 |   end
9 end
```

Algorithm 10: Pseudo code algorithm to keep track of `numberOfPacketsWithoutPayload` and `numberOfPacketsWithPayload`

After n seconds the decision is made whether or not this streams was ReAppAck limited or not. Pseudo code of this solution is provided by Algorithm 11. Both Algorithms perform their measurements in parallel and share the variables `numberOfPacketsWithoutPayload` and `numberOfPacketsWithPayload`.

Check-based ReAppAck detection

The check-based ReAppAck solution keeps track of the same variables `numberOfPacketsWithPayload` and `numberOfPacketsWithPayload` in order to decide whether a connection was ReAppAck limited over a period in its lifetime. The check-based ReAppAck solutions makes this decision after a certain number of packets measured. This number is the parameter m, provided as a parameter to the algorithm. The pseudo code of this algorithm is shown in Algorithm 12.

The above two solutions make the decision whether a stream is ReAppAck limited is made if more than 10 percent of the packets contain a payload. This threshold is copied from the work of Mark Timmer.

The decision which method is used in the implementation of our tool is explained in Chapter 4.

```
input: n; The number of seconds between two decisions
numberOfPacketsWithoutPayload := 0;
numberOfPacketsWithPayload := 0;
for every n seconds do
    if numberOfPacketsWithPayload > 0.1 *
    (numberOfPacketsWithPayload + numberOfPacketsWithoutPayload)
    then
        signal REAPPACKLIM;
    else
        signal not REAPPACKLIM;
    end
    numberOfPacketsWithPayload := 0;
    numberOfPacketsWithoutPayload := 0;
end
```

Algorithm 11: Pseudo code Time-based ReAppAck solution

```
input: m; The number of measurements between two decisions
numberOfPacketsWithoutPayload := 0;
numberOfPacketsWithPayload := 0;
for each ReAppAck measurement performed do
    if packet in measurement has a payload then
        numberOfPacketsWithPayload++;
    else
        numberOfPacketsWithoutPayload++;
    end
    if (numberOfPacketsWithPayload +
    numberOfPacketsWithoutPayload) == m then
        if numberOfPacketsWithPayload > 0.1 *
        (numberOfPacketsWithPayload +
        numberOfPacketsWithoutPayload) then
            signal REAPPACKLIM;
        else
            signal not REAPPACKLIM;
        end
        numberOfPacketsWithPayload := 0;
        numberOfPacketsWithoutPayload := 0;
    end
end
```

Algorithm 12: Pseudo code Time-based ReAppAck solution

Chapter 4

Tool implementation

The improvements made in the previous chapter makes it possible to develop a tool to visualize in realtime the limiting factors for the speed of TCP connections. This chapter first describes a framework in which these methods are implemented. In addition, the Sections 4.2, 4.3 and 4.4 describe the implementation of the receive window, network and application limiting methods, respectively.

4.1 Framework

A framework can be described as is defined in [TCL01]:

> *A framework is a software system especially designed to be reused in different projects, or in the various products of a product line.*

The word "reused" is the most important word of this quote as the framework provides functionality that is common to the different projects, or in the various products of a product. In our work we develop a framework which provides the common functionality used by all measurement methods. Furthermore, this framework eases the implementation of measurement methods.

Summarized, the basic goals of this framework is to provide a software system that:

- captures packets in order to detect and visualize speed limiting factors.
- eases the implementation of methods to detect limiting factors for the speed of TCP in the measurement and interpretation part.

In order to achieve these goals the following decisions had to be made during the design of the framework:

- **The framework is implemented using Java and Object Oriented Techniques**: The tool is not designed to run in a single environment with a single operation system. The tool should therefore support multiple operating systems. This work decided to use Java as it is a computer language capable to run on multiple operating systems as long as it contains a java virtual machine. In addition, the author of this work has some experiences using Java and Object Oriented Techniques due to multiple studied courses.

- **Only TCP packets are supported**: as the methods are developed to detect the limiting factors of the speed of TCP, this framework only provides functionality to capture and process TCP packets.
- **Capturing packets is performed at the sender side**: this requirement is adopted from Mark Timmer's methods by the methods developed in Chapter 3. Detecting a limiting factor is much more accurate when performed at the sender side [Tim05].
- **Measurements methods are implemented using a finite state machine**: the use of finite state machines (FSM) in the design of a measurement method allow to analyze its behavior. A tool like the labeled transition analyzer [lts06] allow to verify this behavior. In order to implement a FSM in the framework, the framework provides some facilities:
  - For each connection measured, a FSM is created as the FSM models the behavior of the connection. In order to process captured packet by the current state of the FSM this state is stored.
  - The states in a FSM produce data and this data may be required by other states of the same FSM. In order to make this data available to all states of the FSM, this data is stored outside the FSM.
- **Packets are processed in sequential order**: parallel processing of packets will make the design of the framework much more complex. For instance, a situation may occur where two packets, belonging to the same connection, are processed in parallel. While capturing packet A occurred before packet B, processing packet B may have finished before packet A. If the result of processing packet B depends on the result of processing packet A the framework needs to recover from such a situation.

The tool is composed of the framework in combination with the implementation of methods to detect the receive window, network and application limiting factors. The framework captures in realtime packets from a network. These packets are processed by each method before visualizing the speed limiting factors in realtime using a pie-chart. Figure 4.1 show the design of the framework as a "black box".

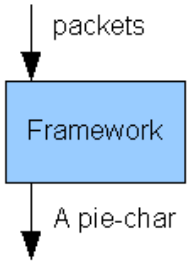

Figure 4.1: Framework designed as a black box

The internal design of the framework is specified in more detail using four components. Each component is described in more details in the next 4 sub-

sections. At the end of this section, the relationship amount the components is described.

Capture component

The main responsibility of this component is the sequential capturing of packets. These captured packets are a representation, in memory, of the packets flowing through a network. The captured packets are used to determine the speed limiting factors. In the implementation of this component, the library jpcap is used[jpc06]. Jpcap just provides in Java an interface to the C-library libpcap in order to passively capture packets. The decision to use jpcap is based on the widely use of libpcap by many programs like ethereal and tcpdump.

Measurement component

This component provides the facilities to implement methods to measure a limitation in the speed of a TCP stream. Therefore, this component is the most interesting component of the framework. This component is directly mapped to the a measurement part of a detection method (see Chapter 3). The first task of this component, on the input of a packet, is to determine to which connection the packet belongs to. This information is needed as each connection contains one instance for each measurement methods in the form of a finite state machine (FSM). Each instance of the FSM processes the packet in order to determine if the connection is limited by a speed limiting factor.

A FSM is described as a model of behavior. The model is composed of states, transitions, guards and actions which are explained using the example presented in Figure 4.2.

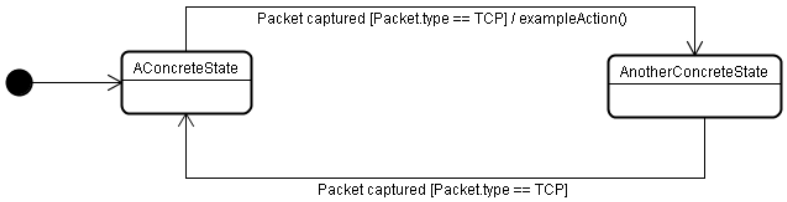

Figure 4.2: An example of a finite state machine

The FSM in Figure 4.2 represents a FSM that switches between the states `AConcreteState` and `AnotherConcreteState`. This transition is made on the reception of a packet with the condition (guard) that the packet is a TCP packet. In addition, the transaction from state `AConcreteState` to `AnotherConcrete-State` contains the action `exampleTransaction()`. This action is called when the transition is made. A trace of the behavior of the example FSM is captured in the sequence diagram presented in Figure 4.3.

This sequence diagram shows the reception of three packets by the FSM. Initially the FSM is in state `AConcreteState`, therefore this state processes the packet. The current state determines that the packet is a TCP packet and it

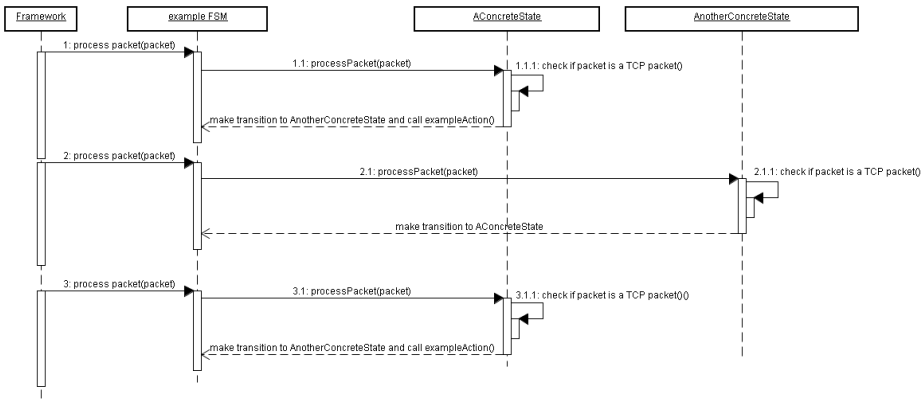

Figure 4.3: Sequence diagram State Pattern

forces the transition of the FSM to the `AnotherConcreteState`. On transition the `exampleAction` method is also called. The next packet is processed by the `AnotherConcreteState` as it is the current state. This state also determines that the packets is a TCP packet and a transition to `AConcreteState` is forced. The next packet is therefore again processed by the `AConcreteState` as described.

The actual FSM can be implemented in multiple ways in the framework. One way is to use a state transition table. This table describes in each column for each state the next state on reception of input and the option corresponding to the transition. Table 4.1 shows the state transition table for the example FSM.

	packet captured[packet.type == TCP]
AConcreteState	AnotherConcreteState/exampleAction()
AnotherConcreteState	AConcreteState

Table 4.1: State transition table

Another way to implement a FSM is the use of the state pattern. This pattern has been designed using object oriented features. Nowadays, object orientation is a popular technique to design an write software programs. As our framework is also designed and implemented using object orientation this pattern is used in the framework.

Figure 4.4 describes an UML diagram describing the structure of the state pattern. In the state pattern each FSM consists of 1 or multiple states. All these states will be concrete states derived from the abstract class `State`. In order to keep track of the current state each FSM has a reference to one `State` stored in the variable `currentState`.

Appendix B presents a more detailed description of the FSM pattern in our framework.

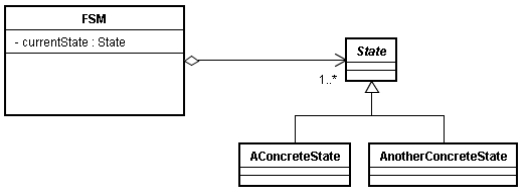

Figure 4.4: Class diagram state pattern

Interpretation component

This interpretation component is directly mapped on the interpretation part of a detection method (see Chapter 3). The responsibility of this component is to determine the limiting factor for each connection over a part in time. This component is implemented in the framework using the time-based solution. The time-based solution is preferable over the check-based solution as the time-based solution provides updates of speed limiting factors with a constant interval.

Visualization component

The visualization component is responsible for showing the limiting factors using a graphical interface. In order to show these limiting factors this component pulls the decision made by the interpretation component for each connection. This component then calculates the number of connections limited by a factor and divides this number by the total number of connections. The outcome of this calculation is the percentage of connections limited by a speed limiting factor. If the percentages for all measurement methods are calculate this component shows these percentages using a pie-chart.

Component relations

The relationship between the four components of a framework can be best described using a diagram. Figure 4.5 shows the revealed "black box" previously presented in Figure 4.1. The input and output data streams of Figure 4.5 stayed the same, but now the internal components and the data streams between these components is shown.

First, the capture components is provided with packets flowing through a network. The capture components captures these packets and send the captured packets to the measurement component. This component performs its detections and send any relevant information about the detection to the interpretation component. This component performs it tasks and sends its decisions to the visualization component when requested by the visualization component. The visualization component creates a pie-chart showing the distribution of the speed limiting factors. This pie-chart is outputted to the program's user interface.

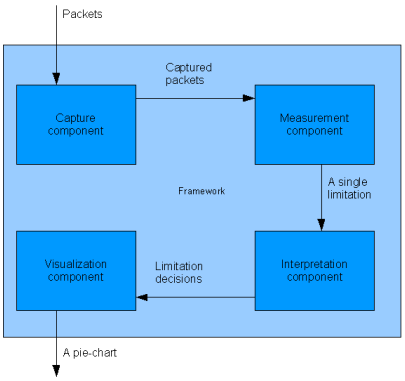

Figure 4.5: The relationship between the components in the framework

4.2 Implementing the realtime receive window limitation detection

The implementation of the realtime receive window limitation method is based on the design described in Chapter 3. This design is split up into two parts: the measurement and the interpretation part. The implementation of the measurement part is based on the improved method described in Section 3.1. The FSM of the measurement part is designed using a state diagram. This diagram is presented in Figure 4.6.

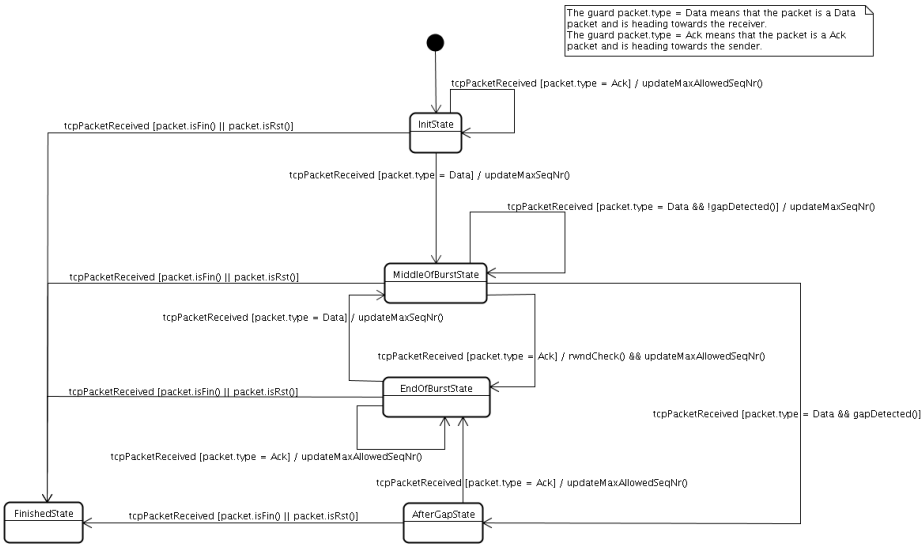

Figure 4.6: State diagram ReRwnd limitation detection

The state diagram consists of five states: `InitState`, `MiddleOfBurstState`,

`EndOfBurstState`, `AfterGapState` and the `FinishedState`. On initialization of the state diagram the initial state is the `InitState`. As long as an ACK packets are received the value `maxAllowedNextSeqNr` is updated. On reception of a DATA packet the `MiddleOfBurstState` is entered and the value `maxNextSeqNr` is updated. The purpose of the `MiddleOfBurstState` is to wait for the end of the burst in order to perform the limitation detection. On the reception of a DATA packet, the `MidleOfBurstState` checks whether this packet creates a gap in the stream of DATA packets. If this is true the `AfterGapState` is entered. If this check is false the value of `maxNextSeqNr` is updated. The reception of an ACK packet in the `MiddleOfBurstState` determines the end of a packet burst. A ReRwnd limitation check is performed, the value of `maxAllowedNextSeqNr` is updated and the `EndOfBurstState` is entered. The `EndOfBurstState` is also entered by the reception of an ACK packet in the `EndOfBurstState` as the method specifies that the limitation check can be continued after the reception of an ACK packets after a gap. In the `EndOfBurstState` the reception of an ACK packet and DATA packet will update the value `maxAllowedNextSeqNr` and `maxNextSeqNr` respectively. In addition, the DATA packet also causes a transition to the `MiddleOfBurstState` in order to indicate the start of a net burst of DATA packets. All states have in common that the reception of a packet with the fields FIN or RST set causes the transition to the `FinishedState`. The `FinishedState` is the end state and causes the `Connection` object to which it belongs to be deleted.

The states in the state diagram can be mapped directly to concrete State objects in the state pattern.

The interpretation part of this detection method is based on the time-based. The check-based solution depends on the data rate of a TCP stream and the number of limiting checks performed. In order to get a regular update in time the time-based solution is used.

4.3 Implementing the realtime network limitation detection

This section describes the behavior of the three FSMs, described in Section 3.2, in order to track the congestion window of the sender.
First, some initial variables are initialized. One of these values is the variable `icwnd`. This variable holds the number of bytes a sender can send after the handshake and before the arrival of the first acknowledge packet. This number of bytes is tracked in order to determine the value of `icwnd`. Another value set is the `ssthresh`. This variable contains the slow start threshold. On initialization this value is set to an extreme value, like 65536 bytes as this in practice done by most TCP implementations[Jai05].

A sender's TCP implementation contains at least two states: the slow start and the congestion avoidance state [Bra89]. The arrival of an acknowledgement packet changes value of the congestion window (`cwnd`, in bytes) depending a connection's current state. In the slow start phase, the `cwnd` variable is increase by "maximum segment size" (MSS) bytes and in the congestion avoidance state this variable is increased by $MSS \cdot \frac{MSS}{\texttt{cwnd}}$ on the reception of a non-duplicate

acknowledgement packet. The detection of a loss by a timeout decreases the variable `cwnd` to MSS bytes and `ssthresh` to $max(\frac{min(\texttt{awnd},\texttt{cwnd})}{2}, 2 \cdot MSS)$. The variable `awnd` is the last advertised receive window of the receiver. This behavior is common to all three TCP implementations whereas their behavior on receiving three duplicate acknowledgement packets is different.

The implementation of TCP Tahoe reacts on the reception of three duplicate acknowledgment packets the same as a timeout. TCP Reno adds fast recovery to the implementation of TCP Tahoe.

TCP Tahoe reacts on the reception of three duplicate acknowledgments packets the same as a timeout.

TCP Reno adds to Tahoe a technique called fast recovery [Ste97]. The fast recovery phase is entered by a TCP sender after three duplicate acknowledgment packet. On the reception of the fourth duplicate acknowledgment packet, the variable `ssthresh` is set to $max(\frac{min(awnd,cwnd)}{2}, 2 \cdot MSS)$ and `cwnd` to $sstresh + 3$. Thereafter, the variable `cwnd` is increased by `MSS` by every duplicate acknowledgment packet. On the reception of a new acknowledgment packet, the variable `cwnd` is set to `sstresh` and the congestion avoidance phase is entered.

TCP NewReno alters the behaviour of the recovery phase of TCP Reno. TCP NewReno checks in the fast recovery phase if the reception of a non-duplicate acknowledgment packet acknowledges all bytes before the fast recovery phase. In this is not the case, TCP NewReno stays in the fast recovery phase.

4.4 Implementing the realtime application limitation detection

The implementation of the realtime application limitation detection method is based on the method described in Chapter 3. As all methods, this method is also split in two parts: the measurement and the interpretation part. The implementation of the measurement part is based on the two methods described in Section 3.3: the method to detect lack of data scenario and the application layer acknowledgments or request scenario. The implementation of these two methods are described in the next two sections, respectively.

Lack of data

Figure 4.7 show the state diagram representing the FSM of the method to detect a lack of data to send by the sender.

The FSM contains, besides the usual start and end states, the two states `InitState` and `IdleState`. The `IdleState` represents the case where a TCP connection has multiple of oustanding bytes. The processing of a data packet or a acknowledge packets by this FSM causes an update of the value for `maxNextSeqNr` or `maxAllowedNextSeqNr` using the actions `updateMaxNextSeqNr` and `updateMaxAllowedNextSeqNr` respectively. The state `InitState` transits to the `IdleState` on the processing of an acknowledgment packet by this FSM if the number of outstanding bytes are zero, after an update of the value `maxAllowedNextSeqNr`. In addition, the time the packet was captured is stored for later use. In the `IdleState` is a transition performed on the reception of a data packet. In

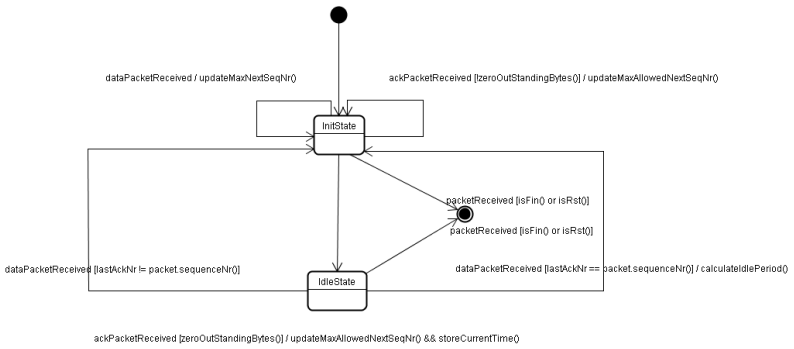

Figure 4.7: State diagram of the detection method to detect a lack of data

this case the last acknowledgment number equals the packet's sequence number the idle period is calculated by subtracting the capture time of the current and the stored time. This time is signalled to the interpretation part. The `IdleState` is not updating the value for `updateMaxNextSeqNr` on receiving an acknowledgment packet; the `IdleState` is only entered when all send bytes are acknowledged. Therefore, a new acknowledge packet is just a reordered, retransmitted or a copy of an acknowledge packet processed before and the value for `updateMaxNextSeqNr` is not changed.

In addition to the transitions described in the last paragraph, all states change to the end state of the FSM on processing a packet with the *RST* or *FIN* field set.

Application layer acknowledgments or requests

Figure 4.8 shows the state diagram used to describe the FSM of the method in order to detect application layer acknowledgments or requests in realtime. The purpose of this FSM is to detect the number of acknowledgment packets containg a payload. Besides the usual start and end state, this FSM only contains the state `InitState` to achieve this purpose.

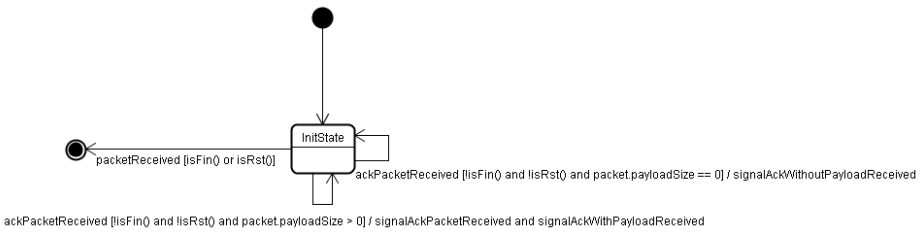

Figure 4.8: State diagram of the detection method to detect application layer acknowledgments or requests

On initialization of the FSM, the state of the FSM is immediately changed from the start state to the `InitState`. In the `InitState` three possible transi-

tions are possible. On the reception of each packet where its *RST* or *FIN* field is set, a transition is made to the FSM's end state. The remaining two transitions are performed from the `InitState` to the same `InitState` on the reception of an acknowledgment packet. During each transition the interpretation part is signaled for an limitation detection by the action `signalAckPacketReceived`. Only in the case of the reception of an acknowledgment packet with a payload the interpretation part is also signaled for the detection of a limitation by the action `signalAckPacketReceived`.

Chapter 5

Using the tool

This chapter is intended for users of the tool like network administrators in order to install and use the tool. In the following paragraphs of this chapter, the capabilities and limitations of the tool are discussed. In addition, this chapter describes how the tool needs to be installed and the graphical interface is discussed.

The tool is capable of visualizing the limiting factors for the speed of TCP connections in realtime. As an input, the tool uses packets flowing though a network and the tool outputs a pie-chart that shows the speed limiting factors of the TCP connections.

The tool is implemented in Java[Jav06] using the library Jpcap[jpc06], therefore, the tool requires both a java virtual machine and the library JPcap to be installed. For further information about how to install these components, their respective websites have to be visited.

As the framework and the methods to detect the speed limiting factors are designed to detect at the sender side, the tool should be installed near the senders in the network. In addition, in order to perform a good measurement of the speed limiting factors of a connection all packets of the packets need to be captured by the tool. In practice this means that the best position to install the tool is on the network gateway or on a computer capable of capturing packets flowing through the network's gateway. Figure 5.1 shows an example of a network topology in which the tool is installed at the sender side. In this example the data flowing through the gateway is copied to the tool in order for it to capture these data.

If the tool is installed it can be started using the command line. By typing the command:

```
java -cp ../lib/jcommon-1.0.0.jar:../lib/jfreechart-1.0.1.jar:
../net.sourceforge.jpcap-0.01.16.jar:.
nl.utwente.ewi.dacs.analyzerRT.Analyzer -D etho -N network-
prefix
```

in the src directory of tool's directory.

The parameter `cp` sets the classpath of the java virtual machine to the external libraries used, separated by a colon. The parameter `D` parses the name of the network device on which packets are captured by the tool. In our example

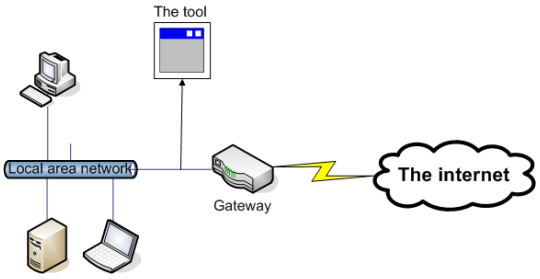

Figure 5.1: Example of a topology in which the tool is installed

the name of this interface is etho. This name is used in Linux to indicate the first Ethernet device. Last, the parameter `N` parses the network prefix. This is the network part of an IP address. It is used to determine if the sender or sender of a packet is in the "sender" network. This parameter is calculated by performing a bitwise AND operation on an IP address of the "sender" network and the netmask of the network.

After a correct start of the tool a graphical user interface is presented. Figure 5.2 shows this interface. The interface consists of three components: a pie-chart, a legend and a slider.

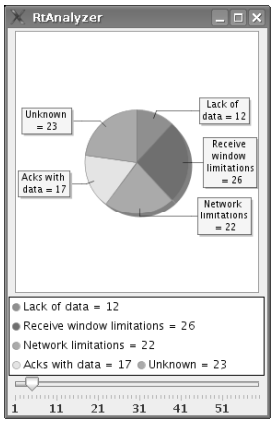

Figure 5.2: The graphical user interface of the tool

As figure 5.2 is showing, both the pie-chart and the legend show the percentage of limited connections for each limiting factor implemented in the tool. For example, the pie-chart and the legend show that 12 percent of the connections are limited by a lack of data. In addition, the pie-chart and legend show for what percentage of connections it was unable to detect the limiting factor.

The slider is used to set the update interval of the pie-chart and agenda in seconds. This interval can be set a value between 1 and 59 seconds. Default this update interval is set to 5 seconds.

Evaluation of the tool

The evaluation that is performed by this work is speed op analyzing TCP streams. On of the major limitaitons, found in this evaluation, is the capturing speed of the tool. On a regular PC with an AMD Athlon 800 Mhz processor and 512 MB of internal memory, the tool can process no more than 10 Mbits per second of data.

Evaluation that has not been performed, due to a lack of time, is the memory and CPU usage of the tool. In order to measure these usages, an external tool has to be used.

Another part of the evaluation was analyzing if the tool measures the same limitations as the tool of Mark Timmer. Since the tool is not detecting sender buffer limitations, the tool may show incomplete information. In order to compare the results of this tool and Mark Timmer's, both tools should perform their measurements based on the same TCP streams. In addition, if the tool of this work performs its interpretation algorithms only after each stream, both tools should show the same limiting factors.

Chapter 6

Conclusions

This work has presented two contributions: the development of realtime methods to identify the speed limiting factors of TCP streams and the implementation of a graphical tool that visualizes these speed limiting factors. This work started by presenting the methods developed by Mark Timmer for detecting the speed limiting factors of TCP streams. In his work, Mark Timmer describes methods to off-line identify the most dominant speed limiting factor over a complete TCP connection. However, these methods do have two major drawbacks:

- The methods cannot determine in realtime the limiting factors of a TCP stream. For network administrators it is interesting to see in realtime the limiting factors for the speed of TCP connections flowing through their network. With this information, network administrators can respond immediately to changes in the distribution of the limiting factors.
- Mark Timmer assumes that a TCP connection is limited by one factor during its lifetime. This assumption may not be true for connections with a larger lifetime. A change in the path between the end hosts of a TCP connection or a change of the size of a buffer in the end host are two examples which can change the speed limiting factors of a TCP stream.

Since Mark Timmer's methods do not enable realtime analysis, this work has addressed the limitations found in his work by improving most of his methods in order to perform realtime analysis.

The realtime methods presented in this work have been described by a common structure; all these methods consists of a measurement and interpretation part.

The measurement part is responsible to detect a single limitation. For each limiting factor under consideration a detection method is implemented in the measurement part. The criteria developed by Mark Timmer in order to detect a receive window and application limitation needed no adaption in order to perform realtime measurements; these criteria measure the limitations based on the current captured packet without the need to reprocess stored packets. To measure a network limitation this work described an adaptation of [JID+02]. This adaptation drops the need to store all captured packets for reprocessing. This method improves the way the network limitation is detected as the congestion window is directly tracked.

Based on the detections of a measurement part, the interpretation part decides whether a connection was limited by a factor. This work has proposed a new method to decided whether a connection was limited over a part of its lifetime as a connection may be limited by multiple factors over its lifetime. The method has improved the method of Mark Timmer as his method makes the decision over the entire lifetime of a connection.

This work has implemented these realtime methods to detect the limiting factor for the speed of TCP in a tool. The tool is capable of visualizing in realtime the speed limiting factors of TCP connections captured by the tool. This tool has improved visualizing the speed limiting factors of TCP connections compared to the tool of Mark Timmer as the tool presented visualizes these limiting factors in realtime. In addition, the tool presents the limiting factors using a pie-chart. This presentation makes is easier for users of the tool to receive a quick indication of the distribution of the limiting factors.

One of the objectives of this work was to improve the methods of Mark Timmer in order to detect the limiting factors for the speed of TCP. The methods of Mark Timmer to detect the receive window and application limitations have been improved in order to perform realtime limitations. This work has not improved the method of Mark Timmer to measure a network limitation as we have adapted the method described in [Jai05] to detect the network limiting factor in realtime. The method to detect a sender buffer limitation was also not improved as a decrease of the buffer size is hard to detect since this buffer size is never advertised.

The other objective of this work was to design and implement a tool in order to visualize in realtime the limiting factors for the speed of TCP. This objective has been met by the implementation of the receive window, network and application methods in a framework which is developed by this work. The framework facilitates the capturing of packets and visualizing the limiting factors.

However, the implemented tool presents some limitations. One limitations is concerned with the amount of packets captured per second. The framework makes use of a library called Jpcap. This library is limited in the number of packets it can capture. This limit is around 10 Mbits per second on a AMD Athlon 800 Mhz. P.C. with 512 MB. of internal memory running on Windows XP. In addition, the tool is only capable of showing one pie-chart. A problem occurs if the tool is extended with a method unrelated to the limiting factor methods. If, for instance, a method is added to measure the number of packets of a TCP connection, the pie-chart is then both showing the number of packets and the limiting factors making the pie-chart meaningless. Another problem of the tool is that it is incapable of detecting the sender buffer limiting factor. As this work has not improved this method, this method was not implemented in the tool. In the case a TCP connection is limited by the sender buffer limiting factor the tool is not showing this limitation. Instead, the tool may indicate that the connection is limited by the network or it indicates the limitation is unknown.

As future work more research needs to be performed in order to find a solution for these limitations and problems. The major bottleneck of our tool is the interface between Java and the library libpcap, provided by the library Jpcap. As libpcap is written in C, one could research the development of the tool in C or C++. In addition, research can be performed to make the framework capable of showing multiple charts. This feature makes the framework more useful as

multiple unrelated measurement methods may be implemented into it. Further research can also be performed in the area of detecting the sender buffer limitation in realtime. Last but not least, the percentages used in the interpretation part are copied from Mark Timmer's work. The realtime detection of limiting factors may take different percentages, compared to its offline equivalent.

Appendix A

Structure framework

This appendix describes the structure of the framework using an UML class diagram. The class diagram is presented by Figure A.1. This UML diagram is not complete; some methods are not showed as they are not important in describing the structure of the framework.

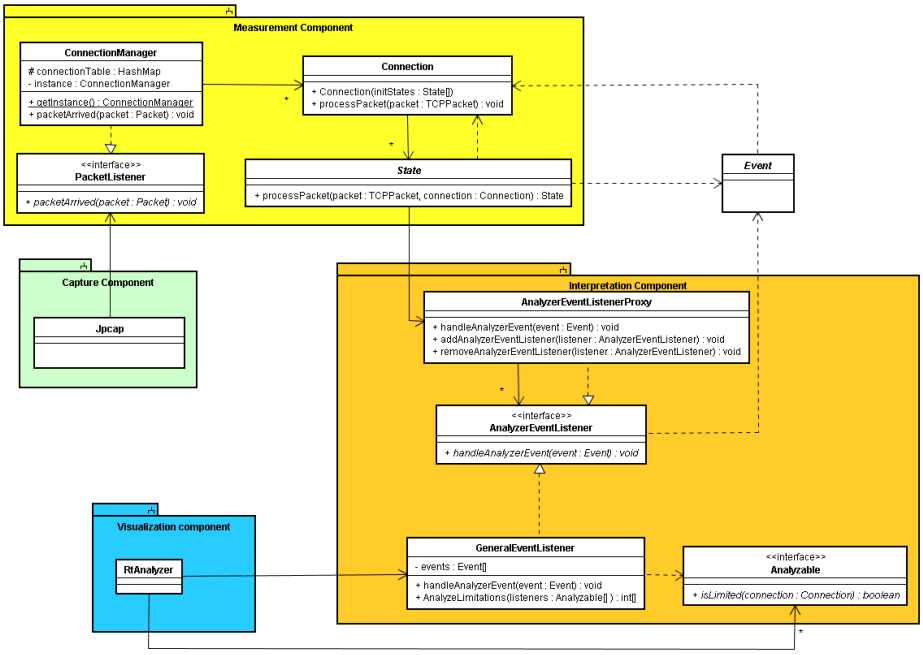

Figure A.1: Class diagram framework

The UML class diagram is subdivided in four components described in Section 4.1; the components are represented by a subsystem. Each subsystem is represented in a different color. The capture, measurement, interpretation and capture components have the color green, yellow, orange and blue, respectively. Each class and part of a component is shown inside the component it belongs

to.

The next 4 sections explain per component their classes and the purpose of these classes. The fifth section explains how detection methods should be implemented into the structure of the framework. Finally, the last section describes the behavior of the framework in order to understand the structure better.

Capture Component

The capture component consists of the class `Jpcap`. This class is just a representation of the Jpcap library; it is not actually implemented in our framework. The library captures packets from a network. For each packet captured the PacketListener's method `packetArrived(packet : Packet)` is called. A `Packet` object is a representation of packet captured. This object is not shown in the UML class diagram as it is part of the Jpcap library.

Measurement Component

One of the interfaces part of this component is the `PacketListener` interface. This interface is implemented by the class `ConnectionManager`. The `ConnectionManager` is receiving the packets captured by the Capture Component. The purpose of the `ConnectionManager` is to determine to which `Connection` the `Packet` is part of. If the `Packet` is the first packet of a connection, the `ConnectionManager` creates a new `Connection` instance. The `Connection` class is a presentation of a TCP connection. The `ConnectionManager` calls the method `processPacket(packet:Packet)` of the `Connection` object the packet is part of. The `Connection` class in combination with `State` classes form the state pattern. This pattern is described in detail in Appendix B. In order to signal the Interpretation Component, in for instance, the detection of a limitation, a `State` object creates `Event` objects. Every `Event` object has a reference to a `Connection` object as the source of the `Event` object as this source is used in the Interpretation Component. This reference is passed as a parameter during the instantiation of the `Event` object. These `Event` objects are send by the framework to the Interpretation Component. Every implementation of a measurement method should define its own `Event` objects by extending `Event` objects.

Interpretation Component

The `Event` objects, send to the Interpretation Component, are processed by the `AnalyzerEventListenerProxy` class. This class keeps references to multiple implementations of the interface `AnalyzerEventListener` in order to proxy the received `Event` objects to them. One of these references is a reference to the `GeneralEventListener` object. This object is responsible of keeping track of all `Connection` references send via the source of an `Event` object. These references are needed as the `GeneralEventListener` needs to determine by what factors each `Connection` is limited (when its `AnalyzeLimitations(analyzers : Analyzable[])` method is called by the visualization component). The parameter analyzers is an array containing references to implementations of the

`Analyzable` interface. Implementations of the `Analyzable` interface should implement the method `isLimited(connection : Connection)`.

Visualization component

The visualization component contains just the `RtAnalyzer` class. This class is responsible of showing the graphical user interface (GUI). In addition, it is the main class of the tool. Therefore, it initializes the framework and starts the capturing in order to measure the speed limiting factors.

In order for the `RtAnalyzer` class to show a pie-chart in the GUI, it needs to known what percentage of the total connections are speed limited by what factor. To retrieve this information is, it calls the method `AnalyzeLimitations` with parameter an Array containing implementations of the Analyzable interface.

Implementing detection methods into the framework

In order to implement a detection methods in the framework, multiple elements have to be implemented: the measurement and interpretation part of a detection method (See Chapter 3) and the events of a detection method.

The measurement part is implemented using the state pattern. The details of this pattern and how it is implemented into the framework is described in Appendix B.

The interpretation part and the events are implemented as is described by Figure A.2 using a UML class diagram.

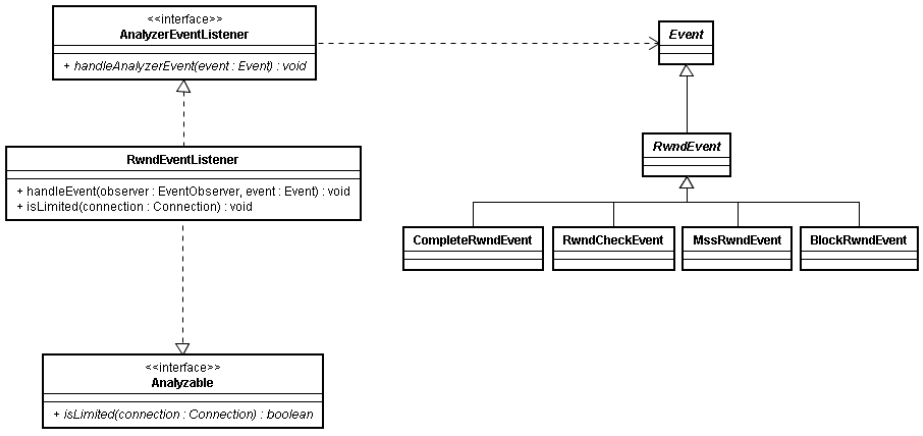

Figure A.2: Class diagram interpretation part and Events

The UML class diagram shows the implementation of the interpretation part of the realtime receive window detection method. The class `RwndEventListener` implements both interfaces `Analyzable` and `AnalyzerEventListener`. On instantiation of the class `RwndEventListener` this object is registered to the

`AnalyzerEventListenerProxy` in order to receive the `Event` objects produced by the measurement part. In order for the measurement part to signal multiple events, multiple `RwndEvent`s are implemented. For instance, the implementation of the `CompleteRwndEvent` corresponds to the detection of a complete receive window detection as is specified in [Tim05].

Appendix B

Finite state machine

This appendix describes in more detail the usage of a finite state machine (FSM) in the framework. The examples presented in this appendix are the more detailed version of the ones described in Section 4.1.

The FSM in the framework is implemented using a state pattern. The current state of a FSM is stored by the framework in order to let this state process a captured packet. Based on the processed packet, the current state can decide to change the current state of the FSM or, for instance, perform a measurement.

Figure B.1 shows an UML diagram where the class `Connection` and `State` form the state pattern used in the framework. The pattern starts with a Connection class.

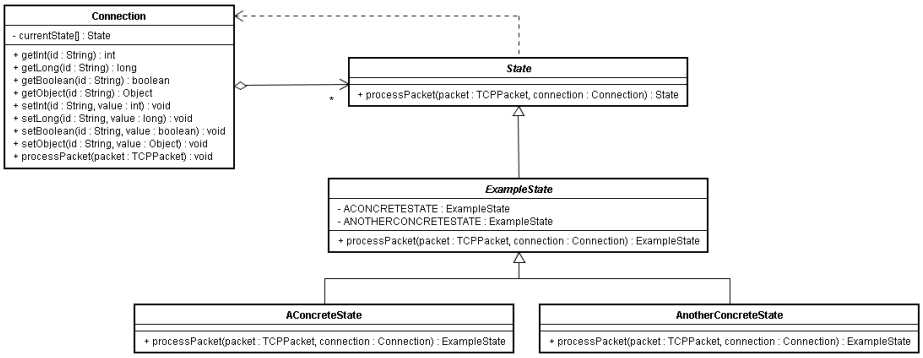

Figure B.1: Class diagram state pattern

The `Connection` class represents a connection and is instantiated for every new connection measured. One of the facilities that the framework offers is the storing of the current state of each FSM for each connection. This facility is provided by the attribute `currentState`. This attribute stores in every `Connection` object the current `State`s for each FSM implemented. These current `State`s are used by the method `processPacket(packet : TCPPacket)` of a `Connection` Object. This method is called every time a packet is captured belonging to the connection. The parameter `TCPPacket` represents the captured packet. The responsibility of this method is to call the method `processPacket(connection`

`: Connection, packet : TCPPacket)` of each `State` stored in the attribute `currentState` and store the returned `State` as the new current `State`. The parameter `TCPPacket` is again the captured packet and the parameter `Connection` is the connection the `TCPPacket` belongs to. The facility to store data outside a FSM is also provided by the `Connection` class. The methods like `getInt(id : String)` and `setObject(id : String, value : Object)` are two examples to retrieve an integer and to save an object represented by the key `id` respectively.

The state pattern further consists of an abstract `State` class. This is a super class for all `State`s in all FSMs. In order to process the captured packets the method `processPacket(connection:Connection, packet:TCPPacket):State` should be overridden by all implemented `State`s of a FSM. The return value of this method should be the next current state of the FSM. Figure B.1 shows the design of an example FSM implementation. The `State` class is extended by the abstract `ExampleState` class. The only purpose of this class is to keep references to the concrete `State` implementations of this FSM: this class has two attributes `ACONCRETESTATE` and `ANOTHERCONCRETESTATE`. Concrete `State`s can use these references as return value of the method `processPacket(connection:Connection, packet:TCPPacket):State` in order to change the current `State` of the FSM. The code in Appendix C shows in further detail how this is implemented and used. The two concrete `State`s `AConcreteState` and `AnotherConcreteState` perform the actual processing of TCP packets. The behavior of the states in the example is to switch between the two `State`s in each reception of a TCP packet (see Figure B.2).

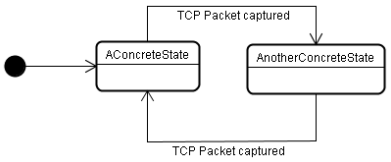

Figure B.2: An example of a finite state machine

To make this behavior more clear an example of a sequence diagram is represented by Figure B.3.

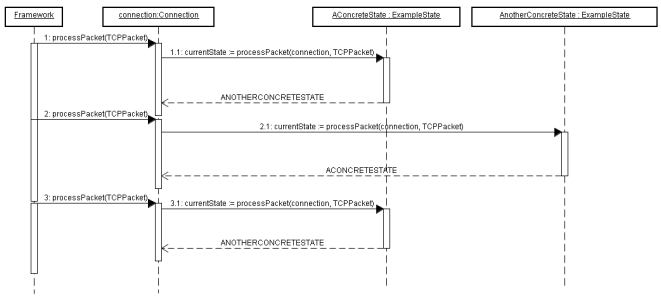

Figure B.3: Sequence diagram State Pattern

The object `Framework` is a virtual object representing the capturing of packets by the framework. On the capturing of the first TCP packet the method `processPacket` of the Connection object is called. The initial `currentState` of the ExampleFSM is set to the `State ACONCRETESTATE` therefore the method `processPacket` of `ACONCRETESTATE` is called. This method returns a reference to the `State ANOTHERCONCRETESTATE`. In the `Connection` object the `currentState` is therefore set to `ANOTHERCONCRETESTATE`. When method `processPacket` of the Connection object is called for the second time the method `processPacket` of `State ANOTHERCONCRETESTATE` is now called. This method returns a reference to the `State ACONCRETESTATE` and the current `State` is set to this reference.

Appendix C

FSM sourcecode

This appendix provides parts of source code of the implmentation of the ReRwnd FSM in the framework. Parts of the source uninterested for showing the implementation of the FSM are replaced by "...".

Connection.java

```
public class Connection extends TimerTask {
  ...

  public Connection(String id, State[] initStates) {
    this.id = id;
    this.initStates = initStates;
    ...
  }

  public void processPacket(TCPPacket packet) {

    ...

    for(int i=0; i<initStates.length; i++) {
      initStates[i] = initStates[i].processPacket(packet, this);
    }
  }

  ...

}
```

State.java

```
public abstract class State {
```

```
  protected AnalyzerEventListener eventListener;

  protected State() {
    eventListener = AnalyzerEventListenerProxy.getInstance();
  }

  public State processPacket(TCPPacket packet, Connection connection) {
    if(connection.isData(packet)) {
      connection.setSizePreviousDataPacket(packet.getPayloadDataLength());
    }

    return this;
  }
}
```

RwndState.java

```
public abstract class RwndState extends State {

  public static RwndState STARTUPFASESTATE = StartupState.getInstance();
  public static RwndState ENDOFBURSTSTATE = EndOfBurstState.getInstance();
  public static RwndState MIDDLEOFBURSTSTATE = MiddleOfBurstState.getInstance();
  public static RwndState AFTERGAPSTATE = AfterGapState.getInstance();
  public static RwndState FINISHEDSTATE = FinishedState.getInstance();

  public State processPacket(TCPPacket packet, Connection connection) {
    State state = super.processPacket(packet, connection);

    if(state != this) {
      // super method changed the state
    }

    ...

    return this;
  }

  ...
}
```

StartupState.java

```
public class StartupState extends RwndState {

  protected static StartupState instance;

  public static StartupState getInstance() {
    if(instance == null) {
      instance = new StartupState();
    }
    return instance;
  }

  public State processPacket(TCPPacket packet, Connection connection) {
    State state = super.processPacket(packet, connection);

    if(state != this) {
      // super method changed the state
    }

    if(packet.isFin() || packet.isRst()) {
      return processEndOfFlow(packet, connection);
    } else if(connection.isAck(packet)) {
      return processAckReception(packet, connection);
    } else if(connection.isData(packet)) {
      return processDataReception(packet, connection);
    } else {
      return this;
    }
  }

  protected State processAckReception(TCPPacket packet, Connection connection) {
    updateMaxAllowedSeqNr(packet, connection);

    return this;
  }

  protected State processDataReception(TCPPacket packet, Connection connection) {
    updateMaxSeqNr(packet, connection);

    return MIDDLEOFBURSTSTATE;
  }

}
```

AfterGapState.java

```
public class AfterGapState extends RwndState {

  protected static AfterGapState instance;

  public static AfterGapState getInstance() {
    if(instance == null) {
      instance = new AfterGapState();
    }
    return instance;
  }

  public State processPacket(TCPPacket packet, Connection connection) {
    State state = super.processPacket(packet, connection);

    if(state != this) {
      // super method changed the state
    }

    if(packet.isFin() || packet.isRst()) {
      return processEndOfFlow(packet, connection);
    } else if(connection.isAck(packet)) {
      updateMaxAllowedSeqNr(packet, connection);

      return AFTERGAPSTATE;
    }

    return this;
  }
}
```

EndOfBurstState.java

```
public class EndOfBurstState extends RwndState {

  protected static EndOfBurstState instance;

  public static EndOfBurstState getInstance() {
    if(instance == null) {
      instance = new EndOfBurstState();
    }
    return instance;
  }
```

```
  public State processPacket(TCPPacket packet, Connection connection) {
    State state = super.processPacket(packet, connection);

    if(state != this) {
      // super method changed the state
    }

    if(packet.isFin() || packet.isRst()) {
      return processEndOfFlow(packet, connection);
    } else if(connection.isAck(packet)) {
      updateMaxAllowedSeqNr(packet, connection);

      return this;
    } else if(connection.isData(packet)) {
      updateMaxSeqNr(packet, connection);

      return MIDDLEOFBURSTSTATE;
    } else {
      return this;
    }
  }
}
```

FinishedState.java

```
public class FinishedState extends RwndState {

  protected static FinishedState instance;
  protected Timer timer;

  public static FinishedState getInstance() {
    if(instance == null) {
      instance = new FinishedState();
    }
    return instance;
  }

  public State processPacket(TCPPacket packet, Connection connection) {
    State state = super.processPacket(packet, connection);

    if(state != this) {
      // super method changed the state
    }

    if(!connection.getScheduledForFinish()) {
      ...
```

```
      connection.setScheduledForFinish(true);
    }

    return this;
  }
}
```

MiddleOfBurstState.java

```
public class MiddleOfBurstState extends RwndState {

  protected static MiddleOfBurstState instance;

  public static MiddleOfBurstState getInstance() {
    if(instance == null) {
      instance = new MiddleOfBurstState();
    }
    return instance;
  }

  public State processPacket(TCPPacket packet, Connection connection) {
    State state = super.processPacket(packet, connection);

    if(state != this) {
      // super method changed the state
    }

    if(packet.isFin() || packet.isRst()) {
      return processEndOfFlow(packet, connection);
    } else if(connection.isAck(packet)) {
      return processAckReception(packet, connection);
    } else if(connection.isData(packet) && !connection.isGap(packet)) {
      return processNormalDataReception(packet, connection);
    } else if(connection.isData(packet) && connection.isGap(packet)) {
      return processGapDataReception(packet, connection);
    } else {
      return this;
    }
  }

  protected State processAckReception(TCPPacket packet, Connection connection) {
    if(connection.isMeasuringAtSendingSide()) {
      rwndCheck(connection);
    }

    updateMaxAllowedSeqNr(packet, connection);
```

```
    return ENDOFBURSTSTATE;
  }

  protected State processNormalDataReception(TCPPacket packet, Connection connection) {
    updateMaxSeqNr(packet, connection);

    return this;
  }

  protected State processGapDataReception(TCPPacket packet, Connection connection) {
    return AFTERGAPSTATE;
  }

  protected void rwndCheck(Connection connection) {
    eventListener.handleAnalyzerEvent(new RwndCheckEvent(connection));

    checkCompleteRwndUtilization(connection);
    checkIntegerPacketRwndUtilization(connection);
    checkBlockBasedRwndUtilization(connection);
  }

  protected void checkCompleteRwndUtilization(Connection connection) {
    ...
  }

  protected void checkIntegerPacketRwndUtilization(Connection connection) {
    ...
  }

  protected void checkBlockBasedRwndUtilization(Connection connection) {
    ...
}
```

List of Figures

List of abbreviations

TCP	Transport Control Protocol
IP	Internet Protocol
UML	Unified Modeling Language
ACK	Acknowledgment
ReRwnd	Realtime Receive Window
ReNwLim	Realtime Network Limitation
ReAppLack	Realtime Application Lack of Data
ReAppAck	Realtime Application Acknowledgments or Requests
FSM	Finit State Machine
MSS	Maximum Segment Size

Bibliography

[Bra89] R. Braden. Requirements for internet hosts – communication layers. Request for Comments: 1122, October 1989. IETF http://www.ietf.org/rfc/rfc793.txt.

[Jai05] Sharad Jaiswal. *Measurements-in-the-middle: inferring end-end path properties anc characteristics of TCP connections through passive measurments.* ph.d. thesis, University of Massachusetts Amherst, Amherst, MA, USA, sep 2005.

[Jav06] Java; a device-independent programming language, 2006. http://www.java.com.

[JID+02] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. Measurement and classification of out-of-sequence packets in a tier-1 ip backbone. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 113–114, New York, NY, USA, 2002. ACM Press.

[jpc06] Jpcap, a network packet capture library for applications written in java, 2006. http://jpcap.sourceforge.net.

[Kla05] Ruud Klaver. *Experimental Validation of the TCP-Friendly Formula.* B.Sc. thesis, DACS chair, University of Twente, Enschede, NL, apr 2005.

[LM97] T. V. Lakshman and Upamanyu Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.*, 5(3):336–350, 1997.

[lts06] Ltsa - labelled transition system analyser, 2006. http://www.doc.ic.ac.uk/ltsa/.

[PFTK00] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling TCP reno performance: a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2):133–145, 2000.

[PJD04] Ravi S. Prasad, Manish Jain, and Constantinos Dovrolis. On the effectiveness of delay-based congestion avoidance. In *In Proceedings of 2nd International Workshop on Protocols for Fast Long-Distance Networks*, 2004.

[Pos81] J. Postel. Transmission Control Protocol. Request for Comments: 793, September 1981. IETF http://www.ietf.org/rfc/rfc793.txt.

[Ste97] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Request for Comments: 2001, January 1997. IETF.

[TCL01] Robert Laganiere Timothy C. Lethbridge. *Object-Oriented Software Engineering*. McGraw-Hill Education, Berkshire, 2001.

[TdBP06] Mark Timmer, Pieter-Tjerk de Boer, and Aiko Pras. How to identify the speed limiting factor of a TCP flow. In *Fourth IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMON'06)*, apr 2006.

[Tim05] Mark Timmer. *How to identify the speed limiting factor of a TCP flow*. B.Sc. thesis, DACS chair, University of Twente, Enschede, NL, sep 2005.

[ZBPS02] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 309–322, New York, NY, USA, 2002. ACM Press.