Implementation and Evaluation of NetInf TP, an Information-centric Transport Protocol

Noman Mumtaz Ali  Róbert Attila Potys
nmali@kth.se     potys@kth.se

Master of Science Thesis
April, 2013

Communication Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden
TRITA-ICT-2013:64

*Examiner:* Björn Knutsson (KTH)

*Academic Supervisor:* Flutra Osmani (KTH)

*Industrial Supervisor:* Björn Grönvall and Ian Marsh (SICS)

©Noman Muntaz Ali and Róbert Attila Potys, April 2013
Abstract

In recent times, there has been a significant growth in the number of Internet users resulting in an increased demand for varying types and amounts of content. As content distribution over the Internet has become a key issue, one proposal is that the Internet architecture could evolve to a more “Information-Centric” paradigm instead of the current “Host-Centric” paradigm. In the host-based architecture, the data is often restricted to a location and will become unavailable if the host holding the data (or network connection) becomes unreachable. Furthermore, the data is always hosted at the location of the server, potentially far from the requestor.

With the Information-centric data approach, the requestor requests data and receives it regardless of where it is actually hosted. Hence, the focus moves from “where” to “what” is of interest. The heterogeneity of access methods and devices makes this type of approach even more appealing, especially when data can be cached in networked elements.

The prototype developed within this thesis builds an important part of the Information-Centric vision, that is a receiver-driven transport protocol. It is in stark contrast to host-centric transport protocols, which are predominately source driven. The advantage of having the receiver driven feature caters for multiple senders or receivers of the same data. That is, one receiver may ask more than one holder for different pieces of the same file.

We have implemented, simulated and assessed the performance of the proposed protocol, hereby called NetInf TP. Since the protocol may have to co-exist with existing sender driven TCP implementations, we have looked at the inter-operation of NetInf TP with TCP variants from both qualitative and quantitative perspectives.
Acknowledgements

We would first like to thank our former SICS supervisor Björn Grönvall for giving us the opportunity to work on the Master’s thesis project in the Communication Networks and Systems Lab, SICS. He had been a great source of inspiration and motivation for us. All the support, guidance and input that he gave throughout the thesis and after is highly appreciable. Also, the way he took us forward and the confidence that he had in us to propose our ideas and suggestions was remarkable. Moreover, that he was always available to help us in various implementation parts was indeed very kind.

Further, we would like to thank our current SICS supervisor, Ian Marsh who has been always available to help with our thesis specially in our Evaluation phase where we had long discussions with him and stayed up till late evenings to figure out the issues. He has always been very friendly yet professional and kind. It was a pleasure working under his supervision.

We would like to express special gratitude to our KTH supervisor, Flutra Osmani who despite of her busy schedule, was always there for us in figuring out issues in our report and guiding how to correct the specific sections. Also, special thanks to our KTH examiner Björn Knutsson for giving his feedback with time and guiding.

Last but not the least, we would like to appreciate the researchers in SICS and our lab leader Bengt Ahlgren for necessary guidance and great support, they were always aware of our progress and wanted to help with whatever they can.
I would like to dedicate this thesis to my Father, who was really looking forward to seeing our work done. Thank you for the support you gave me all my life, this work is dedicated to the memory of you.

Robert

Thanks to my family specially my dad who supported me throughout the entire period of my graduation.

Noman
# Contents

## I Background

1 Introduction 2

1.1 Thesis outline 3

1.2 Problem statement 4

1.3 Research question 5

1.4 Thesis Contribution 5

## 2 Literature Review 7

2.1 Background Literature 7

2.1.1 Information-Centric Networking 7

2.1.2 Network of Information (NetInf) 10

2.2 Related Work 12

2.2.1 Simulation-based comparisons of Tahoe, Reno and SACK TCP 13

2.2.2 Using OMNeT++ to Simulate TCP 13

2.2.3 CUBIC - A New TCP-Friendly High Speed TCP Variant 14

2.3 Other Work 14

2.3.1 ICP - Design and Evaluation of an Interest Control Protocol for CCN 15

2.3.2 Flow-aware Traffic Control a Content-Centric Network 16

## 3 Transport Protocol Overview 18

3.1 Transmission Control Protocol (TCP) 19

3.1.1 TCP’s Congestion Control 19
3.1.2 Fast Retransmit Algorithm .................................. 21
3.1.3 Fast Recovery Algorithm ................................. 21
3.2 TCP Implementations ....................................... 22
  3.2.1 TCP Reno .................................................. 22
  3.2.2 TCP New-Reno ............................................ 23
  3.2.3 TCP SACK .................................................. 24
  3.2.4 TCP Cubic ................................................ 25
3.3 TCP Friendliness ........................................... 26
  3.3.1 TCP Friendly Factor ................................. 26
  3.3.2 Fairness Metric ..................................... 27

4 Environment ................................................. 29
  4.1 OMNeT++ .................................................. 29
    4.1.1 Modelling Concepts ................................. 29
    4.1.2 Messages and packets ................................ 30
    4.1.3 Discrete event simulation ........................... 31
    4.1.4 Simulator and Analysing tool ..................... 31
  4.2 INET Framework ....................................... 32

II Design and Implementation .................................. 33

5 Transport Protocol Design .................................. 34
  5.1 NetInf Transport Protocol ............................. 34
    5.1.1 Objectives ........................................... 34
    5.1.2 Operation ............................................ 35
    5.1.3 Network concept ...................................... 36
    5.1.4 Transport setup ...................................... 37
    5.1.5 Data transfer ......................................... 38
    5.1.6 NetInf TP messages .................................. 39
    5.1.7 Congestion Control ................................. 43
    5.1.8 Retransmissions ...................................... 44
    5.1.9 Name-data integrity check ......................... 45
5.1.10 Security Considerations ................................................. 45
5.2 NetInf TP and existing TCP implementations ......................... 47
  5.2.1 Source and Receiver driven features ............................... 47
  5.2.2 Messages between source and receiver ............................ 48
  5.2.3 Fast recovery and Retransmission techniques ..................... 49
  5.2.4 Loss detection mechanism .......................................... 50
  5.2.5 Summarized table .................................................. 50

6 Transport protocol Implementation ........................................ 51
  6.1 Prototype overview ................................................... 51
  6.2 Applications .......................................................... 53
  6.3 Message definitions .................................................. 59
  6.4 Listing Algorithms ................................................... 63
    6.4.1 Request List .................................................... 63
    6.4.2 Received List ................................................... 65
  6.5 Transport Protocol Algorithms .................................... 68
    6.5.1 Principal terms and variables ................................ 68
    6.5.2 Updating the outstanding data ................................ 69
    6.5.3 Requesting process ............................................. 70
    6.5.4 Congestion control .............................................. 71
    6.5.5 RTT estimate .................................................... 73
    6.5.6 Scheduling timers .............................................. 74
    6.5.7 Detecting packet loss .......................................... 75
    6.5.8 Segmentation at the sender’s side .............................. 77

7 Evaluation ................................................................. 79
  7.1 Single Flow Performance .............................................. 79
    7.1.1 Method .......................................................... 80
    7.1.2 Topology ........................................................ 80
    7.1.3 Results .......................................................... 82
    7.1.4 Summary ........................................................ 85
  7.2 Coexistence Performance ............................................. 86
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>ICN approaches and their related projects</td>
<td>8</td>
</tr>
<tr>
<td>5.1</td>
<td>The SYN-SYNACK process that sets up the distribution tree</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>NetInf TP: data acknowledges the request; TCP: receiver ACKs the data</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>TCP three way handshake process</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>NetInf TP initialing messages</td>
<td>49</td>
</tr>
<tr>
<td>6.1</td>
<td>NetInfApp1.cc</td>
<td>54</td>
</tr>
<tr>
<td>6.2</td>
<td>handleMessage function</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>processPacket call graph</td>
<td>57</td>
</tr>
<tr>
<td>6.4</td>
<td>sendDataRequest caller graph</td>
<td>57</td>
</tr>
<tr>
<td>6.5</td>
<td>NetInfApp2 - processPacket call graph</td>
<td>58</td>
</tr>
<tr>
<td>6.6</td>
<td>Generic message - Inheritance Diagram</td>
<td>59</td>
</tr>
<tr>
<td>6.7</td>
<td>Requesting process</td>
<td>70</td>
</tr>
<tr>
<td>7.1</td>
<td>Network A: Topology for a single NetInf flow</td>
<td>81</td>
</tr>
<tr>
<td>7.2</td>
<td>Evolution of the MaxWindow (250KB-5ms-q5)</td>
<td>82</td>
</tr>
<tr>
<td>7.3</td>
<td>Requested and received segments (250KB-5ms-q5)</td>
<td>83</td>
</tr>
<tr>
<td>7.4</td>
<td>Evolution of the MaxWindow (3MB-5ms-q15)</td>
<td>84</td>
</tr>
<tr>
<td>7.5</td>
<td>Requested and received segments (3MB-5ms-q15)</td>
<td>85</td>
</tr>
<tr>
<td>7.6</td>
<td>Network B: Topology for Multiple NetInf and TCP flows</td>
<td>87</td>
</tr>
<tr>
<td>7.7</td>
<td>Two NetInf flows</td>
<td>88</td>
</tr>
<tr>
<td>7.8</td>
<td>Two TCP flows</td>
<td>88</td>
</tr>
<tr>
<td>7.9</td>
<td>Case I. TCP Reno</td>
<td>90</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>7.10 Case II. TCP New-Reno</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>7.11 Case III. TCP Reno + SACK</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>7.12 NetInf TP - TCP Fairness Index Values</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>A.1 Received Rate, Case I. (TCP Reno)</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>A.2 Received Bytes, Case I. (TCP Reno)</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>A.3 Outstanding data, Case I. (TCP Reno) close-up</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>A.4 Outstanding data, Case I. (TCP Reno)</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>A.5 Received Rate, Case III. (TCP New-Reno)</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>A.6 Received Bytes, Case III. (TCP New-Reno)</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>A.7 Outstanding data, Case III. (TCP New-Reno) close-up</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>A.8 Outstanding data, Case III. (TCP New-Reno)</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>A.9 Received Rate, Case II. (TCP Reno + SACK)</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>A.10 Received Bytes, Case II. (TCP Reno + SACK)</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>A.11 Outstanding data, Case II. (TCP Reno + SACK) close-up</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>A.12 Outstanding data, Case II. (TCP Reno + SACK)</td>
<td>108</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

5.1 Message fields of Resolution Request/Reply ............... 41
5.2 Message fields of Data_SYN/SYNACK ..................... 41
5.3 Message fields of Data_Request/Segment .................. 42
5.4 Feature comparison between NetInf TP and TCP variants ... 50

6.1 MaxSegmentSize ........................................... 68
6.2 outstanding_data ......................................... 68
6.3 MaxWindow .................................................. 69
Part I

Background
Chapter 1

Introduction

Alice and Bob were early Internet users. They exchanged email, remotely accessed terminals and transferred files. Alice and Bob became old and their children grew up using the Internet for browsing information, sharing pictures and news, downloading movies from friends and speaking to their friends via the Internet rather than the phone. Thereafter, the grandchildren of Alice and Bob have started to use services such as watching and listening to time-shifted TV, consumed radio and music on multiple devices, including portable ones, with high quality, whenever, wherever they wished.

Even though many technological developments of the Internet have taken place, the fundamental principles of Internet’s operation have not significantly changed since Alice and Bob’s generation. What has changed however is the user uptake, demands and expectations, and these have driven new technological advances. Nevertheless, the new services have faced limitations [45] in terms of scalability (increasing demand for different content) and availability (content is coupled to location) over the current infrastructure. Alice and Bob’s next generation children may still however experience problems. Therefore, some argue that, the time has come to address the existing architecture designs in order to better fit new requirements.
The main driving force of today’s Internet is information and media content distribution. The BBC alone had 24 separate video streams and many radio streams of the 2012 Olympic games. A high percentage of data traffic consists of video media, which has been increasing [10] since the introduction of time-shifted TV and video on demand services to many countries [30].

The existing Internet model is based on *host-centric* communication, i.e. the data is bound (stored) to a certain location and retrieved by addressing it, often via an http:// web server request. The emerging central role of information and content distribution led to a new concept for data retrieval, that is retrieving the data directly by its name, without addressing the location that holds the data. There are different approaches to realising this concept, commonly named as Information-Centric Networking (ICN). In an information-centric world, data is treated as named objects, which can be replicated and cached at intermediate nodes, avoiding redundant transfers of popular content over the same source and transit links. The data becomes “free standing” as opposed to the being accessed via the http and server transmitting the data. Streaming and even peer to peer protocols retrieve data from pre-determined locations, if not always servers. Network of Information (NetInf) is one of the major approaches of the information-centric networking concept [2].

In this thesis we focus on the transport mechanism of NetInf. We have developed a receiver-driven NetInf transport protocol called **NetInf TP** and tested it in a simulation framework. We evaluated the protocol’s performance with the aim of assessing the feasibility of the developed prototype as a potential transport protocol that can be deployed.

### 1.1 Thesis outline

We have divided the thesis into three parts:

1. **Background:** We provide the necessary background to our work.
The first chapter introduces the project topic and states the problem. The second chapter provides a basic overview of literature and related projects, followed by a general transport protocol overview described in the third chapter. In chapter four we introduce the environment used to develop and test our protocol.

2. **Design and Implementation:** Chapters five and six introduce our design and implementation in detail, then the seventh chapter evaluates the developed transport protocol’s performance, primarily from the perspective of efficiency and TCP-friendliness.

3. **Discussion and Future Work:** We discuss the results as well as future work, and then draw some conclusions.

### 1.2 Problem statement

First, we summarise what problem, we are trying to solve. In order to provide the reliable retrieval of named data objects in information-centric networks, a new type of transport protocol is required. Typically ICN networks have multiple data requestors and there can be multiple sources of data in the network which means the data object is replicated. The replicas should be created and cached at intermediary nodes as a side effect of data transport, thus saving the network from extra cost. In addition, concurrent legacy TCP connections may exist on the same communication paths, which requires inter-operation from a new transport protocol. Given the above stated problem, our objective is to implement a transport protocol that:

1. Enables a receiver-driven retrieval of named data objects.

2. Provides a TCP like congestion control mechanism, adapting to the receiver-driven context.

3. Supports ICN features, such as in-network caching, with a message-based communication concept.
4. Performs efficiently in a constrained network environment, where resources are shared between simultaneous flows.

5. Behaves fairly with TCP in terms of receiving equal share of the available bandwidth.

Additionally, we have the following sub-goals:

- Improve the performance of the transport protocol by introducing a unique retransmission strategy.
- Maintain a data structure for keeping track of request and response messages, as well as collected data segments.
- Implement a packet loss detection mechanism that signals congestion more reliable than by using a timer based approach only.
- Define a metric for measuring performance in terms of efficiency and friendliness.

1.3 Research question

How feasible is NetInf TP, a receiver-driven ICN based transport protocol, for basic reliable data retrieval?

1.4 Thesis Contribution

The main contribution of this thesis is a running prototype of NetInf TP. The prototype is one of the few transport protocols which is based on the concept of ICN that can be used for further developments and experiments. The proposed transport protocol aims to solve the problem of dissemination of data objects in the future Internet. NetInf TP can be considered as a preliminary proof-of-concept on the direction of real word deployments.

The NetInf TP prototype was built in OMNeT++ and tested through simulations. The developed protocol is available as an OMNeT++ project.
that contains the applications and message definitions along with the network topologies used for testing.

**Authors’ contribution**

The thesis work was carried out as a joint collaboration between the authors. The design of NetInf TP was largely performed by Björn Grönvall, with frequent consultations with the authors. The implementation and evaluation work was done by the authors, supervised by Björn Grönvall and Ian Marsh. The thesis and presentations were shared equally, with editorial comments by Ian Marsh and Flutra Osmani.
Chapter 2

Literature Review

In this chapter, an overview of relevant literature is provided. In the first section, we review the background literature related to Information-Centric Networking. We discuss the main ICN approaches, and describe the different naming schemes, routing and forwarding techniques. After that, the NetInf ICN approach is described in more detail. In the second section, we describe some work related to the evaluation methodology we used. Lastly, the chapter will be concluded by discussing parallel research activities in the field of ICN based transport protocols.

2.1 Background Literature

2.1.1 Information-Centric Networking

The objective of Information-Centric Networking (ICN) concept is to efficiently distribute content over an internetwork. The major focus is on the data objects and their properties. The receivers’ interest in the network to distribute data rather than access servers.

As data objects are the core components in this model, the naming of objects such as files, images or other documents is required. This is needed in identifying and determining the content in a distributed network. The ICN approach has begun to be been documented [2] with a brief taxonomy
summarised in Figure 2.1.

Data oriented Network Architecture (DONA) [28]

Content Centric Networking (CCN) [24], Named Data Networking (NDN) project (www.named-data.org)

Publish-Subscribe Internet Routing Paradigm (PSIRP) [3], Publish-Subscribe Internet Technology (PURSUIT) project (www.fp7-pursuit.eu).

Network of Information (NetInf) [1] - Scalable and Adaptive Internet Solutions (SAIL) project (www.sail-project.eu)

**Data oriented Network Architecture (DONA).** In DONA, there are many changes on the existing host based model that resulted in an architecture which built around a data oriented model. The traditional Domain Name System (DNS) names are replaced by flat names and the DNS resolution is based on name based routing i.e. it uses a route-by-name strategy. The DNS servers are also considered obsolete in DONA and hence Resolution Handlers (RHs) are used.

The DONA names are of the P:L form where P is the cryptographic hash of the principal’s public key and L is the label chosen by the principal which ensures that the names are unique. When a client asks for a data with a name P:L, the client will receive the data itself, the public key and it’s
signature, and then the data will be checked if it came from the principal by checking hashing of the public key which will then ensure that names are not explicitly referred to by locations. This means that the data can be hosted anywhere in the network [28]. DONA’s name resolution is based on routing by name. It uses two types of messages which are FIND and REGISTER messages. It works when a host sends a FIND packet to locate the object in the network and handlers route the request to the nearest destination. Moreover, register messages are used by the handlers to route the FIND messages quickly.

**Content Centric Networking (CCN).** There is another information-centric approach known as Content Centric Networking (CCN) in which the primary purpose is to isolate the content from location and retrieve it by name. The model is based and driven by the content itself and does not secure the communicating path over which the data is traversing. The CCN model uses two types of packets to initiate and retrieve data i.e. (i) interest and (ii) data packets. If host A wants a particular named data item then it will broadcast an interest packet into the network. All the nodes will receive the interest packet, and only one node will respond which has the data. The response will be a data packet. Since the interest and data packets are exchanged based on names, this means that several nodes may have the data, so multiple nodes can share the transmission using multicast [24]. The CCN names are hierarchical and can easily be hashed for lookup. As the CCN names are longer than IP addresses, the lookup operation in CCN becomes more efficient than IP lookup. As CCN and IP nodes are similar, the longest prefix match is performed on the name and the action is taken based on the result of that lookup [43]. CCN uses a name-based routing in which hosts/clients ask for a data object by sending interest packets. The interest packets are routed towards the publisher of the name prefix using longest-prefix matching in the Forwarding Information Base (FIB) of each node. These FIBs are built using routing protocols similar to those used in
todays Internet. Moreover, the CCN nodes keep state for each outstanding request in the Pending Interest Table (PIT) [24].

**Publish-Subscribe Internet Routing Paradigm (PSIRP).** PSIRP is another information-centric approach, during communication between hosts, the primary focus is on the successful retrieval of the data rather then the reachability of the end points. The projects approach is based on a publish-subscribe model. The receivers of information have control over their expression of interest and therefore the reception of information. In PSIRP, data objects are published into the network by the sources and these publications belong to a particular named scope. The receivers can subscribe to the NDOs and the publications and subscriptions are matched by a rendezvous system [3]. There are two types of PSIRP names which are Rendezvous Identifiers (RI) and Scope Identifiers (SI). Both belong to a flat namespace. The combination of RIs with SIs are used to name data objects which are then mapped to Rendezvous Points (RPs). These RPs are used to establish contact between publishers and subscribers. Moreover in PSIRP, there are forwarding identifiers (FI) which are used to transport data. But these FIs are not NDOs, they solely identify a path from the publisher to a subscriber [3].

### 2.1.2 Network of Information (NetInf)

The Scalable and Adaptive Internet Solutions (SAIL) project has been responsible for developing the Networking of Information (NetInf). The aim of NetInf is to provide the means for applications to locate, publish and retrieve information objects. An important mechanism of the approach is the in-network caching of content which can provide improved performance by making the content available in multiple locations. Also, the security of the model improves through Named Data Objects (NDOs) [1]. NDOs can be Internet text data, videos and so on.
The naming scheme. Objects are identified by names that are not dependent on their locations. The names of the data objects are used to forward requests, so they should be unique. In NetInf, a flat namespace [11] is generally used which is similar to DONA’s namespace. The common NetInf naming format [27] uses hash digests in the names and are located by associated hashing schemes and a lookup system.

Routing and forwarding. NetInf can perform routing based on the names of NDOs. However, it is not considered as scalable [1] therefore NetInf introduces a Name Resolution System (NRS) to map the names to locators that will identify the physical entities of the network. Moreover, the forwarding takes place in an incremental manner through several NetInf hops [1].

The security model and name data integrity check. Security is one of the most important characteristic of the NetInf architecture. It is considered more reliable then in the host based architecture which relies only on connection security. This is because objects are replicated in many locations in a network and the authenticity of these objects is necessary to verify both the integrity and ownership of the object [1]. It is very important for the receiver and other network nodes to perform a name data integrity check so that they can know from where the request has been made. It works in a way that the object which is retrieved will be match with the name of the object which is requested. This can be done by including a hash of the object with its name and when the object is returned, the hash is calculated and both the values are compared [37]. In NetInf capable routers, only those objects are cached which passes through the integrity check so it is important that the applications should have this feature enabled before transporting data in a network.

Messages and Responses. There are several messages and their responses [1] in a NetInf protocol which are discussed briefly:
GET/GET-RESP. The GET message is used to request an object from the NetInf capable network. When the host receives a GET message, if it has the object then it responds with a GET-RESP message. Both these messages should have the same message ID so that these messages can be attributed to the same transaction.

PUBLISH/PUBLISH-RESP. With the PUBLISH message, the object with this name will be pushed into the network and the PUBLISH-RESP message will result in an acknowledgement.

SEARCH/SEARCH-RESP. The SEARCH message allows an object with a specific name or name pattern to be found in a NetInf capable network. The response will be a part or the full object itself which must have that name within it.

2.2 Related Work

There has a lot of work been done in evaluating different types of transport protocols which includes high speed variants of TCP or traditional flavors of TCP in different simulation environments. The major part of these papers is to examine the performance behavior of different protocols under certain network and simulation conditions. Some of the researchers were able to conclude both qualitatively and quantitatively which protocol behaves efficiently then the rest whereas some have not been able to reach the consensus. Inter protocol fairness is a very important aspect which was highlighted in the Cubic TCP [19] paper. Moreover, there are short and long round trip times (RTT) being used in the experiments which yields somewhat reasonable conclusions.

The simulation results, network conditions and topology maps used in these different papers are summarized one by one.
2.2.1 Simulation-based comparisons of Tahoe, Reno and SACK TCP

Floyd and Fall [12] evaluated the performance behavior of Tahoe, Reno, NewReno and SACK TCP over the same topology. The authors were able to explain in the paper, how the inclusion of selective acknowledgement options in a TCP protocol can solve the performance problems when multiple packets are dropped from a window of data. They also showed that TCP can retransmit a maximum of one packet per round-trip time without the presence of SACK option. The simulation was performed in NS-2 simulator with four different scenarios (every case will increase one additional drop of packet from a window of data). The behavior of TCP Tahoe was pretty much identical in all these scenarios i.e. the sender recovers from the packet loss by going into slow start. Reno was a bit better when one packet loss occurred but it was identical in the rest of the cases. And as proved and argued by the authors, NewReno and SACK were efficient in all four scenarios when they recovered from a loss without having to wait for a retransmit timeout.

The topology was based on one sender and one receiver. A 8Mbps bandwidth link with a RTT of 0.1ms was used from the sender to the router whereas 0.8Mbps link having a 100ms RTT was used from the router to the receiver. Moreover, finite buffer drop tail gateways were used. These were not any realistic or standard values for buffers and links, the point is to show the efficiency and accuracy of TCP SACK variant as compare to other protocols. Similarly, in our thesis work, we have also used Reno, NewReno and SACK variants and they have shown similar behavior with NetInf TP as discussed in this paper.

2.2.2 Using OMNeT++ to Simulate TCP

The purpose of this tutorial [4] is to reproduce and verify the results of Floyd and Fall’s paper in the OMNeT++ simulation environment. The tutorial showed similar results that of ns-2, with minor differences depending on the used TCP variant. In our experiments of NetInf TP, we have used
OMNeT++ simulator and also the same TCP variants, thus considered the methodology and results of this tutorial.

2.2.3 CUBIC - A New TCP-Friendly High Speed TCP Variant

Rhee and Xu [19] have presented a new TCP variant called TCP Cubic, an enhanced version of BIC-TCP which is used in high speed networks. In the evaluation phase, a dumbbell topology was used in which the stability of the protocol was checked along with the TCP friendliness behavior of the protocol. In the simulations, there were four high speed and four regular TCP SACK flows been used with a bottleneck link varying from 20Mbps to 1Gbps. There were different types of experiments performed, one in which the RTT was kept shorter (around 10ms) and in the other one, RTT value of 100ms was used. In both of these experiments, the authors were able to explain in a very concise way that CUBIC showed a better friendly ratio then the other protocols. In our experiments of NetInf TP, we have also followed the values and the topology similar to what is used in this paper. The idea to use a dumbbell topology is to verify the operation of the protocol with multiple flows running in the network and also compare the throughput of NetInf TP with different versions of TCP.

2.3 Other Work

With a future Internet technology in place which addresses named data objects instead of the host locations, it is important to address problems at the transport layer. As TCP has been widely used as a transport protocol in host based architecture, it does not entirely fit in the content based models. The ICN models such as Content-centric networking (CCN) have features such as storage capabilities in the network and receiver being the driving force behind the retrieval process, it was necessary for the researchers to introduce a new design and implementation of ICN based transport
protocols. Few of the research papers are discussed one by one.

2.3.1 ICP - Design and Evaluation of an Interest Control Protocol for CCN

The objective of the paper [6] revolves around three important points (i) Design of a receiver driven Interest control protocol for CCN based on AIMD algorithm. Also, a window based flow control is used. (ii) The designed protocol has been analyzed on a single and multiple bottleneck links. Moreover, in-network caching model have also been proposed (iii) Packet level simulations to test the protocol. The topic of the paper is very significant to our thesis work as the concept of the paper is somewhat similar to what we are doing but the design and implementation is completely different. In a similar way as our protocol is designed, the receiver is driving the communication and maintaining the window size and also initiating the communication.

The receiver driven protocol is responsible for efficient data retrieval by adjusting the request rate of the receiver which should be aligned with the network resources. In CCN, Interest queries are continuously sent out by the receiver through which a transport session should be created. There are certain goals which are achieved by ICP transport protocol i.e. to have a reliable and efficient data transfer and to achieve fair bandwidth among several ICP flows. In the protocol, data packets are known with a unique name as been used in the ICN approaches. The names are the combinations of the content name with the segment ID. The contents are requested via Interest packets and from receiver window it can be identify how many Interests, a receiver is allowed to send.

Reliability    The reliability is ensured by expressing the Interest again after a packet loss occurs. ICP schedules the retransmitting of Interest packets after the expiration of time ’t’ which means that the protocol depends on timer expiration instead of the signal loss. On the other hand, in NetInf TP,
the retransmitting strategy is different then ICP as we are dealing with a cyclic manner retransmission which would be discussed in later chapters.

**Efficiency**  The efficiency factor is maintained by minimizing the completion time of data transfers. For this, ICP uses additive increase multiplicative decrease (AIMD) mechanism to use the maximum available rate allowed by using the window size.

### 2.3.2 Flow-aware Traffic Control a Content-Centric Network

For ICN models such as CCN, it is important to have a framework that ensures control on sharing of network resources by concurrent flows. This is what is discussed by J. Roberts et al. in the paper [34] where traffic should be controlled based on per-flow bandwidth sharing. This means that if a user wishes to download a file with higher rate, it will not effect the download speed of other users and hence maintains a fair share of bandwidth among these downloads. It will forbidden any unfair sharing of the resources.

In CCN model, Data packets (carries the actual payload) are sent in response to the Interest packets. Both these packets have the same object name from where the flows can be identified. For storing these flows and controlling the traffic, buffer management system on the router is essential. Occasionally buffers are very small in size which cannot handle huge amount of traffic so a cache system has been introduced which is called Content Store (CS) that is pretty cheap and have higher capacity. The idea of having cache storage is different from what we have in our thesis as we are relying on the router’s buffer memory to store and forward the traffic.

As traffic is dynamic and it can remain for a finite period of time over the network, so it is essential to impose fair traffic share. Moreover, if the demands of the traffic exceeds the flow arrival rate then overload occurs. The paper focused on the advantages of imposing fair sharing such as (i) end systems are relived by performing TCP friendly congestion control algorithm, (ii) flows who exceeds the fair rate will receive packet loss and delay.
However, in our thesis, TCP friendly congestion control algorithm has been implemented to differentiate flows that are fair to others.

The authors simulated a basic dumbbell topology showing the performance of the CCN where there are multiple flows sharing the same set of links to transfer packets. The results were carried out with and without the Interest discarding technique which is used in CCN. Different parameter values (such as AIMD’s additive and multiplicative rates and RTT) were changed while recording the throughput values. The conclusions were reached as which flow is fair as compare to others after getting the values. Moreover, it was observed in the paper that fairness allows applications to implement aggressive congestion control.
Chapter 3

Transport Protocol Overview

Transport layer lies between network and application layer in a layered architecture of Internet. It deals with an end to end communication between processes existing on different hosts in a network. Currently, the various transport protocols includes Transmission Control protocol (TCP) [35], User datagram protocol (UDP) [36], Stream Control Transmission Protocol (SCTP) [41] and Datagram Congestion Control Protocol (DCCP) [14].

The role of a transport protocol is to provide different functionalities to the processes/applications such as data delivery, data reliability, flow control and congestion control. However, not all of these transport protocols provides such services, few of these services are provided by one transport protocol and few by the others. The objectives of general purpose transport protocols are transparent to the applications such as convergence, efficiency and fairness.

A transport protocol needs to be very efficient when it comes to utilizing the available bandwidth. Here, the term "efficiency" means that it should probe the maximum available bandwidth and recovers to the maximum speed once it experiences a loss or congestion. But once it reaches to the maximum speed, it should remain in a constant state until the network state changes.

Congestion control is a very important feature of a transport protocol.
A transport protocol uses a number of techniques to achieve optimal performance and avoid congestion in a network. The transport protocol adjusts the data sending rate using a certain congestion control algorithm. Network congestion control is usually generated from intermediate nodes such as routers or it can be estimated by packet losses which increases trends in packet delays or timeout events.

3.1 Transmission Control Protocol (TCP)

TCP is the most widely used transport protocol and a standard Internet data transport protocol. TCP is renown for providing reliable data service to applications. The major success of TCP is mainly due to the fact that it is very much stable in its connectivity and has reliable transportation in a network. However, the usage of high performance applications are quite different from that of traditional Internet applications because of the fact that data transfer often lasts very long at high speeds and some applications required multiple data connections. Since the time when TCP was first proposed, there have been different models that came into being to improve its performance or to rectify the issues found in previous models. The versions includes Tahoe [22], Reno [39], New-Reno [15], Selective Acknowledgement (SACK) [16, 17] and other TCP variants.

3.1.1 TCP’s Congestion Control

Congestion in a network occurs if there are loads of packet flows across the network without any control over it. This can happen if the packets are injected into the network without realizing that the old packets are not gone out. To avoid this, TCP uses a Congestion control algorithm [22] which includes slow start, additive increase and multiplicative decrease schemes.

In order to make the TCP connection into an equilibrium (balance) state, slow start is used which gradually increases the amount of data in transit.
This part of congestion control algorithm is also known as exponential growth phase as the data increases exponentially. It works by increases the TCP congestion window (cwnd) each time the acknowledgement is received which means that the window size is increased by the number of segments acknowledged. For example, the initial value of cwnd is usually set to one segment and if the acknowledgement is received then that cwnd size will increase by '1 segment'. The sender can transmit up to the minimum of the congestion window and the advertised window [39]. These two windows are different from each other as congestion window is based on the assessment done by the sender with respect to network congestion whereas the received window is related to the queue capacity of the receiver.

As there is an initial value (usually one segment) assigned to cwnd, just like this there is a threshold value known as \textit{ssthresh} is set to usually 65535 bytes. With the congestion window grows larger, it stops at a point once the cwnd goes larger then ssthresh or if a packet gets lost which is either due to network congestion or insufficient buffer capacity [39]. At this point, the second phase of TCP congestion control algorithm takes place which is called "Congestion avoidance" [26]. In congestion avoidance, the congestion window is additively increased by one packet per round trip time (RTT) after receiving every ACK from the receiver. Moreover, if a timeout occurs then the congestion window is reduced to half the current window size which is known as multiplicative decrease [22].

The ssthresh value determines which phase of congestion control is in place. There are two particular cases which can happen

- If cwnd is less then or equal to ssthresh , this means that the slow start is in progress.
- If cwnd is greater then ssthresh , this shows that congestion avoidance has taken over.
3.1.2 Fast Retransmit Algorithm

**Fast retransmit** [22] is a very important algorithm of TCP which immediately retransmits lost packets once the sender finds out that the packet has been lost from the window of data. The sender tags a particular packet as lost after receiving a small number of duplicate acknowledgements (duplicate ACKs). This will result in an immediate retransmission of the lost packets leading to a higher connection throughput and channel utilization.

By doing fast retransmission, the sender is not waiting for the timer to expire, resulting in reducing the time from the source side. This is achieved by using the concept of duplicate ACKs which are the acknowledgements with the same acknowledgement number. For example, if the source sends a packet with sequence number '1', the receiver acknowledges back with acknowledgement number '2' which means that it is expecting the next packet from the source with sequence number '2'. If the next packet from the source end is not '2' (meaning that packets have lost in between) then receiver will continue to send ACKs with acknowledgement number '2' and these multiple acknowledgements with the same number are called duplicate ACKs. In TCP implementations, after receiving three duplicate ACKs (meaning four ACKs in total) with the same acknowledgement number, the packet is considered as a loss packet and will be retransmit immediately.

3.1.3 Fast Recovery Algorithm

The Reno implementation of TCP introduced an algorithm known as **Fast Recovery** [23] which works together with Fast retransmit mechanism. The algorithm allows the communication path to be filled with packets and not becoming empty after fast retransmit which will avoid the use of slow start after a packet loss. The mechanism considers each received duplicate ACK as a single packet leaving the path, resulting in a better measurement of the outstanding data from the sender’s side. The outstanding data is the amount of data currently flowing in the network.

Fast recovery algorithm is an improvement that allows high throughput
under moderate congestion. The duplicate ACKs will be generated by the receiver and if any data packet is received in between then that will remain in the receiver’s buffer meaning that it will leave the network. This will make sure that the data will still be flowing between the two hosts and TCP does not want to reduce the flow by going into slow start [39].

3.2 TCP Implementations

3.2.1 TCP Reno

The first implementation of TCP was Tahoe which introduced the concept of Fast retransmit. The behaviour of TCP Tahoe is not reasonable when there is a packet loss incurred because it does not recover from losses. Due to this, a new model was proposed known as TCP Reno which modified the operation of Fast retransmit with Fast recovery. In fast recovery, a parameter is used known as "tcprexmitthresh" which is a threshold value and is generally set to three. Once the threshold of duplicate ACKs is received, the sender retransmits one packet and reduces its congestion window by one half. This makes sure that instead of going into slow start which was done in the case of TCP Tahoe, the Reno sender has a better approach of dealing with packet losses.

In Reno, the sender’s window is the minimum value of the receiver’s advertised window and the sender’s congestion window plus the number of duplicate ACKs (ndup). This value of ndup remains at zero until it reaches tcprexmitthresh. As each duplicate ACK signals that a packet has been removed from the network and it has remained in the receiver’s buffer so during fast recovery the sender artificially inflates its window by the number of duplicate ACKs it has received. This shows that Reno significantly improved the behaviour of TCP Tahoe when a single packet has been lost from a window of data but once there are multiple losses then
it faces performance problems [12] as for every lost packet, it triggers the fast recovery several times hence reducing the congestion window significantly [40].

Also, there is a known problem in TCP Reno’s congestion control algorithm that it remains idle for a pretty long time once the recovery period is over [40]. After the idle period finishes, TCP cannot strobe new packets in the network as all the ACKs have gone out of the network so [22] suggested that TCP should use slow start to restart transmission after relatively long idle time.

### 3.2.2 TCP New-Reno

An enhanced version of TCP Reno known as TCP NewReno came into being which rectified somewhat the performance problems of TCP when there are many packet losses in a window of data [15]. Due to this reason, it makes the newer version of TCP Reno much more efficient and scalable as compared to the older one. In the case of fast retransmit, TCP NewReno enters into this mode just like TCP Reno when it receives multiple duplicate packets but for fast recovery mode, it differs from the older Reno version. The newer version of TCP Reno does not exit the fast recovery phase until all the data which was outstanding at the time it entered into the fast recovery is acknowledged. Due to this, the problem faced by Reno of reducing the congestion window many times has been solved [15].

As New-Reno in the fast recovery phase allows for multiple retransmissions, it always looks for the segments which are outstanding when it enters this phase. When a fresh ACK is received then NewReno works according to one of the mentioned below two cases:

(i) If all the segments which were outstanding are acknowledged then the congestion window is set to ssthresh and continued with the congestion avoidance phase.

(ii) If the ACK is a partial one then it will assume that the next segment in line was lost and retransmits that segment and sets the number
of duplicate ACKs received to zero.

As soon as all the data in the window is acknowledged, it exits the fast recovery phase. There is a major problem with this implementation which is it takes one round trip time to detect a packet loss meaning that when the acknowledgement of the retransmitted segment is received only then the segment loss can be identified [20].

### 3.2.3 TCP SACK

The Reno implementations of TCP did not really solve the performance issues in the case of dropping out multiple packets from a window of data. The problems that were faced by Reno and NewReno such as unable to detect multiple lost packets and inability to retransmit more then one lost packet in a round trip time has been solved by a newer TCP model known as TCP with Selective Acknowledgements (TCP SACK). From the cumulative acknowledgements concept which was used before, a TCP sender can have limited information which means it can either retransmit only one loss packet in a round trip time or chose to retransmit more packets in a RTT, but those packets may have already been received successfully.

A Selective Acknowledge (SACK) concept came into being to overcome these issues. A TCP receiver sends SACK packets back to the sender informing him that he has received a particular packet so that the sender knows which packets have not reached to the receiver and retransmit only those packets [16]. This is the most efficient and effective way of detecting and retransmitting lost packets and with that it also increases the performance. Moreover, SACK has the ability to operate with slow start and fast retransmit algorithms which were part of Reno implementation.

SACK version of TCP demands that the segments should be acknowledged in a selective way rather then in a cumulative way. Each ACK has a field which describes which segments have been acknowledged so that the sender can differentiate between the acknowledged and outstanding segments in a network. Once the sender enters the fast recovery mode,
it uses a pipe variable describing the estimate of outstanding data in the network and setting the window size to half the current size of it. So, every time when it receives an ACK, it reduces the value of pipe by 1 and whenever a retransmission takes place then it increments by 1. A new packet is sent by the sender if there is no outstanding data left in the network. In this way, it is being able to send more then one lost packet in a round trip time [17].

3.2.4 TCP Cubic

In previous TCP models, the window size grows linearly meaning that in one round trip time the window size is increased by one packet. This results in under utilization of the bandwidth which was a real problem to deal with. To counter such problem, different TCP variants are proposed and Linux community implemented these protocols in their operating system.

TCP Cubic [19] is the default TCP algorithm in Linux which improves the scalability of TCP over wide and long distance networks by modifying the linear window growth function of existing TCP standards to a cubic function. The use of TCP Cubic became necessary as the Internet is evolving with high speed and long distance network paths resulting in larger bandwidth and delay representing the total number of packets in transient which utilizes the bandwidth completely. In other words, the size of the congestion window should be fully utilized. The version also simplified the window adjustment algorithm in the earlier Linux based protocol BIC-TCP [44].

CUBIC has many advantages over previous TCP models but its key feature is its window growth that depends on the real time between two consecutive congestion events. Out of these two events, one congestion event is the time when TCP is in fast recovery mode which makes window growth independent of RTTs. This feature allows CUBIC flows to be in the same bottleneck having the same window size independent of their RTTs and achieving good RTT fairness [19].
3.3 TCP Friendliness

3.3.1 TCP Friendly Factor

When a new transport protocol is developed, it is important that the protocol receives network shares not greater then the shares being used by the concurrent TCP protocol running on the network. This is important in order to avoid congestion collapse as TCP is running on majority of the networks on the Internet. So, the non TCP-flows are termed as TCP-friendly if their throughput doesn’t exceed the throughput of a corresponding TCP flow under similar conditions [42].

For controlling the congestion smoothly, it is important that the resources are shared fairly in the network to achieve TCP-friendliness nature. This has been a problem with TCP when it comes to dealing with long round trip times (RTT) [5]. For example, if there are multiple connections in a network and the RTT of these connections have either smaller or larger RTT then they won’t achieve the same throughput. One way of controlling congestion is by implementing AIMD algorithm that TCP have been using [22] but it will lead to fairness when all the connections increase their rates with a fix margin [9]. This type of fairness is called max-min fairness [13] where the the bandwidth is allocated equally to all flows and the bandwidth of the bottleneck link matters regardless of the consumption of other links.

TCP-Friendliness for Unicast Under similar conditions, if a non-TCP flow is receiving network share lesser then what concurrent TCP flow is receiving or when it doesn’t reduce the throughput of the TCP flow then it is considered TCP-friendly.

TCP-Friendliness for Multicast TCP-friendliness is maintained in a network if the multiple non TCP flows are treating the TCP flows fairly. This doesn’t always mean that all the flows on the bottleneck link receive the same amount of throughput. For example flows with different RTTs can transmit/receive at different rates.
3.3.2 Fairness Metric

There is another way of knowing how the set of resources are shared by the number of users. This form is called the quantitative approach where a fairness metric is calculated via an equation called Raj Jain’s equation [25]. There are different characteristics of fairness and it depends on which network parameter, the fairness need to be calculated.

**Fairness (Response time)** When the aim is to provide similar response time to all the flows then following equation will be used

\[
\text{Fairness (response time)} = \frac{\left( \sum_{i=1}^{n} R_i \right)^2}{n \cdot \sum_{i=1}^{n} R_i^2}
\]

where \( R_i = \) Response time for ith user = \( h + \sum_{i=1}^{n} c_i \)

and \( c_i = \) Window size of ith user ; \( h = \) number of hops

**Fairness (Throughput or Window size)** If the network have many simultaneous flows and every flow should take the same amount of throughput then following fairness index would be used

\[
\text{Fairness (throughput)} = \frac{\left( \sum_{i=1}^{n} T_i \right)^2}{n \cdot \sum_{i=1}^{n} T_i^2}
\]

where \( T_i = \) Throughput for ith user = \( c_i / h + \sum_{i=1}^{n} c_i \)

Similarly, for window sizes, the fairness index would be similar

\[
\text{Fairness (window size)} = \frac{\left( \sum_{i=1}^{n} c_i \right)^2}{n \cdot \sum_{i=1}^{n} c_i^2}
\]
**Fairness (Power)**  When it comes to providing equal power to all the flows then the fairness formula would change to

\[
\text{Fairness (response time)} = \frac{\left( \sum_{i=1}^{n} P_i \right)^2}{(n \cdot \sum_{i=1}^{n} P_i^2)} = \frac{\left( \sum_{i=1}^{n} c_i \right)^2}{(n \cdot \sum_{i=1}^{n} c_i^2)}
\]

which means that Fairness (window size) = Fairness (throughput)
Chapter 4

Environment

This chapter gives a brief overview about the simulation environment used in the implementation and evaluation of the transport protocol. In section 4.1, different features, files and concepts of OMNeT++ simulator are discussed. This will be followed by the INET Framework (section 4.2) which is solely responsible for the network related packages.

4.1 OMNeT++

OMNeT++ is a discrete event simulation environment based on C++ programming language and is primarily designed for building network simulators [33]. OMNeT++ provides a modular and component based architecture for designing communication networks. The driving force behind our choice to use OMNeT++ was the availability of INET Framework which provides various network models as described in section 4.2.

4.1.1 Modelling Concepts

OMNet++ models networks and network entities with using hierarchical modules which are communicating with each other by sending messages [32]. The top level of the module hierarchy is the system module which consists of several submodules nested in each other. Modules containing other submodules called compound modules, while the modules on the lowest
level of the hierarchy with no further nesting are referred as *simple modules* [32]. Simple modules are written in C++ to files with .cc and .h extensions, implementing e.g. applications, hosts or protocols.

**NED language**

Network topologies are described in a compound language called Network Description (NED) where the user can define the structure of the modules providing certain parameters. Network topology descriptions are written in NED files having .ned as an extension.

**Configuration file**

The configurations used by the simulator are provided in *omnetpp.ini* files where the parameters used by the .ned or .cc/.h files are assigned. The *omnetpp.ini* file also accepts wild-card matching of parameters, hence certain functionalities of modules e.g. the transport protocol used by a host can be set via matching the corresponding parameter in the configuration file.

**4.1.2 Messages and packets**

The communication of OMNet++ modules is done by exchanging messages. A message represents any network packet, frame or other mobile entity containing arbitrary complex data structures. Messages can be sent from simple modules to certain destination addresses or can follow predefined paths.

**Self-messages**

A message arriving to the same module where it was originated is a *self-message*. Self-messages are used to implement timers, which is described in 4.1.3.
Message Definitions

Message definitions are written in a compact syntax in files with .msg extension and the corresponding C++ code including set and get methods, which are used to access the values stored in the message fields during the simulation, are generated by OMNet++.

4.1.3 Discrete event simulation

Each sending or arrival of a message corresponds to a simulation event which represents the simulation time. Message events can launch other events such as function calls of a module for processing or sending further messages.

Implementing timers

Timers are implemented with scheduling and sending self-messages. The message will be sent to the module itself at the scheduled simulation time, which corresponds to the expiration time of a timer. Timers (scheduled self-messages) can be cancelled with calling the corresponding cancel and/or delete functions, and the scheduled message will then be removed from the Future Event Set (FES) [32].

4.1.4 Simulator and Analysing tool

The simulation executable is a standalone program which can be run under the following user interfaces:

Tkenv: A Tcl/Tk-based graphical user interface

Cmdenkenv: A command-line user interface for batch execution

In our implementation the Tkenv graphical interface was used for simulations as it supports interactive execution with tracing and debugging opportunities, as well as provides a detailed picture of network activities.
Analysing tool is a built-in feature of the OmNet++ IDE. It offers adequate plotting and analysing opportunities of data recorded during the simulation.

4.2 INET Framework

The INET Framework is an open source communication networks simulation package for the OmNeT++ simulation environment [21]. The INET Framework provides several models for network elements such as routers and terminals or protocol emulations like IP, TCP, UDP, Ethernet or PPP to use in OmNet++ simulations.

As our transport protocol has to interact with the existing IP infrastructure and underlying protocols, the reliable implementation of these network modules and protocols is of key importance.

TCP implementations

The INET Framework includes implementations of different TCP flavours such as TCP Reno, TCP Tahoe or TCP NewReno with several settings option e.g. enable or disable SACK support for TCP Reno or enable/disable the usage of delayed ACK algorithm and so on. These implementations are used in our project for simulations investigating the co-existence of NetInf TP and different TCP variants.

Network Simulation Cradle

The Network Simulation Cradle (NSC) is a framework which allows real world operating systems’ network stacks to be used inside a network simulator [31]. Therefore the real world Linux TCP implementation could be integrated in INET, providing in a simulation a more accurate picture of how NetInf TP co-exists along with TCP. This can be seen as a good future work and it is outside the scope of this thesis.
Part II

Design and Implementation
Chapter 5

Transport Protocol Design

In this chapter, section 5.1 describes the broad design concepts and architecture of the NetInf transport protocol. Section 5.2 draws a comparison with existing TCP implementations. The particular details of what is implemented within this wider scope design, and how it is realized are covered by the following chapter (chapter 6) which introduces the prototype.

5.1 NetInf Transport Protocol

The NetInf Transport Protocol (NetInf TP) is a receiver driven protocol, using Additive Increase Multiplicative Decrease (AIMD) algorithm as feedback control in its congestion avoidance mechanism. Similarly to TCP’s congestion window, NetInf TP adjusts a window size to control the transmission of the data over the network, but instead of the sender, the receiver defines the actual value of this window. AIMD algorithm is used to keep the window size optimal, hence avoiding congestions and packet loss, but transferring the data with efficient rate.

5.1.1 Objectives

NetInf TP is designed for supporting such scenarios, when home users from different locations with different Internet connection speeds would like to get the same e.g. video content. In other words, NetInf TP is designed for
catering multiple receivers that can be slightly time shifted and operate at
different rates [18].

In wider scope NetInf TP is also responsible for the creation of new
replicas, which shall happen as a "side effect" of the transport process,
where the transport setup is also an important factor for
routers, the ability to operate on line speeds is also an important factor for
the protocol, as well as the security considerations, i.e. to avoid denial-of-
service (DoS) attacks [18].

5.1.2 Operation

From a higher perspective, the NetInf TP’s operation can be divided into
several phases.

- In the first phase the protocol locates the replicas, and selects the best
  replica based on distance or other metrics.

- The second phase is the transport setup, where the distribution tree
  is created for the later data transfer. The protocol sets up a transient
  state in the NetInf capable nodes of the network which records the
  information of requestors and used to fork the data flow when there
  are multiple requests for the same content. The transport setup phase
  is described in detail in 5.1.4.

- In the third, data transfer phase, the actual transfer of data begins,
  as well as creating new replicas along the network. Once the data
  is collected, a name data integrity check is performed. This phase is
  described in 5.1.5.

- In the last phase, new replicas are announced.

In our implementation we are focusing on the data transfer phase, which
is explained here as a high level concept and the particular realization is
described in the next chapter 6.
The selection and announcement of replicas which also requires interaction with the Name Resolution System (NRS) or within the data transfer phase, the in-network caching of data objects are outside the scope of this work and implemented partially with major simplifications.

5.1.3 Network concept

In NetInf context, a network is divided into zones and all the zones are either connected to each other via some routing protocol or are manually connected using any physical connectivity. Conceptually, a term zone is a synonym of area which is considered as a smaller region of a network. The use of this division is to have less processing overhead inside a network and have faster routing of packets.

Each zone has a set of NetInf capable routers. Every zone has a designated router (DR) on each NetInf capable interface.

- Terminals in a zone are connected to the DR. The DR acts as a gateway for a particular zone.
- The DR forwards the request which is received from the requesting party towards the source.
- Once the DR receives the request, it records the terminal and maintains a state. Here the state is being maintained due to the fact that every DR may have more then one terminals connected to it so it should know where to deliver the actual data.

When a host requests for the data then this request flows across many zones before locating the actual source. After the host has been located, the data is sent to the host who requested the data. This traversing of data flows across many routers and it will be cached along the path so that the next time when some host from the same or closest sub network asks for the same data then it will be fetch from the nearby router instead of going to the original source. This formation of data’s replica is done by the
pivot router which holds transient soft state during these operations. It is necessary beforehand to pick a pivot router so that every router in the zone knows who is creating a replica. All the routers in the zone collaborate in picking a pivot router [18].

5.1.4 Transport setup

Assume that the first phase has been completed, which means that the receiver’s terminal have successfully picked the "best" replica of the content, and the receiver knows where to retrieve it from. The next phase follows with setting up the distribution tree on the network for the later data transfer. Figure 5.1 shows the procedure of the transport setup.

The second phase begins with receiver sending a Data SYN message to its DR. While the Data SYN message is forwarded zone by zone to the selected replica, the message is processed by each zone’s ingress router (IR) and then pivot routers are selected. In each zone the address of traversed pivot routers are recorded in the Data SYN packet corresponding field. When the message reaches the destination, the replica host will answer with a matching Data SYNACK packet and sends it back pivot router by pivot router in the reverse direction to the receiver. As the SYNACK message passing through the pivot routers, each pivot router creates a local state to record that it is now part of the distribution tree.

When another request for the same data object hits a pivot router, it is checked whether it has matching state for the name of the object (object_id) and replica address, and the new SYN packet is not forwarded further to the replica destination. Instead, the pivot router adds the new requestor to its downstream branch, i.e creates a transient state and generates a matching SYNACK packet as if it held a copy of the requested data. During the data transfer phase, the pivot router caches the data segments while sending it to the first requestor so if another requestor wants the same segments then it will be transferred via the pivot router instead of going back to the source again. The data is cached by the router for a certain amount of time.

37
period before removing it due to the limited storage capacity of the router. Moreover, data flow will be forked at these points towards all the requestors [18].

![Figure 5.1: The SYN-SYNACK process that sets up the distribution tree](image)

### 5.1.5 Data transfer

When all the resolution and SYN processes are done, i.e. the receiver knows where to retrieve the data from and the distribution tree is created, the actual data transfer can be started with sending the first Data Request packet to the replica host (source). The Data Request packet contains information about the amount of bytes requested. The source replies the requested data with Data Segment messages.
Segmentation of the requested bytes are done at the source to fulfil the network layer’s maximum transmission unit (MTU) size requirements, hence avoiding any fragmentation of the reply packets.

The maximum amount of requested bytes one Data Segment packet could carry without exceeding the MTU size, excluding all the headers (NetInf TP header + underlying protocols header) is the Maximum Segment Size (MSS).

If the receiver in a Data Request packet asks for more bytes then the MSS, the source will reply with multiple Data Segment messages, sending the requested amount of bytes segment by segment. The receiver could requests either for the twice of the MSS, one MSS, or less then one MSS. The amount of bytes to request is defined by the congestion control mechanism.

The Data Request packets are first sent from the receiver terminal to their DR and forwarded along the distribution tree (created by the SYN-SYNACK process earlier) to the source. On the way, each pivot router records a request state additionally to the previous transient states. From the source side the matching data segment(s) are flowing the reverse direction towards the receiver(s). The data flow is forked at the nodes where the request state is set, but will not travel down branches that does not have a matching request state [18].

5.1.6 NetInf TP messages

The communication between NetInf nodes, how a terminal finds any named data object or how the data segments are requested, is based on NetInf TP messages. Also, the messages are understood by the NetInf capable nodes of the network, which means they can set certain states according to the information carried by the message packet. This has importance when a e.g. a pivot router is caching the object or forking the data flow towards multiple requestors.

Each phase of the protocol’s operation has its own message packets, which are the following:
Resolution Request - Resolution Reply

Data SYN - Data SYNACK

Data Request - Data Segment

Resolution Request/Reply operates during the first operation phase. It is used for locating a replica object. Table 5.1 shows the message fields of Resolution Request and Resolution Reply packets.

The Transaction ID (xid) field is a unique identifier for each pair of messages. It has to be a "hard to guess" random number. It is used for identifying and keeping track of the requests, as well as matching the reply messages. The main field of the Resolution Request and Reply message is the Object ID (obj_id) which holds the 16 bytes name of the data object, and used for locating the object.

When a NetInf capable router receives a Resolution Request message, the router checks whether it has knowledge about the requested object name, stored in the obj_id field. If it does not have, the router just passing the message further to other nodes. If it has, that router will answer with a Resolution Reply message, returning the host address holding the requested data in the field of Replica Address (replica_addr). Then the receiver will know where to retrieve the data from, and the next phase starts with sending the Data SYN message to the replica address.

Data SYN/SYNACK messages are used to create the distribution tree in the second phase of the protocol as it is described in 5.1.4. Syn messages are forwarded to the selected replica hopping zone by zone. Each zone records the passed pivot routers’ address to the pivot_routers message field of the SYN packet. The host holding the replica will create the SYNACK message and sends it back pivot router to pivot router, following the reverse path of the SYN message.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xid</td>
<td>uint32_t</td>
<td>Transaction id. Unique identifier for each pair of messages.</td>
</tr>
<tr>
<td>message_type</td>
<td>uint32_t</td>
<td>Type of the message (resolution_request/reply).</td>
</tr>
<tr>
<td>obj_id</td>
<td>uint8_t[16]</td>
<td>Name of the requested data object.</td>
</tr>
<tr>
<td>replica_addr</td>
<td>uint32_t</td>
<td>Address of the replica host (resolution_reply field only).</td>
</tr>
</tbody>
</table>

Table 5.1: Message fields of Resolution Request/Reply

The SYN/SYNACK messages have another additional field called expected_length which is the size of the entire data object in Bytes.

The structure of Data SYN and SYNACK packets with their message fields can be seen in table 5.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xid</td>
<td>uint32_t</td>
<td>Transaction id. Unique identifier for each pair of messages.</td>
</tr>
<tr>
<td>message_type</td>
<td>uint32_t</td>
<td>Type of the message (data_syn/synack).</td>
</tr>
<tr>
<td>obj_id</td>
<td>uint8_t[16]</td>
<td>Name of the requested data object.</td>
</tr>
<tr>
<td>replica_addr</td>
<td>uint32_t</td>
<td>Address of the replica host.</td>
</tr>
<tr>
<td>expected_length</td>
<td>uint32_t</td>
<td>Length of the data object to retrieve</td>
</tr>
<tr>
<td>pivot_routers</td>
<td>uint32_t[]</td>
<td>List of traversed pivot routers</td>
</tr>
</tbody>
</table>

Table 5.2: Message fields of Data_SYN/SYNACK
Data Request/Segment messages are sent during the data transfer phase. The receiver sets the amount of bytes and offset position to retrieve from the data object in the Data Request offset and length fields. The Replica host will reply the requested bytes in Data Segment messages, where the message field data is carrying the actual bytes of the object.

The Data Request/Segment messages with their additional fields are shown in table 5.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xid</td>
<td>uint32_t</td>
<td>Transaction id. Unique identifier for each pair of messages.</td>
</tr>
<tr>
<td>message_type</td>
<td>uint32_t</td>
<td>Type of the message (data_request/segment).</td>
</tr>
<tr>
<td>obj_id</td>
<td>uint8_t[16]</td>
<td>Name of the requested data object.</td>
</tr>
<tr>
<td>replica_addr</td>
<td>uint32_t</td>
<td>Address of the replica host.</td>
</tr>
<tr>
<td>expected_length</td>
<td>uint32_t</td>
<td>Length of the data object to retrieve.</td>
</tr>
<tr>
<td>offset</td>
<td>uint32_t</td>
<td>The offset position from where to request the current amount of bytes of data object.</td>
</tr>
<tr>
<td>length</td>
<td>uint32_t</td>
<td>The amount of bytes to retrieve from the data object.</td>
</tr>
<tr>
<td>data</td>
<td>uint8_t[]</td>
<td>Array of the actual data segment (Data_Segment field only).</td>
</tr>
</tbody>
</table>

Table 5.3: Message fields of Data_Request/Segment
Error messages

NetInf TP indicates errors also by sending messages. Each phase’s request message has its own response error message for reporting problems, which are the following:

- Resolution Request - Resolution Unreach
- Data SYN - Data Unreach
- Data Request - Data Request Unreach

All the error response messages has the same fields as their corresponding request messages but differs in the message type field, which contains the related error message name, e.g. RESOLUTION_UNREACH.

Resolution Unreach error message received when no routers has knowledge about the requested object name.

Data Unreach received when the there is no matching replica object available on the requested host.

Data Request Unreach Sent from the source back to the receiver when the request respect to the expected length or offset and requested bytes does not make sense, and the sender unable to respond with valid Data Segments.

5.1.7 Congestion Control

Congestion can create a lot of problems in a network as it does not block new connections but it also drops useful packets which flow in the network. As discussed in 3.1.1, a congestion control algorithm is used to avoid congestion in the network and controls the transmission of data after a packet loss or a timeout. Just like in previous TCP implementations, NetInf transport protocol also comprises of several phases in congestion control algorithm such as slow start, fast recovery phase and congestion avoidance phase.
In NetInf TP, the attribute values such as round trip time, congestion window and amount of outstanding data are maintained by the receiver [18]. During the slow start phase, multiple data requests are sent and single data reply is received resulting in an exponential growth of data in the network as more data is outstanding in the network [18]. Once a loss occurs, the phase changes to fast recovery in which requests cannot be made but the replies can arrive. This phase will end once the size of the outstanding data goes below the congestion window. At this point, congestion avoidance phase comes into the scene and take over from the recovery phase.

In the congestion avoidance phase, additive increase and multiplicative decrease (AIMD) concept is implemented similar to previous TCP implementations. It works by increasing the congestion window by a fixed amount in every round trip time (additive increase) and once a loss occurs, the window size is reduced to half (multiplicative decrease).

5.1.8 Retransmissions

The receiver requests data from the first till the last Byte of the object in multiple segments. The order of reception is of less importance. Once all bytes are requested, the received segments are counted. If there are holes in the received data, i.e there are missing segments, those segments will be re-requested in a cyclic fashion, till the received data object is completed [18].

Consider an example where a receiver requests data of length 300 bytes, for simplicity with a segment size of 8 Bytes. The first cycle of requesting is finished when all the 300 Bytes are requested. Assume that a data segment between offset position 42-50 bytes has not been received. The missing segment will again be requested resulting in the execution of a second cycle which will retransmit the missing segment.
Changes in the RTT

In case of TCP, the sending of the data takes place from the sending host to the receiving host and if the packet gets lost in between then the retransmission will again start from the same terminals which means that there are no changes in the round trip time (RTT) while the connection is established.

In case of NetInf during the data transfer new replicas can be created along the network. The source of the data retrieval in a retransmission cycle can be any of the new closest available replicas. For instance, consider that there are several routers in between receiver and source. The packet gets lost in between the first and the second router - counted form the receiver - but the transmission was successful till the second router and a new replica has been created there. In that case the retransmission wont starts again from the original source, but from the second router instead, which holds a new replica of the object.

This means that the RTT can vary when a new retransmission cycle starts as the missed segments are being collected from a closer replica [18].

5.1.9 Name-data integrity check

After the completion of the data transfer, the receiver performs a name-data integrity check meaning that the data which is received is compared with the one requested. Given that the name of the object is the hash of the data itself, it is done by re-calculating the hash value of the received object, and comparing it with the originally requested name. If both values are equal then the data is correct otherwise it is either being forged or attacked in the middle, therefore the entire received data must be discarded.

5.1.10 Security Considerations

In a network, security threats can occur by certain malicious users by either injecting bogus data in the network or causing denial of service (DOS) attacks by generating large volumes of TCP/IP traffic. The injection
of bogus data does not only generate bogus alarms but also depletes or consumes the already constrained resources of the intermediate nodes such that routers become busy in forwarding or processing the bogus data rather then on the actual meaningful packets. Also, DOS attacks creates an environment where network services becomes unavailable to its intended users as the attacker keeps the server busy by sending forged authentication requests. As the attacker uses fake addresses so server can’t find the actual address to send the response back.

To deal with such attacks, NetInf uses a parameter inside the messages and the parameter is called **Transaction ID (xid)**. The xid value is a random value chosen by the host who generates the request message and places this random value in the xid field of the request message. When this message traverses across the network and reaches the destination, the receiving host takes the xid value from the request message and puts it in the xid message field of the reply message. Once this reply message reaches back to the host (who generated the request), it will compare the xid fields of the response message with the xid field of the request message. If they are equal then it will generate further messages otherwise it will discard the reply message assuming that an attacker might have generated the message from between or some sort of forged message has came instead of an actual message.

Injection of bogus data is mainly a flood attack that misuses the properties of a TCP handshake process and thus creates a lot of fake pending connections. These pending connections becomes so much that the actual TCP resources becomes saturated with the attacker’s bogus connections and the victim then denies the legitimate connections thus denying service to the victim’s intended clients. A feature is then introduced in TCP known as **SYN cookie approach** to detect a SYN flood attack and prevents any bogus states along the path in a network.
5.2 NetInf TP and existing TCP implementations

While there are many differences between NetInf TP and legacy TCP models (TCP Tahoe, Reno, NewReno and SACK), there are similarities also exists between them. NetInf TP in its current form can be compared with the NewReno variant of TCP but there are some TCP SACK properties (knowledge of missing segments to the receiver) also exists. NetInf TP could implement more TCP CUBIC style features in order to make it more scalable and less sensitive to parameters such as round trip time (RTT).

5.2.1 Source and Receiver driven features

In TCP models, the sender side sends the data whereas the receiver responds with an acknowledgement. If the acknowledgement is missing the sender resends the data. This means that it is the sender side who takes decisions and drives the whole communication. In the case of NetInf TP, the receiver is sending requests for the data and the data reply itself is considered as the acknowledgement from the sender for the request. The receiver maintain the requests and replies and will send a request again if any segment is missing.

Figure 5.2 shows the different source and receiver driven features of TCP and NetInf TP.

Figure 5.2: NetInf TP: data acknowledges the request; TCP: receiver ACKs the data
5.2.2 Messages between source and receiver

TCP has a three way handshake mechanism to set up a connection before it starts sending data packets. This technique involves passing of several messages and ensuring the start of a TCP session between the hosts. During the process, the hosts who are attempting to build a connection with each other negotiate parameters related to the communication as shown in Figure 5.3.

In NetInf TP, there is no such connection establishment process. However, there are several messages which are passed before sending the data packets which includes finding the location of the host who holds the data along with synchronization and acknowledgement messages between hosts as discussed in 5.1.6. The exchange of messages is shown in Figure 5.4.

Figure 5.3: TCP three way handshake process
5.2.3 Fast recovery and Retransmission techniques

TCP uses fast recovery technique to recover from a loss which occurs either due to a segment lost in the middle or due to a timeout. This will avoid using slow start making a more appropriate calculation of the outstanding data. NetInf TP also uses similar recovery feature to recover from segment losses.

In TCP, fast recovery feature is combined with the fast retransmit mechanism which gives a much better utilization of the throughput such as if a packet is lost then retransmission of that packet will take place without waiting for the timer to be expired. This is not true in the case of NetInf
TP which retransmits the lost packets in a cyclic fashion and waits for one cycle to be finished before retransmitting packets in the second cycle upon request.

### 5.2.4 Loss detection mechanism

In NetInf TP, the detection of lost packets is similar to what is defined in TCP’s fast retransmit algorithm. It is just NetInf does not retransmit the packets immediately but the mechanism used is the same. In TCP, after receiving a small number of duplicate acknowledgements (preferably three) for the same TCP segment, the source concludes that a packet has been lost. Similarly in NetInf TP, if the same segment is received three times to the receiver then it will be considered as a loss packet as discussed in 6.5.7.

### 5.2.5 Summarized table

Table 5.4 describes the major differences between different TCP variants and NetInf in a summarized way.

<table>
<thead>
<tr>
<th>Features</th>
<th>Reno</th>
<th>NEWReno</th>
<th>SACK</th>
<th>CUBIC</th>
<th>NetInf TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source-driven</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>Receiver-driven</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>Connection establishment</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>Fast re-transmit</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>Fast recovery</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 5.4: Feature comparison between NetInf TP and TCP variants
Chapter 6

Transport protocol
Implementation

6.1 Prototype overview

We have implemented a prototype of NetInf TP with the primarily focus on to have a functional receiver driven transmission and congestion control mechanism with optimal resource utilization, hence several simplifications has been made on other functionalities of the overall design.

Even though NetInf is proposed as a future Internet solution, nevertheless it has to interact with the existing Internet architecture. NetInf TP in the current prototype is implemented on top of UDP to simplify the replica host addressing and terminal communications, however the protocol could operate directly over IP as well.

In OMNeT++, NetInf TP is realized as applications above the transport layer module of terminals. The receiver functionality is fulfilled by an application named NetInfApp1, while the source tasks are undertaken by NetInfApp2. Both applications are described in detail in section 6.2.
Network environment

The networking concept is simplified to a naive architecture where each phases of the NetInf TP communication happens between a receiver and a source terminal connected with regular IP routers, i.e. no NetInf capable routers are involved. In lack of NetInf enabled nodes, the source terminal takes all the different networking functionalities, such as being the replica host as well as the Designated Router. As in the simplified architecture there are no multiple replica sources, the testing of receiver-driven features are also limited to the congestion control mechanism.

Phases of operation

The following three main phases of operation are implemented in our prototype with the customized functionalities described below:

- phase I.: Locating replicas
- phase II.: Transport setup
- phase III.: Data transfer

Phase I.: Locating Replicas

In this phase Resolution Request packets, containing the name of requested data object are sent by NetInfApp1 from the receiver terminal to locate the replica objects.

The message is sent out to a predefined destination address which is normally the address of the Designated Router, but in our naive network implementation, which is used for testing the transport protocol, the destination address is the source terminal itself as no NetInf capable routers are involved in the current model.

On the destination terminal, NetInfApp2 will do a lookup operation on the queried data object’s name and if it finds any match, will respond with a Resolution Reply message including the Replica host’s IP address, which
happens to be the address of the responding source terminal itself in our simplified network case.

**Phase II.: Transport setup**

When the Resolution Reply message reached the receiver, NetInfApp1 processes the destination address for retrieving the replica object, and generates a *Data SYN* packet.

Conceptually in a complex network including NetInf capable routers, the SYN message would have been sent to the DR and forwarded zone by zone towards the destination address recording all the pivot routers passed along the path. In our simplified network model it is directly sent to the single source terminal.

NetInfApp2 on the source terminal looks up the requested object and generates the matching *Data SYNACK* packet. As no NetInf capable routers are involved, creating the distribution tree with setting any transient state are skipped at this point.

**Phase III.: Data transfer**

The processing of SYNACK message by NetInfApp1 on the receiver terminal launch the requesting procedure for all the segments of the data object.

The receiver’s NetInfApp1 application is responsible for detecting congestion events on the network and react accordingly with stop sending requests, then start requesting again when the network condition can allow that.

### 6.2 Applications

NetInf TP is realized as applications, namely *NetInfApp1* running on the receiver, *NetInfApp2* on the source terminals.

NetInfApp1 and NetInfApp2 are *simple module* applications written in C++. In case of NetInfApp1 and accordingly in case of NetInfApp2,
the application consists of the following source files: `NetInfApp1.cc`, `NetInfApp1.h` and `NetInfApp1.ned`, where the class `NetInfApp1` is defined in `NetInfApp1.h` header file, and `NetInfApp1.cc` describes the member functions. `NetInfApp1.ned` is the module definition for OMNeT++, including parameters and input, output gates definitions used for communication between network modules.

**NetInfApp1**

`NetInfApp1` is responsible for the main phases of NetInf TP’s operation including sending requests and providing congestion control mechanism for the transferring data. Figure 6.1 shows the flow-chart diagram of `NetInfApp1.cc`.

![NetInfApp1 Flow-chart Diagram](image)

**Figure 6.1: NetInfApp1.cc**
When a simulation starts and the network is configured, the initialize member function is called in each terminal. It does not only set the variables to their initial values, assigns parameters, binds the required ports and so on, it also schedules a self-message called *timerInit* to start up the application’s handleMessage function.

**HandleMessage**

The member function handleMessage is called when any message event occurs, which can be either a message arrived from any external source (Network Event), or a self message sent from the module itself like timerInit during initialization (startTime), or other scheduled self messages sent due to expired timers (Timeout Event).

The function first checks whether the arrived message is a self-message or not and reacts accordingly. If it is a self message (case 1.) Further sub-cases are investigated, as shown in figure 6.2. If the message arrived from an external source (case 2.) the packet is processed by the function *processPacket*.

![Figure 6.2: handleMessage function](image-url)
Case 1

- **Sub-case 1.1: timerInit**: The received self messages is the initializing message. Receiving the timerInit message calls **SendResRequest**, starting the resolution phase.

  **SendResRequest**: The function starts the first phase of NetInf TP operation, locating the replicas. The Resolution Request packet is sent out on the bound local port of UDP (port 4711). A timer (TimeoutResolutionRequest) is scheduled when sending the request. If no matching Resolution Reply message arrives in the given expiration time, the request is retransmitted with the same transaction ID (xid).

- **Sub-case 1.2: timeoutResReq**: The received self message indicates expired timers for sent Resolution Request messages. Receiving the time-out message calls the **RetransmitResoReq** function, which retransmits the Resolution Request with the same xid.

- **Sub-case 1.3: timeoutDatSyn**: The received self message indicates expired timers for sent Data SYN messages. Receiving the time-out message calls the **RetransmitDataSYN** function, which retransmits the Data SYN message with the same xid.

- **Sub-case 1.4: timeoutDataReq**: In the last case the incoming message must be a self message indicating expired timers for Data Request messages as no other self messages are used. The function **timeoutDataReq** is a more complex function, and it does not used for retransmitting the Data Requests, as a different retransmission strategy is applied. The function instead updates the timer fields recorded in the request entries.

Case 2
The `processPacket` function is called whenever any external message (coming from another module) hits the application. The function first identifies the arrival packet type, and calls the corresponding processing function. The further function calls can be followed through the call graph of `processPacket` in figure 6.3.

When a Resolution Reply message matching a Request has processed by the `processPacket` member function, `SendSynPacket` is called to create and send the Data SYN message, starting the second phase of the operation. The corresponding timer, `timeoutDataSyn` is scheduled.

When the matching Data SYNACK is processed, the `sendDataRequest(uint32 maxlen)` function is called, starting the data requesting procedure. The timer scheduled for each request is based on the actual RTT estimate.

```
Figure 6.3: processPacket call graph
```

`sendDataRequest` is the key function for the third, data transfer phase of operation, realizing the data requesting procedure. This function is called from several caller functions, which can be seen in figure 6.4.

```
Figure 6.4: sendDataRequest caller graph
```
NetInfApp2

NetInfApp2 is the source side application which provides all the response messages back to the receiver (NetInfApp1), as well as disposing the data object, and also responsible for the segmentation of the requested bytes into Data Segment reply packets.

The class NetInfApp2, similarly to NetInfApp1 consists of two major member function, initialize and handleMessage, the latter containing the processPacket function.

- The initialize function, in addition to setting variables or binding ports, is also responsible for creating the data object of a specific length. The length of the object is assigned from a parameter named "datsize", which is set in the omnetpp.ini configuration file.

- Unlike NetInfApp1, NetInfApp2’s handleMessage function does not deal with self-messages, hence only contains the processPacket function, receiving messages arriving from other network modules and provides the corresponding responses.

- The processPacket function, as in NetInfApp1, identifies the message type and calls the corresponding function to read the packet content and create the response. Figure 6.5 shows the call graph of NetInfApp2’s processPacket function.

Figure 6.5: NetInfApp2 - processPacket call graph
processResReq - lookup

After receiving the request, the processing function performs a lookup operation whether it has knowledge about the requested object name. If it has information, it calls the sendRplyPacket function that creates and sends the reply packet containing the IP address of the replica host.

processDataRequest - segmentation

Processing the Data Request message the requested length is checked, and if it is larger than the MaxSegmentSize then the function performs the segmentation of the reply into Data Segment packets.

The sendDataUnreach function creates and sends a response error message in case of the requested bytes exceeds the twice of the MSS size. The message contains the requested offset and length information of the faulty request.

6.3 Message definitions

NetInf TP message packets in OMNeT++ are implemented with using message definitions, as its described in section 4.1.2. The message packets are independent entities, i.e. they are not part of any applications. The terminals can create and send new instances of messages, access the message fields and read or write their contents.

The NetInf TP message packets are subclassed from Generic message, which contains general fields used by all NetInf TP messages. The inheritance diagram of Generic message is showed in Figure 6.6.

Figure 6.6: Generic message - Inheritance Diagram
Generic message

```c
packet Generic_message
{
    uint32_t xid;
    uint32_t message_type;
}
```
Source Code 6.1: Generic message

Resolution Request

```c
packet Resolution_request extends Generic_message
{
    uint8_t obj_id[16];    //Name of the object to query
    uint32_t binding_type; //Query type e.g. ANY
    uint32_t resolution_routers[];
}
```
Source Code 6.2: Resolution Request

Resolution Reply

```c
packet Resolution_reply extends Generic_message
{
    uint8_t obj_id[16];    //Name of queried object
    uint32_t binding_type; //NETINF_LOCATOR
    uint32_t host_addr;    //IP address of replica host
    uint32_t resolution_routers[];
}
```
Source Code 6.3: Resolution Reply
Resolution Unreach

```c
packet Resolution_unreach extends Generic_message
{
    uint8_t obj_id[16];          //Name of queried object
    uint32_t binding_type;

    uint32_t resolution_routers[];
}
```

Source Code 6.4: Resolution Unreach

Data SYN

```c
packet Data_syn extends Generic_message
{
    uint8_t obj_id[16];
    uint32_t expected_length;
    uint32_t replica_addr;

    uint32_t pivot_routers[];  //list of pivot routers
}
```

Source Code 6.5: Data SYN

Data SYNACK

```c
packet Data_synack extends Generic_message
{
    uint8_t obj_id[16];
    uint32_t expected_length;
    uint32_t replica_addr;

    uint32_t pivot_routers[];  //list of pivot routers
}
```

Source Code 6.6: Data SYNACK
**Data Unreach**

```c
packet Data_unreach extends Generic_message {
    uint8_t obj_id[16];
    uint32_t expected_length;
    uint32_t replica_addr;
}
```

Source Code 6.7: Data Unreach

**Data Request**

```c
packet Data_request extends Generic_message {
    uint8_t obj_id[16];
    uint32_t replica_addr;
    uint32_t offset;
    uint32_t length;
}
```

Source Code 6.8: Data Request

**Data Segment**

```c
packet Data_segment extends Generic_message {
    uint8_t obj_id[16];
    uint32_t replica_addr;
    uint32_t offset;
    uint32_t length;

    uint8_t data[];
}
```

Source Code 6.9: Data Segment
Data Request Unreach

```cpp
packet Data_request_unreach extends Generic_message
{
  uint8_t obj_id[16];
  uint32_t replica_addr;
  uint32_t offset;
  uint32_t length;
}
```

Source Code 6.10: Data Request Unreach

### 6.4 Listing Algorithms

During the data transfer, several messages are exchanged. Request packets sent earlier in time has to be matched with responses received later. Reply segments can arrive in different order and contain different portions of the requested bytes. Timers has to be cancelled when the corresponding replies arrive. To keep track of all these messages lists are used in the implementation.

There are two double linked lists maintained by the receiver which are named 'request' and 'receive' lists. These lists are the central point of the transport protocol implementation as beside keeping track of messages, they are also responsible for the retransmissions of missing segments.

#### 6.4.1 Request List

Request list stores all the request information are to be sent in the Data Request packets. The offset and length values are set in the list first and then assigned to the created Data Request message. Moreover, Request list records the sending time of the message (`send_time`) and sets a counter used for detecting packet loss (`delay_cntr`). Along with setting the values, it also stores the pointer associated with the timer (`Timeout-Data_Request`). Whenever a request is made, the timer is scheduled with an expiration time
based on the actual RTT estimate. The pointer has to be stored as the reply for the request can come in multiple segments and it has to be matched with the request. When all the expected segments of a request arrive, the corresponding timer has to be cancelled. The request entry will then be removed from the list. Source code 6.11 shows the data structure used for set the values in the Request list.

```c
struct Segment rqst;

rqst.xid = rand();
rqst.length = maxlen; // x*MSS where 0 < x < 2
rqst.offset = inreqlist->offset;
rqst.send_time = simTime();
rqst.timer = new cMessage("Timeout−Data Request");
rqst.delay_cntr = 0;
```

Source Code 6.11: Struct storing the request information

The request list is also used for managing the retransmissions of missing data segments. This does not require any separate algorithm as the list sequentially creates requests for the entries it has. The successfully received segments are removed from the list, hence anything remained has not been received yet. Once the last segments of the object is requested and there are still entries in the list, it will keep going sending request from the first remained element’s offset position and the process continues till the list becomes empty.

The steps of Request algorithm are shown in Algorithm 1.
Algorithm 1 Request algorithm

```cpp
iterator = requestList.begin()
assign values of rqst.elements
requestList.push_back(rqst)
iterator.offset += rqst.length
iterator.length -= rqst.length
```

The iterator set to the beginning of the list, where the first entry contains the overall bytes to request from the current offset position (initially [offset 0; Length 0-EOF]). Than the new request values are assigned and stored in the end of the list. The offset position will be updated with the length of current request. The overall length is subtracted by the size of current request. Therefore by the next request the fis list entry offset will point to the next offset position to retrieve bytes from and the overall length will show how many bytes are left from that offset position till the end of the file.

6.4.2 Received List

The purpose of Received list is to store the successfully received Data Segments. After receiving the entire data file, the list should contain only one entry with its offset and length values that will match the object’s start and end byte positions. As segments can arrive in arbitrary order and there could be missing segments which are received during the retransmissions, there is a merging algorithm implemented in the Received list. This merging algorithm merges the entries of the data segments received so that the entries are shrunk together whenever they are received. The list stores the received segment’s offset, length, and xid parameters. The merging results in coalescing the length of the segments with each other (making different segment entries as one) if the matching offset values are continuous.

This merging concept is a bit complex because received segments can come out of order and there are few different merging cases to be taken...
care of as the incoming segment can match with any of the segments in the received list. If the list is empty then the incoming segment will be placed in the list without looking at any condition as mentioned in algorithm 2.

**Algorithm 2** Implementation of received list - CASE I

if (iterator = rcvdlist.end()) then

rcvdlist.insert(rcvdsegment,rcvdlist.end()-1)

end if

Algorithm 3 refers to the condition when the incoming segment is be inserted to the left position of an already placed segment in the list.

**Algorithm 3** Implementation of received list - CASE II

if (rcvdsegment.length + rcvedsegment.offset < iterator.offset) then

rcvdlist.insert(rcvdsegment,rcvdlist.end()-1)

end if

The incoming segment can be merged with either to the left or right segment in the list. This depends on the size of the offset value of an incoming segment. Algorithm 4 defines the case when the incoming segment will be merged to the left segment in the list. Whereas algorithm 5 refers to the scenario when the incoming segment is getting merged to the right segment in the list.

**Algorithm 4** Implementation of received list - CASE III

if (rcvdsegment.offset <= iterator.offset) then

rcvdsegment.length = iterator.offset  rcvdsegment.offset
iterator.length = iterator.length + rcvdsegment.length
iterator.offset = rcvdsegment.offset
iterator.xid = rcvdsegment.xid

end if

66
Algorithm 5 Implementation of received list - CASE IV

if (rcvdsegment.offset <= iterator.length + iterator.offset) then

    newoffset = iterator.offset + iterator.length
    newlength = rcvdsegment.length (newoffset rcvdsegment.offset)
    newiterator = iterator
    newiterator = newiterator + 1

end if

Moreover, if the incoming segment is the successor to the already placed segment in the list then algorithm 6 will be valid.

Algorithm 6 Implementation of received list - CASE V

if (newiterator != rcvdlist.end() AND (rcvdsegment.offset + rcvdsegment.length) >= newiterator.offset) then

    newlength = newiterator.offset - newoffset
    iterator.length = iterator.length + (newlength + newiterator.length)
    iterator.xid = rcvdsegment.xid
    rcvdlist.erase(newiterator)

else

    iterator.length = iterator.length + newlength
    iterator.xid = rcvdsegment.xid

end if
6.5 Transport Protocol Algorithms

6.5.1 Principal terms and variables

MaxSegmentSize

Parameter representing Maximum Segment Size (MSS). Basically MSS is the maximum number of Bytes can be sent in a single Data_Segment message and it is derived from the MTU size by subtracting all the underlying protocols’ headers as shown in equation 6.1.

\[
\text{MaxSegmentSize} = \text{MTU size} (1500\text{bytes}) - \text{IP header} (20\text{bytes}) - \text{UDP header} (8\text{bytes}) - \text{NetInfTP header} (36\text{bytes})
\]

(6.1)

outstanding_data

Variable representing outstanding data, which is the amount of bytes presents in the network. Analogous to the term outstanding data used in TCP with the only difference that it is derived from the number of requested and received bytes on the receiver side.
MaxWindow

In its concept, similar to TCP’s congestion window. The size of MaxWindow over the value of outstanding data defines the amount of bytes a receiver can request by the time.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxWindow</td>
<td>uint32_t</td>
<td>-</td>
<td>variable</td>
</tr>
</tbody>
</table>

Table 6.3: MaxWindow

6.5.2 Updating the outstanding data

Whenever a request is sent, the outstanding data value is updated with adding the requested amount of bytes. Whenever a packet is received, matching a request sent from the receiver host, the received amount of bytes are subtracted from the outstanding data sum. In case of expired timers, the outstanding data value is also subtracted with the missed bytes as those bytes could be lost, hence are not present in the network. If the packet was only delayed, not lost, it is checked whether the lately arrived bytes were already counted, and the outstanding data is not subtracted again. Algorithm 7 shows the pseudo code of the updating process.

Algorithm 7 Updating outstanding data

if DataRequest(rqst) sent then
    outstanding_data+ = rqst.length
end if

if Data_segment(msg) received AND msg.timer = not expired then
    outstanding_data− = msg.length
else if timeoutDataReq(rqst) received then
    outstanding_data− = rqst.length
end if
6.5.3 Requesting process

Initially Data Request packets are sent to the source with requesting two times MSS length of data, which results in two Data Segment reply packets generated by NetInfApp2 at the source side, containing one MSS of data in each. When a Data Segment packet arrives to the receiver and all the carried information is processed by NetInfApp1, another request is sent for 2*MSS length of data from the next offset position as shown in figure 6.7.

As each arrival single Data Segment gains a new request of double MSS length, the amount of bytes outstanding in the network starts growing in an exponential fashion. This is a TCP like slow-start phase of the transport protocol in a receiver driven way.

NetInfApp1 on the receiver side will keep sending two times MSS data requests up on every arrival Data Segment, filling up the network with packets till the first packet loss detected. Then the congestion control mechanism takes care of the amount of requests, as described next.
6.5.4 Congestion control

NetInf TP follows the logic of TCP congestion control with implementing the phases of slow start, recovery and congestion avoidance.

NetInfApp1 maintains the NetInf TP’s congestion window, MaxWindow to control the number of requests as described below and showed in Algorithm 8

Control of requests by adjusting MaxWindow

- If the amount of bytes of a request added to the current outstanding data would exceed the actual MaxWindow size, no request will be sent.

- If there is a room for adding one segment length to outstanding data, below the limit of MaxWindow, a request with one MaxSegmentSize will be sent.

- If there is room for requesting two or even more segment, a request for 2 * MaxSegmentSize will be sent.

Algorithm 8 Request of next segments

\[
\begin{align*}
&\text{if } \text{outstanding data} + 2 \times \text{MaxSegmentSize} \leq \text{MaxWindow} \text{ then} \\
&\quad \text{sendDataRequest}(2 \times \text{MaxSegmentSize}) \\
&\text{else if } \text{outstanding data} + \text{MaxSegmentSize} \leq \text{MaxWindow} \text{ then} \\
&\quad \text{sendDataRequest}(\text{MaxSegmentSize}) \\
&\end{align*}
\]

Note that a request could never exceed 2*MSS, as that would result in a multiplied exponential growth which would be an enormous flood of packets. Hence, the source refuses any requests larger than 2*MSS, and sends an error message in response.

The way of achieving the exponential growth with the requesting approach is different than in TCP. Therefore the windows size in slow start is maintained with a slightly different logic as well.
The MaxWindow size is initialized to infinite (practically to an arbitrary vast value) letting the packets exponentially grow in slow start without any limit, till the first loss is detected, when MaxWindow is reduced to half of the current outstanding data and the protocol steps into the recovery phase.

Additive Increase Multiplicative Decrease

In the congestion avoidance phase the MaxWindow is being updated according to the Additive Increase Multiplicative Decrease (AIMD) algorithm.

In additive increase upon every successfully received segment the MaxWindow size is incremented with the fraction of the size of the received segment, as described in Algorithm 9.

Algorithm 9 MaxWindow - Additive Increase

Require: \texttt{NOTinSlowStart}

\begin{algorithmic}
\Function{MaxWindow}{Additive Increase}
\If{$0 < \text{outstanding data} \leq \text{MaxWindow}$}
\State $\text{MaxWindow} += \frac{\text{ReceivedSegSize} \times \text{MaxSegmentSize}}{\text{outstanding data}}$
\EndIf
\EndFunction
\end{algorithmic}

The amount of increment is calculated by the size of the received segment multiplied by the MaxSegmentSize, divided by the actual outstanding data.

The additive increase continues till the next congestion event, when the MaxWindow size is decremented to the half of the actual outstanding data. Algorithm 10 shows how the MaxWindow is reduced in multiplicative decrease.

Algorithm 10 MaxWindow - Multiplicative Decrease

\begin{algorithmic}
\Function{MaxWindow}{Multiplicative Decrease}
\If{$\text{loss_detected} = \text{TRUE}$}
\If{$\text{outstanding data} \leq \text{MaxWindow}$}
\State $\text{MaxWindow} = \text{outstanding data}/2$
\EndIf
\EndIf
\EndFunction
\end{algorithmic}

72
6.5.5 RTT estimate

The round trip estimate ($\text{rtt}$) and the mean deviation ($mdev$) are used to schedule timers for Data Request messages and calculated as known in case of TCP [22].

$$\text{rtt}_{\text{new}} \leftarrow \alpha \times \text{rtt}_{\text{old}} + (1 - \alpha) \times \Delta t \quad \alpha = \frac{7}{8}$$

$$mdev_{\text{new}} \leftarrow \beta \times mdev_{\text{old}} + (1 - \beta) \times \text{err} \quad \beta = \frac{3}{4}$$

Where $\alpha$ and $\beta$ are the gain constants ($0 < \alpha < 1; 0 < \beta < 1$) used for weighting the new estimate. $\Delta t$ is the actual measurement of RTT which is equal to the time elapsed between sending a Data Request and receiving the corresponding Segments. The error ($err$) is the difference between the $\text{rtt}$ estimate and $\Delta t$ measurement.

Using a large $\alpha$ and $\beta$ values gives more weight to the previous estimates in the calculation than to the current measurement. Therefore the timers based on the estimate $\text{rtt}$ and $mdev$ will be more smoothed, instead of changing continuously when the RTT varies. Algorithm 11 shows how the estimate calculated in the implementation.

**Algorithm 11 Calculation of the rtt estimate**

$$\Delta t = \text{receive\_time} - \text{send\_time}$$

if $\text{rtt} = 0$ then

$$\text{rtt} = \Delta t$$

$$mdev = \beta \times \Delta t$$

else

$$\text{err} = |\text{rtt} - \Delta t|$$

$$\text{rtt} = \alpha \times \text{rtt} + (1 - \alpha) \times \Delta t$$

$$mdev = \beta \times mdev + (1 - \beta) \times \text{err}$$

end if
In the calculation first $\Delta t$ is updated with the new measured value, which is the difference between received time of the received segment and send time of the request. In the initial case $rtt$ is equal to the measured $\Delta t$, as there are no previous estimates. In case of $mdev$ also only the measurement counted, as no error value collected till the first estimate is calculated. In any further cases the calculation of the round trip time estimate and mean deviation is according to the algorithm in equation 6.2

6.5.6 Scheduling timers

There are three types of timers implemented in NetInf TP. The types are based on the request message the timer is associated with. For Resolution Request and Data SYN messages a basic retransmit timer is scheduled.

**Timeout Resolution Request** For Resolution Request a constant timer is set with a value of 1 second when sending out the packet.

**Timeout Data SYN** Similarly to Timeout Resolution Request, when a Data SYN packet is sent, a timer is set with a constant 1 second expiration time. If the timer is expired, i.e. no matching Data SYNACK packet received, the Data SYN message is retransmitted.

**Time-out Data Request** The role of Data Request timers are more complex. First, as we seen earlier the outstanding data is updated based on the state of timers. The outstanding data is decremented if a Data Request timer expires, or incremented when a matching reply arrives that has an associated timer which is not expired. Second, the listing algorithms of NetInf TP are also relying on timers. The Request list keeps track of request and reply messages. When certain reply segments of a request are matched, the timer provides the information whether it can be removed from the list. The state of the associated timer is checked during the retransmission cycles as well.
Whenever a Data_request packet is created and sent out from the receiver host, a timer is set with a certain expiration time ($t_{\text{expire}}$). The value of $t_{\text{expire}}$ is calculated based on the current $\overline{\text{rtt}}$ and and $\text{mdev}$ values:

$$t_{\text{expire}} = \overline{\text{rtt}} + 4 \times \text{mdev}$$

(6.3)

Algorithm 12 Scheduling time-out for Data_request

```
if rtt=0 then
    t_expire = 1
else
    t_expire = rtt + 4*mdev
end if
```

schedule time-out for DataRequest at: current_simtime + t_expire

In the initial case a constant value is set for the expiration time, as no estimates are calculated yet. In any further cases the value is set according to the equation 6.3.

The Data Request timers, however are not explicitly used in NetInf TP for detecting the loss of the reply segment. NetInf TP implements a packet loss detection mechanism which is described in the next.

### 6.5.7 Detecting packet loss

Basically, an expired timer could mean three things.

- The packet got lost during the transmission.
- The packet is delayed.
- The timer gets expired before the packet could arrive, due to inaccurate $\overline{\text{rtt}}$ estimate.
In order to determine which packets are actually lost and not only delayed, NetInf TP uses a similar logic which is used in the Fast retransmit feature of the TCP models as discussed in 3.1.2 where after receiving a small number of duplicate acknowledgements (preferably three) for the same TCP segment, it is considered a loss.

The "duplicate ack" of NetInf TP is basically a counter that labels each request stored in the Request List. Whenever a segment arrives, the counter is incremented for all entry in the list. Normally, upon receiving the matching segments a request entry is removed from the list. If three further segments are received and the request entry is still on the list, therefore its counter reached three, the segment is considered loss.

Algorithm 13 describes the loss detection mechanism used in NetInf TP.

Algorithm 13 Detecting packet loss

1: $\text{MaxDuplicateAcks} = 3$
2: if $(\text{rcvdsegment.offset} \neq \text{element.offset})$ then
3: \hspace{1em} $\text{rcvelement}++$
4: end if
5: for $(\text{currelement} = \text{rqstlist.begin}(): \text{currelement}! = \text{rcvelement} : \text{currelement}++)$ do
6: \hspace{1em} $\text{currelement}++$
7: if $(\text{currelement.counter} = \text{MaxDuplicateAcks})$ then
8: \hspace{1em} $\text{loss_detected} = \text{true}$
9: end if
10: end for
11: if $(\text{loss_detected} = \text{true})$ then
12: for $(\text{currelement} = \text{rqstlist.begin}(): \text{currelement}! = \text{rqstlist.end}(): \text{currelement}++$) do
13: \hspace{1em} $\text{currelement} = \text{MaxDuplicateAcks}$
14: end for
15: end if
In the algorithm, the iterator `currentElement` runs through the request list until it reaches the entry which contains the received segment where another iterator `receiveElement` is pointing to. A boolean data type `loss_detect` is used for indicating whether the loss has been detected or not. Moreover, an integer `counter` is used which is incremented for every segment in the list as the iterator traverses along. If the counter value reaches to ’3’ then the segment is inferred as a loss packet.

6.5.8 Segmentation at the sender’s side

When the receiver makes a data request, the source treats the request in different ways depending on the size of the data segment. There are several cases which are examined by the source application (NetInfApp2) before sending the data segment to the receiver.

(i) If the length of the data segment requested by the receiver is equal or less then `MaxSegmentSize` then the source sends the data segment in one message which will have the same offset and length values as requested. Hence, no segmentation will take place as mentioned in the algorithm 14 where `datareqlen` is the length of the segment requested.

```
Algorithm 14 First case of source segmentation
if (datareqlen <= MaxSegmentSize) then
    Send data segment
end if
```

(ii) The second case is the segmentation case where one request message yields two separate reply messages. The segmentation is performed when the length of the requested segment is greater then `MaxSegmentSize`.

77
Algorithm 15 Second case of source segmentation

if (datareqlen > MaxSegmentSize) then
    len2 = datareqlen − MaxSegmentSize
    datareqlen = MaxSegmentSize
    Send data segment
    datareqoffset + = MaxSegmentSize
    datareqlen = len2
    Send data segment
end if

An example here will give more clarity about the mentioned two cases. For example, the receiver sends a request asking for a segment of length 2048 bytes from offset 0. The source will check if it is greater then MaxSegmentSize or not, if it is greater then segmentation will be performed and two replies will be sent. The first reply will have the offset value equals to 0 and length value equals to 1024. Similarly, the other message will have the offset value of 1024 and the length value of 1024 too. If we combine these two messages again then the offset and length values will be the same as received in the request message.
Chapter 7

Evaluation

This chapter evaluates NetInf TP in different performance aspects. First of all, section 7.1 *Single Flow Performance* analyses the protocol’s performance as a single flow by testing its intended behaviour via network simulations. Second, section 7.2 *Coexistence Performance* investigates how the developed transport protocol performs in a constrained network environment where the available resources are shared among simultaneous flows. As starting point we test pairs of homogeneous flows, than we investigate heterogeneous concurrent flows. In the later, NetInf TP competes with concurrent flows of different TCP variants and evaluated with respect to efficiency and TCP friendliness. By TCP friendly we mean that a flow with same initial conditions as well as equal round trip delay receives nearly equal share of the bandwidth as the competing TCP connections.

7.1 Single Flow Performance

The intended behaviour of the protocol is defined by the following metrics:

1. Data transfer
   - Data object is transferred completely (all bytes are received).
   - Retransmissions take place after the first cycle of requests.
   - Size of requests and number of reply packets are consequent.
2. Congestion control

- Fully utilized network resources.
  - Optimal window size operation compared to the Bandwidth-Delay Product.
  - Burst-less, continuous transmissions during congestion avoidance.
- Packet loss can be detected and reacted.
- Requesting adopts to the congestion control phases.

7.1.1 Method

A simple network topology is simulated with one NetInf TP source and a receiver terminal connected via regular IP routers. A bottleneck link is created in between the routers with the aim of experiencing “natural” packet drops during the data transfer. The used network parameter values, such as queue buffer capacity or link delay are arbitrary chosen but coordinated in order to realize loss. In order to analyse performance, the evolution of the congestion window and outstanding data is observed along with the received segments during a transmission.

7.1.2 Topology

**Network A:** Topology for a single NetInf flow (figure 7.1)

- One NetInf TP Receiver - one NetInf TP Source terminal
- Terminals are connected via regular IP routers
- Bottleneck link created in between routers

**Parameters**

- Receiver/Source link capacity: **100 Mbps** (full duplex)
Figure 7.1: Network A: Topology for a single NetInf flow

- Receiver/Source link delay: 0 (0.4 us)
- Bottleneck link capacity: 10 Mbps (full duplex)

1. Data transfer: **Single NetInf flow - 250KB-5ms-q5**
   - Data length: 250 KB
   - Bottleneck link delay: 5 ms (per direction)
   - Queue length: 5 (frame)

2. Data transfer: **Single NetInf flow - 3MB-5ms-q15**
   - Data length: 3 MB
   - Bottleneck link delay: 5 ms (per direction)
   - Queue length: 15 (frame)
7.1.3 Results

Figure 7.2 and 7.3 are results of the first data transfer. A simulation with 250 KB total length of transferred data in *Network A*, with a 5 ms bottleneck delay and a 5 frame queue buffer capacity.

By observing figure 7.2, the phases of congestion control can be followed. The square symbols are the outstanding bytes in the network connected with a solid line to make the changes of outstanding data upon requests or received segments easier to follow. The NetInf TP’s congestion window (MaxWindow) are the dots connected with a sample-hold line, drawing a boundary limit for requests on the plot. Expired timers of requests are indicated with "x". If the packet arrives later a "+" symbol indicates the delayed segment. Otherwise an "x" without any "+" symbol afterwards is a lost segment.

![Figure 7.2: Evolution of the MaxWindow (250KB-5ms-q5)](image)

In slow start the outstanding data grows exponentially till the first congestion is detected. Then the MaxWindow is reduced to half of the current outstanding data, to 19 386 bytes. This is followed by the recovery phase when earlier requested segments are either received or their corresponding timers are getting expired, decrementing the outstanding data
sum in both case. At 0.496 seconds the amount of bytes outstanding the network is already less than the MaxWindow, hence the receiver is able to make requests for data with a size of single MSS. The protocol steps into the congestion avoidance phase. The first successfully received segment starts the additive increase process, while the window size is incremented upon every further received segment. At 0.521 seconds there is a room for a request with a size of double MSS. The additive increase continues till the next congestion occurs, which further reduces the window size (multiplicative decrease). The new value of MaxWindow is 10 052 Bytes, which is close to the value of the Bandwidth-Delay Product (10Mbit/s * 10ms = 12500 Bytes). From 0.575 seconds, when the outstanding data is actually 11 488 bytes, the transmission is continuous till the object is completed. From 0.642 second, requests are made only for single MSS, however there is a room for requesting double MSS. This is due to that those requests are part of the retransmission cycle, which can be confirmed on figure 7.3.

![Requested and received segments](image)

Figure 7.3: Requested and received segments (250KB-5ms-q5)

By looking at 7.3, the requested and received segments can be followed. The square symbols on the left are the requests (usually in pairs, on top of
each other when the 2*MSS is requested) and the received segments are the
dots next to them further to the right (as received them in time). Expired
timers of requests are indicated with "x". The triangle represents the loss
detection signal.

On the right side, from 0.642 seconds the group of further requests and
received segments are the retransmissions of missing segments. The re-
requesting of missed segments are taken place directly after the last segment
requests of the 250 KB object. The missing segments are empty spaces in
the array of received segments. That means there is no received segment
dot in the horizontal line of a request square. When the corresponding
timer has expired an "x" (expired timer) symbol appears slightly next to
the empty space of a missing segment. At the end in the same horizontal
position (same offset in bytes) the segment is re-requested (a single square)
and finally received (a single dot).

Figures 7.4 and 7.5 are the results of the second data transfer. A
simulation with a longer, 3 MB of transferred data in Network A, with
a 5 ms bottleneck delay and a 15 frame queue buffer capacity.

![Figure 7.4: Evolution of the MaxWindow (3MB-5ms-q15)](image)

For simplicity, in figure 7.4 a solid line displays outstanding data only
and MaxWindow is again connected with a sample-hold line. In this longer data transfer the continuous saw-tooth behaviour of additive increase and multiplicative decrease can be observed. Apparently the first decrement of MaxWindow after slow start was not sufficient in terms of finding the optimal operation point, hence in a short while another network congestion occurred. The second decrement is followed by a continuous, burst-less requesting and receiving of segments in congestion avoidance phase till the transmission is finished.

Figure 7.5 shows the total number of bytes requested and received including the retransmissions at the end (the relevant symbols are the same as in figure 7.3). The requests and received segments are forming continuous and parallel lines showing the optimal transfer of requested bytes.

![Figure 7.5: Requested and received segments (3MB-5ms-q15)](image)

7.1.4 Summary

Based on the results above the following criteria of congestion control mechanism and data transfer has fulfilled:

- NetInf TP followed properly the different phases of congestion control such as slow start, recovery, and congestion avoidance.
The probing of available bandwidth via additive increase and multiplicative decrease was successful.

NetInf TP transferred the data entirely and responded to loss according to the retransmission strategy.

The results have been verified with different network parameters and data lengths as well. The further results were showing coherence, adapting to the given network conditions.

7.2 Coexistence Performance

The coexistence performance is measured based on the following metrics:

- Efficiency
  - The protocol achieves full utilization of the available resources
- TCP friendliness
  - The protocol receives nearly equal share of the link capacity

7.2.1 Method

A dumbbell topology is simulated as a test network with a bottleneck link shared among multiple simultaneous TCP and NetInf TP flows. That is two flows of TCP in parallel with two flows of NetInf TP. The sender and receiver link capacities are 1 Gb/s and the bottleneck link is set to 100 Mb/sec. The used network parameter values are arbitrary chosen, roughly representing a Gigabit Ethernet LAN - broadband WAN subscription scenario. The transmissions are starting at the same time and the round trip delay and MSS are equal for all the four flows.

To evaluate the transport protocol’s coexistence performance both a qualitative and a quantitative analyses is performed based on average received rate, received bytes and outstanding data measurements.
7.2.2 Topology

**Network B:** Topology for Multiple NetInf and TCP flows (figure 7.6)

- Two NetInf TP receiver - two NetInf TP source terminals
- Two TCP receiver - two TCP sender terminals
- Bottleneck link created via regular IP routers

![Network B: Topology for Multiple NetInf and TCP flows](image)

Figure 7.6: Network B: Topology for Multiple NetInf and TCP flows

### Parameters

- Data length: **30 MB** (for all transmissions)
- Transmission starting time: **0.4 s** (for all transmissions)
- Receiver/source link capacity: **1 Gb/s**
- Receiver/source link delay: **0.4 us**
- Bottle neck link capacity: **100 Mb/s**
- Bottleneck link delay: **2 ms** (in each direction)
- Queue length: **32** (frame)
7.2.3 Results of the intra-protocol analyses

In this simulation only two simultaneous flows are present at the time. First two parallel NetInf flows, than with the same initial conditions two TCP connections are tested.

Figure 7.7 shows the results of a simulation with two simultaneous NetInf flows. The light (green) solid line and dark (black) dashed line represents the two NetInf flows. Figure 7.8 shows the results of a simulation with two simultaneous TCP flows. The two TCP flows are marked with semi-dark (blue) dotted and light (purple) dotted lines.

In both cases the resources are shared completely ideally, with symmetric received rate graphs, almost overlapping received segments slopes. In case of NetInf, one can see as the two flows are taking turns in their window size operation, while in case of TCP the two flows window size are equal throughout.

Figure 7.7: Two NetInf flows

Figure 7.8: Two TCP flows
7.2.4 Results of the inter-protocol analyses

Test Cases

Three different cases have been analysed with the same topology based on the used TCP variants, which are TCP Reno, TCP New-Reno, and TCP Reno with SACK support.

- Case I.: TCP Reno
- Case II.: TCP New-Reno
- Case III.: TCP Reno + SACK

In each test case (a) Received Rate, (b) Received Segments and (c) Outstanding Data are analysed as coherent results of the same measurement.

By looking on the (average) received rate we can infer that how the available 100 Mb/s link capacity was shared in time among the flows. Similarly, the slope of received segments, i.e. received bytes over time gives a picture of rate based performance. By observing the evolution of outstanding data we can draw conclusions about the size of congestion window and therefore about the utilized resources, as in an ideal case with the same delay and mss values the same congestion window size results in equal share of the available bandwidth.

In all the following results the light (green) solid line and dark (black) dashed line represents the two NetInf flows. The two TCP flows are marked with semi-dark (blue) dotted and light (purple) dotted lines. The same results can be seen in large with detailed explanations in appendix A.1.
Case I. TCP Reno (figure 7.9); Initially the two flows of TCP having difficulties with effectively grabbing resources, however all the transmissions start in the same time. During the first two seconds the two NetInf flows are fully utilizing the bottleneck link capacity (a) as both the TCP flows are in an idle period after slow start (c). After two seconds the two TCP flows manage to restart and NetInf TP slows down (b). After a while the TCP flows are operating with nearly the same congestion window size as NetInf TP. From that time the link capacity is shared nearly equally among the flows. As both of the NetInf TP transmissions are finishing earlier - including retransmissions at the end - from nine seconds, the two TCP flows are utilizing the freed resources.

Case II. TCP New-Reno (figure 7.10); After the slow start periods of NetInf TP and TCP New-Reno, all the four flows are operating almost with the same rate (a), sharing the available resources equally. NetInf TP happens to be slightly more aggressive due to the vantage it could have
Case III TCP Reno + SACK (figure 7.11): One of the TCP flows has more aggressive start (a). In slow start its opening the congestion window high and it has a short recovery afterwards. The other TCP flow turns into idle state and only able to restart two seconds later (c). When the second TCP flow is back in operation the first TCP flow slows down. Thereafter all the four flows are being balanced in resource utilization. Close to ten seconds when all NetInf flows and the first TCP flows are finished, the second TCP flow is fully utilizing the bottleneck link capacity and finishes the transmission soon.

Fairness

A quantitative method is used on the same scenario as previously analysed to express the fairness ratio between NetInf TP and the tested TCP variants. The recorded received rate measurements are used to calculate the fairness index by applying the Raj Jain’s formula as discussed in section 3.3.2. The fairness index values are calculated for every sample measurement. Figure 7.12 shows the values for each sample in case of (a) TCP Reno, (b) TCP New-Reno and (c) TCP Reno + SACK. The lower sample numbers are
belonging to measurements from the beginning of the transmission. The higher sample numbers are from the end of transmission.

Figure 7.12: NetInf TP - TCP Fairness Index Values
7.2.5 Summary

The performance evaluation based on received rate measurements and the observed window evolution showed how the power relations were changing among NetInf TP and different TCP variants in the given constrained environment.

In case of TCP Reno, we could see that TCP was in a long idle period after slow start, which gave opportunity for NetInf TP to utilize more resources, resulting in a higher rate performance of NetInf TP during the transmission in overall.

In case of TCP New-Reno we could see that NetInf TP co-exists with TCP the most ideal way. The received rate and therefore the bandwidth utilization was close to an optimal sharing, which in case of 4 flows would be $100 \text{ [Mbit/s]} / 4 = 25 \text{ [Mbit/sec]}$. NetInf TP’s received rate was slightly higher than TCP’s rate, but the overall performance were smooth and even.

In the last case with TCP Reno using Selective Acknowledgement we could see that TCP could achieve even higher performance than NetInf TP but the idle period can still breaks TCP’s efficiency and overall NetInf TP was operating on a higher rate.

Using Raj Jain’s formula, we were able to quantitatively express the fairness ratio of NetInf TP to different TCP variants. In the evaluated scenario we could see that NetInf TP ensures fairness consistently with TCP New-Reno, however in case of TCP Reno with or without SACK support during the transmission’s initial and final periods the fairness index was lower.
Part III

Discussion and Future work
Chapter 8

Discussion

The overall focus of this thesis was on the basic transport operation of NetInf TP. Our primary objective was to assess whether the receiver-driven, message based concept is feasible on the level of a standard reliable transport protocol. That said, NetInf TP is not a basic transport protocol. In its concept, NetInf TP aims to address transport problems of information-centric networks. One of the main benefits of deploying NetInf TP would be integrating content caching directly in the data retrieval phase. This would require adding transport functionality to intermediary nodes of the network, which means a major paradigm shift from the end-to-end principle.

Prior to implementing such features, our philosophy was to ascertain that the main concept works as a standard reliable transport protocol. Specifically, the protocol should able to deliver the data correctly and efficiently as well as provide a mechanism to avoid congestion collapse. Additionally, the protocol should be fair with competing flows in terms of allocating the available resources. We could then incrementally add functionality to the protocol by using it in a full ICN context. Therefore we implemented a prototype, based on the information-centric properties, with basic functionality of the data transfer phase. That is in principle, a receiver-driven protocol, which operates with NetInf TP messages and provides TCP-like congestion control.
The messages are key to implementing features such as caching replicas or forking the data flows towards multiple requestors. As the messages are independent entities and can be understood by NetInf capable nodes, it is possible to extend the protocol operation to intermediate routers without making any changes to the end nodes. In other words, the messages concept allows an incremental development of the prototype to support information-centric features.

In the approach we took, we have placed a strong emphasize on congestion control. It would be possible, however, to address the transport problems of ICN in a different manner. In some fields, the term congestion is considered somewhat controversial with the ICN community as it is not clear whom we should be fair with or to. More radical still would not include congestion control at all, as in CCNx [8]. Others might consider adapting existing implementations, as in the paper of Information Centric Transport Protocol (ICTP) [38]. ICTP is based on CCNx, hence operates with “interest” packets. Also, it is a receiver-driven protocol, but implements the same algorithms of TCP. There is another similar, but earlier proposed receiver-driven ICN transport protocol, the Interest Control Protocol (ICP) [6]. The authors use an analytic model and developed a custom written simulator to show their fairness results. Their paper is described in more detail in section 2.2. In summary, we have chosen a more deployment-based approach to implement NetInf TP and considered that it will have to coexist with TCP.

A key point in this type of work is the receiver driven nature of the ICN based transport protocols. Classical throughput measures are more difficult to show as the receiver (and its perceived rate) determines the protocol performances and not source-to-receiver measures based on bits per second metrics. In other words, the metrics used with receiver driven protocols probably needs to be readdressed. Therefore, one point of discussion in our work is whether to choose a window-based or rate-based approach. For analyzing a single flow performance we have followed a similar approach as in
Fall and Floyd’s paper, *Simulation-based comparisons of Tahoe, Reno, and SACK TCP* [12]. They evaluated the performance of different TCP variants in ns-2. As we were using OMNeT++, we also considered the validation results of Andelin et al., presented in their paper, *Using OMNeT++ to Simulate TCP* [4]. A section describing both papers can be found in 2.2.

Beside the window-based method which is observing sent/received packets, as Fall and Floyd did, we visually inspected the evolution of the congestion window (MaxWindow). Comparing the MaxWindow with the Bandwidth-Delay Product also gave us a chance to observe the correct behavior of the protocol. For example to observe the success of bandwidth probing or verify that no inappropriate reduction of the MaxWindow occurred due to misidentified packet loss. We describe the common issues of detecting congestion events in section 6.5.7. However, at this point the protocol has not yet been simulated with concurrent flows. Also it is important to mention that the evaluation has been performed in low-latency networks with retrieval of small data objects. How NetInf TP would scale in long and variable RTT networks needs to be investigated. This is especially noteworthy due to the way we handle retransmissions in a cyclic manner (see section 5.1.8).

In case of multiple flows, we used a rate-based approach along with the window based visual inspections. We primarily considered the methodology of fairness measurements of *CUBIC: A New TCP-friendly high-speed TCP variant* [19]. We used a dumbbell topology with similar parameters but performed the evaluation based on received rate sampled measurements. Indeed throughput is another measure as used as a fairness metric in [19], but doesn’t necessarily reflect a receiver driven nature of an ICN compatible protocol. Also, we did use the rate-based approach but with the goal of showing equal rate allocation between multiple flows. We evaluated up to four simultaneous flows, however more should be used in order to draw valid conclusions for larger scale deployments.

From a pure experimental point of view we found simulation artifacts
that needed further investigation. For example Reno could experience long idle periods after a slow start, which made us believe that there was a problem in the simulator’s TCP implementation. Therefore we verified the TCP implementations against the RFCs, and saw that the idle behavior of TCP Reno is a known problem as described in RFC 2581. Therefore proper verification of the basic premises were done.

In the evaluated scenario the results showed that NetInf TP co-exists with TCP New-Reno as expected. However, there are some performance differences between NetInf-TP and TCP Reno with, or without SACK support. This difference is mainly due to the fact that NetInf-TP took the bandwidth whilst the TCP flows were in an idle period. We believe this is due to the shorter recovery times achieved by the cyclic retransmission strategy and the problems we reported earlier with TCP Reno (in OMNET and in the RFC). Again, as a point of discussion, the topic of receiver driven performance metrics should be explored in further detail. Given the experimental setup we used an analytic approach for ascertaining whether we were “fair” or not. The quantitative fairness analyses validated our assumptions regarding the resource sharing based on the performance results. In case of New-Reno, the fairness index was higher than in case of Reno or Reno+SACK.

As with all dynamical systems the initial conditions had a major impact on the performance and fairness results. This means that by starting the transmissions with different random seeds or varying the number of flows as well as the link delay and queue buffer parameters the “power relations” among the concurrent flows could be significantly changed. Therefore the conclusions drawn based on the evaluated scenario should not be considered as too general. However, during the congestion avoidance periods the flows aimed for balanced resource sharing, which has more representative implication in a real world coexistence scenario, where transmissions unlikely happen at the same time.
Chapter 9

Future Work

One should further investigate NetInf TP and TCP co-existence. First a more accurate assessment can be obtained by running real-world TCP implementations with NetInf-TCP. TCP implementations are available in the Network Simulation Cradle (NSC) as mentioned in section 4.2. Additionally, further fairness analysis could be carried out by running many simultaneous flows. To inspect the behavior of TCP and NetInf one could measure throughput at the intermediary network nodes as well as at the end-nodes. Metrics other than the received rate, such as time to transfer a chunk or object could be performed, thus considering the important delay aspect.

Secondly, in the current implementation, the protocol is running between single receiver and source terminals with no NetInf capable routers involved. The next major step would be to implement a NetInf capable router, that would allow on-path -network caching of data objects. Such would make it possible that data objects could be retrieved from multiple sources. However, that brings additional work to be done, as the Round Trip Time (RTT) could vary between the transmission cycles. Therefore one should consider to implement TCP CUBIC style features in the congestion control mechanism, so the transport protocol would be less sensitive to changes in the RTT.
Chapter 10

Conclusions

In this thesis, we have implemented the first prototype of NetInf TP, a receiver-driven transport protocol for the NetInf ICN architecture. The protocol was built by keeping the potential deployments in mind. The platform on which the protocol was developed and different test cases were experimented is OMNeT++. The evaluation was performed in different aspects with the aim of assessing the basic feasibility of the designed concept.

The developed prototype enables receiver-driven data retrieval between receiver and source terminals, with using a message based communication concept. The NetInf TP messages allows additional information-centric functions to be implemented such as dissemination and caching of data objects. For providing reliable data transfer, a well known TCP congestion control algorithm has been adapted. Additionally, a unique retransmission policy was applied to improve the protocol’s performance in congested scenarios.

Different analytical investigations were conducted on the operational and bandwidth sharing behavior of NetInf TP. The key point of our analysis is that the present form of NetInf TP is capable to operate effectively along with fair sharing of resources with TCP or other concurrent NetInf TP flows in a constrained environment.

We expect future investigations to be done at a larger scale and on real
platforms. On the basis of our results and conclusions, the experiments confirm the hypothesis on the feasibility of the proposed transport protocol. Thereby Alice and Bob’s next generation children have a new hope, that can make the Internet, a better future.
Appendix A

Further Results

A.1 Section 7.2.4 results in large

Figure A.1: Received Rate, Case I. (TCP Reno): Both TCP flows (bottom pair of dotted lines) having difficulties with starting and grabbing bandwidth, even though all the transmissions are started in the same time. Once the NetInf flows are out from the network, TCP flows are utilizing the freed resources.
Figure A.2: Received Bytes, Case I. (TCP Reno): NetInf and TCP flows from left to right in order as displayed in the legend. When the TCP flows managed to start (with effective operation), NetInf slows down. The more sparse sequence of points at the end of the two NetInf flows are the retransmissions. When NetInf flows are finished, TCP speeds up at the end with the same slope as NetInf initially.

Figure A.3: Outstanding data, Case I. (TCP Reno) close-up: Here we can see that all the flows begin at the same time but the two TCP flows could not start effectively and they are in idle for around 2 seconds. Once they are operating, the flows are taking turns equally in the number of bytes outstanding in the network.
Figure A.4: Outstanding data, Case I. (TCP Reno): The high perspective picture shows that when the NetInf flows are out from the network the TCP flows are widely increasing their congestion windows, and completing their data transfer. This is exactly the time when we could see TCP allocating the freed resources in figure A.1 or in figure A.2 that the slope of received bytes of TCP speeds up.

Figure A.5: Received Rate, Case III. (TCP New-Reno): The two TCP flows still could only start with utilizing bandwidth slightly later, but mostly we can see the fair sharing of the 100 Mb/sec link. All flows are consuming around 25 Mb/sec overall, which is full resource utilizing for the four simultaneous flows. NetInf TP still happens to be slightly more aggressive.
Figure A.6: **Received Bytes, Case III. (TCP New-Reno):** Almost completely parallel slopes of received bytes close to each other which also represents the nearly equal bandwidth sharing (received Bytes over time). The sightly faster slopes of the NetInf flows showing its higher throughput.

Figure A.7: **Outstanding data, Case III. (TCP New-Reno) close-up:** Again, more aggressive slow-start from TCP turning into long recovery, but later they both reach the common operation point and the flows are taking turns equally.
Figure A.8: **Outstanding data, Case III. (TCP New-Reno):** Equal operation in terms of congestion window and outstanding data, when NetInf TP finished the two TCP flows continue with larger cwnds.

Figure A.9: **Received Rate, Case II. (TCP Reno + SACK):** One of the TCP flows has a fairly aggressive start and managed to grab more resources than the two NetInf flows in the beginning. The other TCP flow still having difficulties with starting at all. At the end, from around 10 seconds, that TCP flow remains alone, hence fully utilizing the 100 Mb/s bottleneck capacity.
Figure A.10: Received Bytes, Case II. (TCP Reno + SACK): The slope of the initially aggressive TCP flow slows down later, and the late-starter TCP speeds up when the other transfers are completed.

Figure A.11: Outstanding data, Case II. (TCP Reno + SACK) close-up: In the close view we could see that actually both TCP flows are starting quite aggressively, opening their congestion windows large during the slow-start, but soon the second flow turns and remains in recovery for long time and only the first TCP flow along with the NetInf flows are in effective operation instead.
Figure A.12: Outstanding data, Case II. (TCP Reno + SACK): The higher perspective shows that after two seconds, all flows are in equal operation, each maintaining nearly equal size of congestion window. After 10 seconds when all the other concurrent flows are finished the delayed TCP flow opens its congestion window highly and finishes the transmission soon.
Bibliography


