BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2017 Bc. František Bureš

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS
ÚSTAV TELEKOMUNIKACÍ

NEW SIMULATION SCENARIOS IN NS3 ENVIRONMENT
NOVÉ LABORATORNÍ ÚLOHY V PROSTŘEDÍ NS3

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. František Bureš**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Jan Jeřábek, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. František Bureš ***ID:*** 154241
Ročník: 2 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Nové laboratorní úlohy v prostředí NS3

POKYNY PRO VYPRACOVÁNÍ:

Nastudujte problematiku základních komunikačních protokolů. Dále nastudujte možnosti různých simulačních a vývojových prostředí vhodných pro tyto účely a zaměřte se zejména na prostředí NS3. Navrhněte a popište dva komplexní scénáře, na kterých bude možné si vyzkoušet vlastnosti vybraných komunikačních protokolů a nastavovat různé atributy v různých situacích. Tyto scénáře budou vhodné jako laboratorní úlohy pro předměty zaměřené na komunikační technologie. Výstupem práce budou dva kompletní scénáře včetně podrobných návodů pro studenty, předpřipravených výchozích situací, doplňujících úkolů pro studenty a vzorového řešení. Předpokládá se, že délka realizace jedné úlohy bude přibližně 2 hodiny času.

DOPORUČENÁ LITERATURA:

[1] FOROUZAN, Behrouz A. TCP/IP protocol suite. 4th ed. Boston: McGraw-Hill Higher Education, 2010, xxxv, 979 s. ISBN 978-0-07-337604-2.

[2] JEŘÁBEK, J. Komunikační technologie. Skriptum FEKT Vysoké učení technické v Brně, 2016. s. 1-172.

Termín zadání: 1.2.2017 ***Termín odevzdání:*** 6.8.2017

Vedoucí práce: doc. Ing. Jan Jeřábek, Ph.D.
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRACT

The goal was to design two simulation scenarios in NS-3 environment. First scenario contains ARQ (Automatic Repeat Request) methods in TCP (Transmission Control Protocol). There is comparison of Stop-and-Wait, Go-Back-N and Selective-Repeat method. Theoretical part with TCP and ARQ is included. Second scenario is about ways of message transfer. Created scenario especially with Packet and Cell switching and theoretical basics are included. There is comparison of switching methods with different size of packet/cell, amount of nodes and latency impact in each method. Scenarios are in laboratory task form with instructions.

KEYWORDS

NS3, ARQ, TCP, Segment, Stop-and-Wait, Go-Back-N, Selective-Repeat, Packet switching, Cell switching, UDP, Latency

ABSTRAKT

Cílem bylo navrhnout dva simulační scénáře v prostředí NS-3. První scénář obsahuje ARQ (Automatic Repeat Request) metody v TCP (Transmission Control Protocol). Je v něm porovnání Stop-and-Wait, Go-Back-N a Selective-Repeat metod. Teoretická část obsahuje TCP a ARQ. Druhý scénář je o způsobech přenosu zpráv. Vytvořený scénář převážně s komutací paketů a buněk a teoretické základy jsou obsaženy v práci. Je v něm porovnání metod komutací s různou velikostí paketu/buňky, počtem uzlů a důsledek zpoždění v každé metodě. Scénáře jsou ve formě laboratorní úlohy s instrukcemi k vypracování.

KLÍČOVÁ SLOVA

NS3, ARQ, TCP, Segment, Stop-and-Wait, Go-Back-N, Selective-Repeat, Komutace paketů, Komutace buněk, UDP, Latence

BUREŠ, František *Nové laboratorní úlohy v prostředí NS3*: master's thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Ústav telekomunikací, 2017. 90 p. Supervised by doc. Ing. Jan Jeřábek, Ph.D.

DECLARATION

I declare that I have written my master's thesis on the theme of "Nové laboratorní úlohy v prostředí NS3" independently, under the guidance of the master's thesis supervisor and using the technical literature and other sources of information which are all quoted in the thesis and detailed in the list of literature at the end of the thesis.

As the author of the master's thesis I furthermore declare that, as regards the creation of this master's thesis, I have not infringed any copyright. In particular, I have not unlawfully encroached on anyone's personal and/or ownership rights and I am fully aware of the consequences in the case of breaking Regulation § 11 and the following of the Copyright Act No 121/2000 Sb., and of the rights related to intellectual property right and changes in some Acts (Intellectual Property Act) and formulated in later regulations, inclusive of the possible consequences resulting from the provisions of Criminal Act No 40/2009 Sb., Section 2, Head VI, Part 4.

Brno ....

(author's signature)

ACKNOWLEDGEMENT

I would like to thank my supervisor doc. Ing. Jan Jeřábek, Ph.D for expert advices, consultation, patience and contributing ideas for thesis.

Brno ....

(author's signature)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

ACKNOWLEDGEMENT

Research described in this master's thesis has been implemented in the laboratories supported byt the SIX project; reg. no. CZ.1.05/2.1.00/03.0072, operational program Výzkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .

(author's signature)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

CONTENTS

1 Introduction **13**

2 Network Simulator 3 (NS-3) **14**
2.1 History and previous versions 14
2.1.1 Ns-2 and comparation to ns-3 14
2.2 Basic objects and comparing to real devices and applications 14
2.2.1 Object Node 15
2.2.2 Application 15
2.2.3 Channel 15
2.2.4 Net device 15
2.2.5 Topology helpers 16
2.3 Documentation and other resources 16

3 Transmission Control Protocol (TCP) **17**
3.1 Services and features of TCP 17
3.2 Segment header of TCP 18
3.3 Connection-oriented behavior of TCP protocol 20
3.4 TCP send and receive windows 21
3.5 Flow control in TCP 23
3.6 TCP Error Control 24
3.7 Congestion Control in TCP 26
3.7.1 Congestion Window 26
3.7.2 Congestion Policy 26
3.8 Timers 29

4 Automatic Repeat Request (ARQ) **31**
4.1 Stop-and-Wait method 31
4.1.1 Numbering system 31
4.2 Go-Back-N method 32
4.2.1 Numbering system and windows 33
4.2.2 Windows and timers 33
4.3 Selective-Repeat method 34
4.3.1 Windows and timers 34

5 Ways of message transfer **35**
5.1 Types of message transfer and usage 35
5.1.1 Circuit switching 35
5.1.2 Message switching 35

5.1.3 Packet switching . . . 35
5.1.4 Cell switching . . . 36
5.2 Processing of message on interconnected device . . . 36
5.2.1 Store-and-Forward Switching . . . 37
5.2.2 Cut-Through switching . . . 37
5.3 Packet switching . . . 38
5.4 Cell switching . . . 38
5.4.1 Asynchronous Transfer Mode (ATM) . . . 39

6 Automatic Repeat Request (ARQ) scenario 41
6.1 Creating scenario in ns-3, used methods and issues . . . 41
6.2 Topology and scenario . . . 42
6.2.1 Implementing base of the scenario . . . 42
6.2.2 Simulation tasks . . . 46
6.2.3 Answers to control questions and assignment check . . . 62

7 Packet and cell switching scenario 64
7.1 Creating scenario in NS-3, used methods and issues . . . 64
7.2 Topology and scenario . . . 64
7.2.1 Implementing base of the scenario . . . 65
7.2.2 Simulation tasks . . . 70
7.2.3 Answers to control questions and assignment check . . . 81

8 Conclusion 84

Bibliography 85

List of symbols, physical constants and abbreviations 87

List of appendices 89

A Content of included CD 90

LIST OF FIGURES

3.1 Basic structure of TCP segment header 19
3.2 Establishing a connection using three-way handshake 21
3.3 Terminating a connection using four-way handshake 22
3.4 Send window structure with changing size options 22
3.5 Receive window structure with changing size options 23
3.6 Case of communication using TCP 24
3.7 Slow start mechanism with congestion window change 27
3.8 Congestion avoidance with additive increase of congestion window . . 28
3.9 Example of reactions after detecting congestion (Time out or 3 ACK) 29
4.1 Stop-and-Wait method: Solving problem with lost segment 32
4.2 Stop-and-Wait method: lost acknowledgement 32
4.3 Go-Back-N method: lost segment 33
4.4 Selective-Repeat method: lost segment 34
5.1 Comparing packet switching and cell switching transfer of data 36
5.2 Ethernet frame 37
5.3 IPv4 and IPv6 header 39
5.4 UNI ATM and NNI ATM cell 40
6.1 Topology of ARQ scenario 42
6.2 Set-up of configuration file 45
6.3 Wireshark output of TCP data transfer with default settings in NS-3 46
6.4 Transfer of data in Wireshark in default settings with loss 48
6.5 Retransmission of data in Wireshark in default settings 48
6.6 Transfer of data in Wireshark with Stop-and-Wait method simulated by receive window 1000 B 49
6.7 Transfer of data in Wireshark with Stop-and-Wait method simulated by receive window 1050 B 49
6.8 UDP communication between Node 0 and Node 2 53
6.9 Size of transmitted data with more nodes for 50000 B message 54
6.10 Time needed for transmission with more nodes for 50000 kB message 55
6.11 Size of all transmitted messages for different size of data 55
6.12 Time needed for transmission of different size of data 56
6.13 Resending same packet more times because of missing acknowledgement 57
6.14 Problem with error communication with smaller receiver window . . . 57
6.15 Example of lost packet - receiver side is missing one sent packet . . . 58
6.16 Example of lost packet - sender did not receive Ack to one packet and has to resent it again 58

6.17 Example of lost packet - receiver received all packets and sent Ack packet to sender . . . 59
6.18 Example of lost packet - sender did not receive Ack for one packet . . 59
6.19 Example of changing waiting time for resending packet . . . 59
6.20 Impact of packet error rate on time needed for transmission of 100000 B with different ARQ methods, percentage increase of needed time in comparison to transmission time without errors . . . 60
6.21 Impact of packet error rate on transmitted data needed for transmission of 100000 B with different ARQ methods, percentage increase of needed data for trasmission in comparison to data for transmission without errors . . . 61
7.1 Topology of packet and cell switching scenario . . . 64
7.2 Set-up of configuration file . . . 69
7.3 Console output of UDP ECHO communication . . . 69
7.4 Packet fragmentation with packet size bigger than MTU . . . 70
7.5 Message is fragmented to more packets with MTU 70B . . . 72
7.6 Comparison of sent headers size for 2048 B message with packet switching with different values of MTU and cell switching . . . 73
7.7 Animation of message transfer in NetAnim . . . 74
7.8 Latency of sending UDP echo request until receiving echo reply with different number of nodes . . . 75
7.9 Wireshark output with UDP echo messages on Node 1 interface towards Node 0 . . . 76
7.10 Wireshark output with UDP echo messages on Node 1 interface towards Node 2 . . . 76
7.11 Console output with 2 UDP echo messages transmitted via 1 way between Node 0 and Node 2 . . . 79
7.12 NetAnim output of load balanced UDP messages between Node 0 and Node 2 . . . 80
7.13 Console output with 2 UDP echo messages transmitted each message different way between Node 0 and Node 2 . . . 80
7.14 Comparison of needed time for transmission of two 2048 B messages with and without load balancing . . . 81

LIST OF TABLES

3.1 Frequently used well-known TCP ports 17

1 INTRODUCTION

Simulation environments are very useful tool not only on informatics field. With simulation software, we can test real situations without need of buying real devices, which is appreciated not only by students but experienced professionals too. In this project, open-source complex network simulator NS-3 was used. The goal of this project is to design two scenarios in simulation environment.

Reliability in transfer is a crucial part of data transfer, especially in packet networks. Basic ARQ (Automatic Repeat Request) mechanisms are Stop-and-Wait, Go-Back-N and Selective-Repeat. These mechanisms are used in protocols that provides flow control, error control and congestion control. In wire networks, these mechanisms are used on transport layer of ISO/OSI (International Standards Organization / Open System Interconnection) model. TCP (Transmission Control Protocol), as a reliable and base protocol of transport layer, uses these mechanisms. In wireless networks, ARQ mechanisms are used even on link layer.

First scenario contains ARQ methods used in TCP protocol on specific examples and comparing methods between each other. Scenario contains also comparing TCP (with different ARQ method) and UDP transmission. Crucial behavior in error communication is also simulated. Creating of scenario in NS-3 includes nodes creation, generation of communication and static routing.

There are some basic ways of message transfer, which usually depends on kind of signal. Transfer of voice has different signs than transfer of data. For example, Circuit switching is used in telephone networks and Packet switching is the most common way in data networks these days. Process of receiving, storing and forwarding of message can have different procedures. Cell switching, used in ATM technology, with small exactly defined size of message parts brings possibility of interesting comparison with Packet switching behavioral.

Second scenario contains comparing message transfer methods by changing network attributes. Scenario is mostly focused on Packet switching and Cell switching as very popular switching techniques. There is comparison of impact of different size of packet and cell. Latency comparison in switching methods with more nodes is also included.

2 NETWORK SIMULATOR 3 (NS-3)

The ns-3 simulator is an open-source discrete-event network simulator mainly used for research or education. The project offers an open environment for developing and sharing of own project. There are basically implemented models of how computer networks work. Simulation engine provides environment for various simulation scenarios. Ns-3 environment brings possibility of studying system behavior and detailed observing of how networks work in highly controlled, reproducible environment. Ns-3 is created as a set of libraries with possibility of combining them, even with external libraries, together. Linux systems are primary platform for ns-3 and many graphical programs offer output view such as Wireshark and Netanim. Sophisticated wireless models, especially LTE (Long-Term Evolution) and WiFi (Wireless Fidelity), create a very popular areas of research [1].

2.1 History and previous versions

The first version was known as ns-1 [2]. Ns-1 was developed at Lawrence Berkeley National Laboratory in 1990s. The core of the simulator was written in C++, with Tcl-based scripting of simulation scenarios.

2.1.1 Ns-2 and comparation to ns-3

Ns-2 [3] was a very popular tool that preceded ns-3. Unlike ns-3, which is entirely written in C++ with optional Python bindings, some components are written in OTcl and others in C++. Ns-3 is not a backwards-compatible extension of ns-2. Ns-3 is actively maintained and provides some features not available previously, but ns-2 has more modules owing to its long history. Some limitation, such as supporting multiple types of interfaces on nodes, was fixed by lower base level of abstraction in ns-3.

2.2 Basic objects and comparing to real devices and applications

In this section, there will be comparison of real terms and operation in computer networking with abstraction used in ns-3.

2.2.1 Object Node

A host or end system is a name for device that connects to a network, when we talk about Internet. Because ns-3 is not only an Internet simulator, it was decided that it should have a more specific name - the node [4].

The node is a basic computing device abstraction, which is represented in C++ by the class `Node`. The class `Node` provides options for maintaining of computing devices in simulation environment. Basically, node represents a computer to which you will add functionality as a protocol stack and peripheral cards.

2.2.2 Application

Typically, the line of separation between system software and application software is done by privilege level change that happens in operating system. In ns-3, there is no real concept of operation system and privilege levels of system calls. Ns-3 applications [4] run on ns-3 nodes to simulate applications in simulated environment.

The abstraction is represented in C++ by the class `Application` and it is the basic abstraction for a user program that generates some activity. The class `Application` provides option of our version of user-level application in simulated environment. Example of class `Application` is `UdpEchoClientApplication` and `UdpEchoServerApplication`, which is used to generated echo packets in client/server communication.

2.2.3 Channel

When you connect a computer to a network in the real world, medium over which data flows is called channel. In ns-3 simulated environment, `Node` can be connected to an object representing communication channel.

The abstraction is represented in C++ by the class `Channel` [4] and offers methods for managing communication objects and connecting nodes to them. The channel can be represented as simple as a wire but it can also represent a large Ethernet switch.

Frequently used versions of the `Channel` are `CsmaChannel`, `PointToPointChannel` and `WifiChannel`.

2.2.4 Net device

When you want to connect a computer to a network, it has to have a peripheral card with networking function and then the card is called NIC (Network Interface Card). A NIC will not work without a driver to control the hardware.

The abstraction is represented in C++ by the class `NetDevice` [4] and it covers both software driver and the simulated hardware. A net device is installed in a `Node` to enable communication with other `Nodes` over `Channels`.

There are more versions of `NetDevice` such as `CsmaNetDevice`, `PointToPointNetDevice` and `WifiNetDevice`. Specific type of `Netdevice` is designed to work with the `Channel` with the same specification, for example `CsmaNetDevice` should operate with `CsmaChannel`.

2.2.5 Topology helpers

In large simulated network, you have to connect `NetDevices` to `Nodes`, `Netdevices` to `Channels`, assigning IP addresses, etc. Topology helpers [4] provide as easy management as possible. For example, with topology helpers, there is no need to add a MAC address to a `NetDevice` and configure the node's protocol stack.

2.3 Documentation and other resources

Ns-3 as a community project relies on contributions of community to develop new models, maintain existing ones and share results. Basic information can be found on official website `https://www.nsnam.org/` and more detailed in tutorials for specific release such as `https://www.nsnam.org/docs/release/3.21/tutorial/ns-3-tutorial.pdf` . Detailed specification of used classes and models is located in ns-3 documentation maintained using Doxygen `https://www.nsnam.org/doxygen/index.html` and in ns-3 wiki page `https://www.nsnam.org/wiki/Main_Page` .

3 TRANSMISSION CONTROL PROTOCOL (TCP)

Transmission Control Protocol (TCP) [5] is one of the main protocols on transport layer in ISO/OSI (International Standards Organization / Open System Interconnection) model. TCP is used by many application protocols. User Datagram Protocol (UDP) is considered as another important protocol on transport layer with some different features and services, which are often compared with TCP.

3.1 Services and features of TCP

Process-to-process communication
Protocol of transport layer in ISO/OSI model provides process-to-process communication. Running program on application layer in ISO/OSI model, which uses transport layer, is called a process.

Port numbers are used for definition of process. In communication on transport layer, socket address are used for definition of client and server process. Socket addresses include both IP address and port number. Port numbers are divided into three parts [6]:

- Well-known (0 - 1023), the most used are listed in Tab. 3.1
- Registered (1024 - 49151)
- Dynamic or private (49152 - 65535)

Tab. 3.1: Frequently used well-known TCP ports

Port number	Protocol	Application
23	TCP	Telnet
25	TCP	Simple Mail Transfer Protocol (SMTP)
53	TCP, UDP	Domain Name System (DNS)
80	TCP	Hypertext Transfer Protocol (HTTP)
110	TCP	Post Office Protocol version 3 (POP3)
443	TCP	Hypertext Transfer Protocol Secure (HTTPS)

Stream delivery service
TCP allows sending and receiving processes to deliver and obtain data as a stream of bytes. Sending and receiving processes do not always have the same write or read data rate. Because of the difference in data rates, there is a sending and receiving buffer for each direction [6].

Full-Duplex Communication
In TCP, data can be sent in both directions at the same time [7].

Multiplex and Demultiplex
There are input and output queues, separated for each application, which can be sorted and then multiplexed or demultiplexed [7].

Connection-Oriented protocol
TCP is connection-oriented protocol [6] with three phases:

- Connection establishment (three-way handshake)
- Data transfer
- Connection termination (four-way handshake)

Sequence and acknowledgement number
In segment header, there is a sequence number and acknowledgement number [7]. Sequence number is for numbering each segment for correct order at receiver side. Acknowledgement number defines the number of the next byte, which should be received.

Flow control
The receive side controls with sequence and acknowledgement number how much data can be sent by the sending side because of overwhelming [5], more details can be found in section Flow control.

Error Control
TCP implements a byte-oriented error control mechanism [6] , more details can be found in section Error control.

Congestion Control
Data transfer can be controlled by receiver (flow control), but there is still possibility of congestion, which is limited by congestion control mechanism [6], more details can be found in section Congestion control.

3.2 Segment header of TCP

TCP segment consists of header (20 to 60 bytes) and data. TCP header parts [5] with size are shown in Fig.3.1 with meaning below.

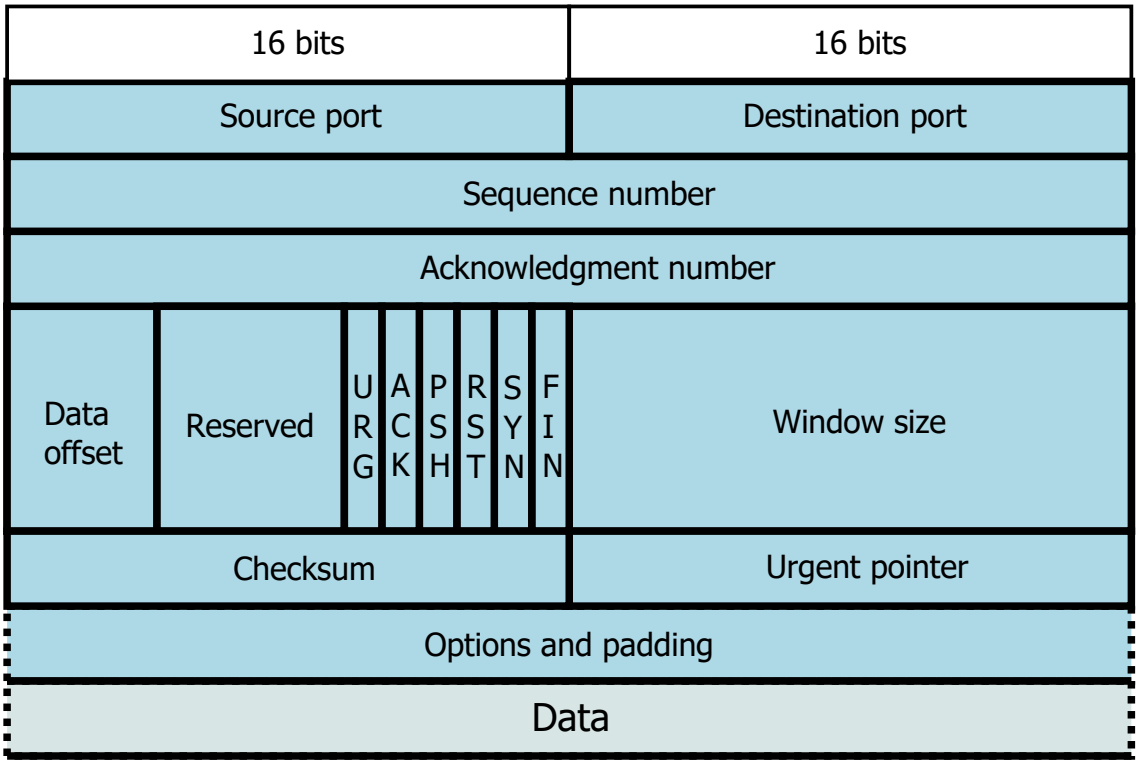


Fig. 3.1: Basic structure of TCP segment header

Source port
Port number on the sender side , same as the source port in the UDP header.

Destination port
Port number on the receiver side , same as the destination port in the UDP header.

Sequence number
Number assigned to the first byte of data in this segment. Each side uses a random number during connection establishment as a initial sequence number.

Acknowledgement number
Byte number that receiver side is expecting from the sender side (number of the last received byte + 1).

Data offset
This indicates where the data begins. The TCP header do not have exact length (variable length of the field Options).

Reserved
Reserved for future use (must be 0).

Flags
6 bit field with options 0 or 1 for each functionality.

- **URG** - Urgent pointer is valid
- **ACK** - Acknowledgement is valid (acknowledgement of received data)
- **PSH** - Request for push
- **RST** - Reset the connection
- **SYN** - Synchronize sequence numbers (used in connection establishment)
- **FIN** - No more data from sender

Window size
Number of maximum bytes, which can by send by sender side without acknowledgement from receiver.

Checksum
It is used to protect the entire TCP segment against errors in transmission and errors in delivery.

Urgent pointer
Value of urgent pointer as a positive offset from the sequence number in segment. Only used if the URG bit is set to 1 in the Flags field.

Options and padding
There can be optional information up to 40 bytes. There are options such as maximum segment size, window scale and Selective Acknowledgement, which is used in Selective repeat ARQ (Automatic Repeat Request) method.

3.3 Connection-oriented behavior of TCP protocol

Protocol TCP is connection-oriented [6], which means that it establishes a virtual path between source and destination side, transfer data and terminates connection.

Before data transfer, there is a three-way handshake for creating a connection (Fig. 3.2). The side, which wants to open a connection, sends a segment with random initial sequence number and SYN Flag is set. The other side responds with segment, where SYN and ACK flags bits are set. The initial random sequence

number and acknowledgement number, which is received sequence number + 1, are set. After the segment is received, acknowledgement segment is send with ACK flag bit set. Sequence number is the same as in the first segment. Acknowledgement number is received sequence number + 1. After these three steps, connection is opened.

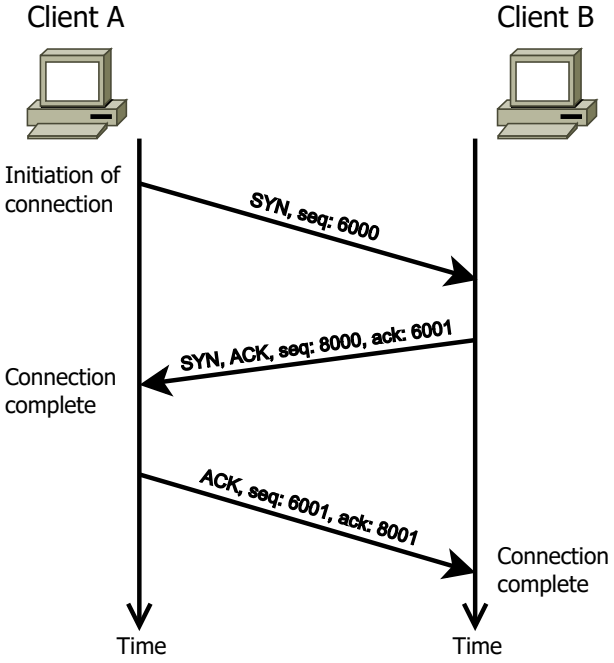


Fig. 3.2: Establishing a connection using three-way handshake

After connection is established, data transfer can proceed. Sequence and acknowledgement numbers are changed according same logic as in establishing the connection, the numbers are increased by the size of the data.

Four-way handshake between two sides terminates the connection (Fig. 3.3). The connection is terminated separately in each side. First side sends segment with FIN flag set. Other side sends one acknowledgement segment with ACK flag set and one termination segment with FIN flag set. After the first side receives two previous segments, acknowledgement segment with ACK flag bit set terminates the connection.

3.4 TCP send and receive windows

Windows in TCP are crucial in features such as flow or congestion control mentioned in separated sections. TCP uses two windows for each direction of data transfer:

- Send window
- Receive window

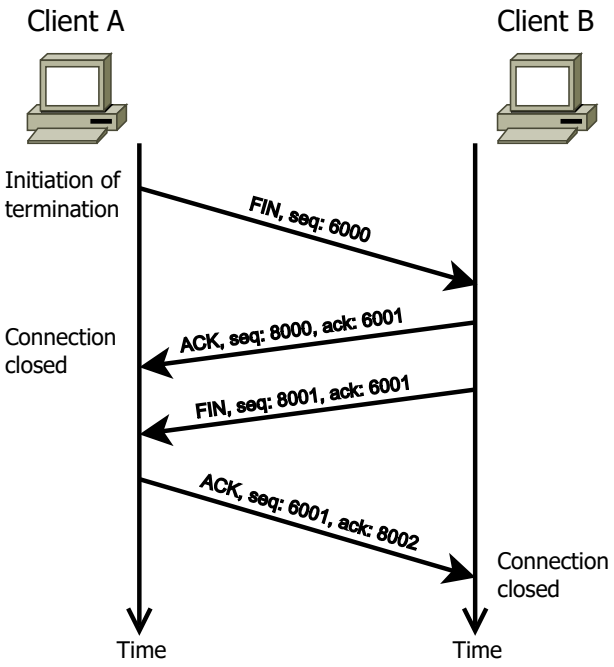


Fig. 3.3: Terminating a connection using four-way handshake

Send window

The window [6] size is changed by the receiver as a flow control and by congestion control mechanism. Send window opens, closes or shrinks. In send window, there is a usable window part and the part, where are bytes waiting to be acknowledged. The window size is usually thousands of bytes. Timer is another important part in send window function and will be discussed in section Timers. Example of send window is shown in Fig. 3.4.

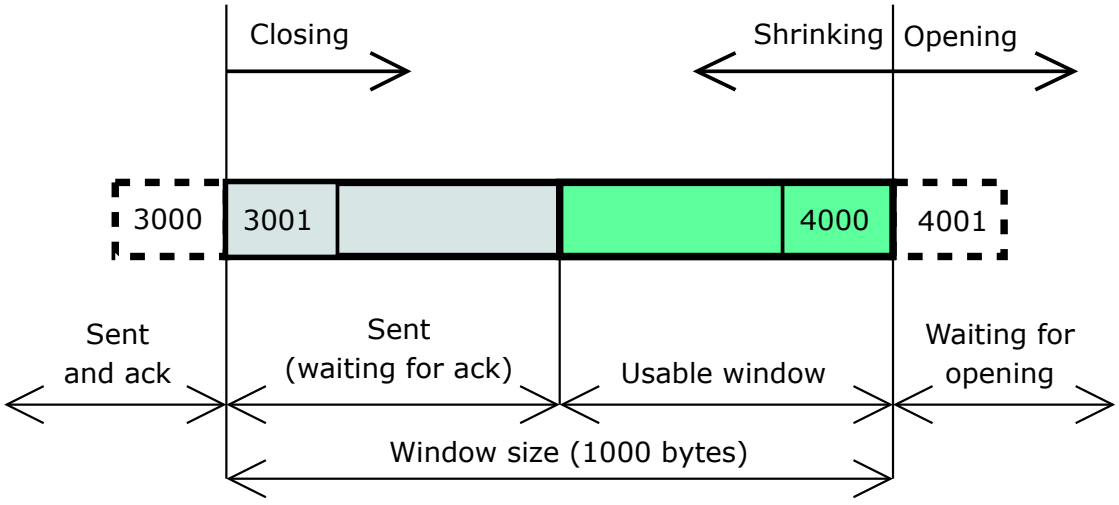


Fig. 3.4: Send window structure with changing size options

Receive window

The receiver window [6] size defines how many bytes can be received from the sender before being overwhelmed, this process is called flow control. TCP uses by default cumulative acknowledgement by announcing the next expected byte to receive. Selective acknowledgements can be set in TCP header options. The window size is usually thousands of bytes. Example of receive window is shown in Fig. 3.5.

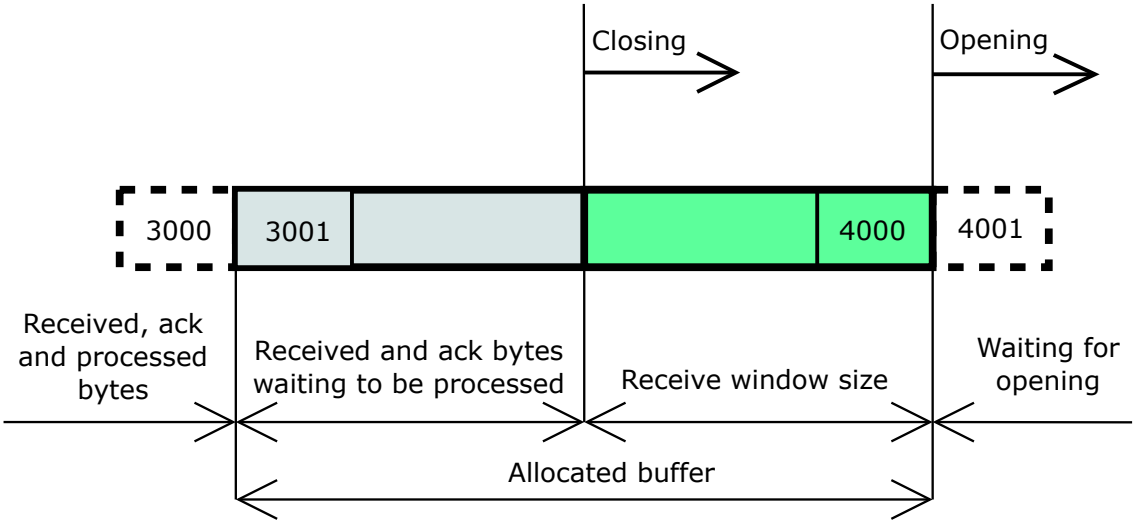


Fig. 3.5: Receive window structure with changing size options

3.5 Flow control in TCP

TCP flow control [5] is separated from TCP error control, which is discussed in the next section and ignored, because of simplicity, in this section. Data originated from the sending process on application layer proceed to the sending transport layer, where is TCP, and eventually travel to the receiving transport layer and finally to the receiving process.

Flow control feedback is provided from receiver TCP to the sender TCP and from the sender TCP to the sender process. The opening, closing and shrinking of the send window is controlled by the receiver (Fig. 3.4).

Scenario of how the receive and send windows are changing is shown in Fig. 3.6. For simplicity, there is a transfer of data only from one side (client/server model) and no errors or losses. Consumed data are the data, which were received and consumed by receiver process on application layer.

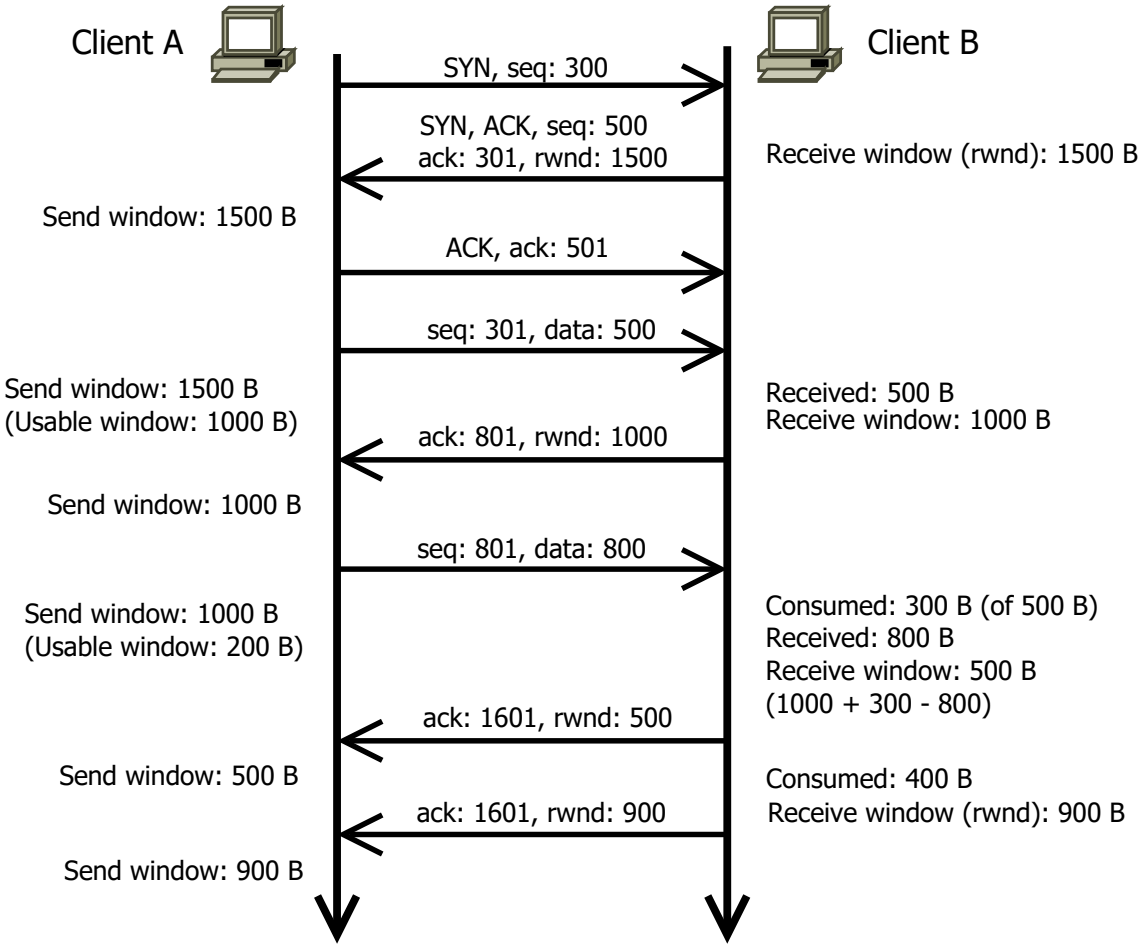


Fig. 3.6: Case of communication using TCP

3.6 TCP Error Control

TCP as a reliable protocol is responsible for delivering entire ordered data to the receiver application program without errors, loss or duplicates [6]. To achieve this, TCP error control uses three mechanisms:

- Checksum
- Acknowledgment
- Time-out

Checksum

TCP header of each segment contains checksum field [7]. It allows certain control of transfer without errors on transport layer. TCP header, data and part of IP header (a pseudo-header) are used for calculation. The pseudo-header included in checksum ensures that the IP header is not corrupted and it will be delivered to correct host. Checksum is implemented in UDP too.

Acknowledgement

Acknowledgement [5] confirms receiving of data. Acknowledgement segments do not consume sequence numbers and are not acknowledged. TCP usually used cumulative acknowledgement, but selective acknowledgement is possible and used in some TCP implementations.

Cumulative acknowledgement [6] was the original acknowledgement mechanism in TCP and advertise the next byte expected to receive (it ignores received out of order segments). There is no feedback of discarded, lost or duplicated segments. The Acknowledgement number field in TCP header is used.

Selective acknowledgement (SACK) [5] provides additional information from receiver to sender. Informations contains a block of received duplicated segments and a block of out of order data. Option field in TCP header is used for implementation of SACK.

Acknowledgements are generated in specific occasions with specific rules [6], some of the most used are:

- When a data segment is sent, an acknowledgement with next sequence number expected to receive must be included for reduction of traffic.
- The receiver delays sending of acknowledgement (usually 500 ms) if there is no data to send, receives an in-order segment and previous segment has been acknowledged (reduction of acknowledgement traffic).
- At the receiver part, there should be maximum of two in-order unacknowledged segments at any time for preventing unnecessary retransmission and eventual congestion in the network.
- With received out-of-order segment (higher sequence number), acknowledgement must be immediately sent with number of the next expected segment for fast retransmission.
- After receiving missing segment, acknowledgement must be sent with number of the next expected segment.
- Duplicate segment is discarded, acknowledgement must be immediately sent with number of the next expected segment for solving the problems with lost acknowledgement segment.

Retransmission

Segment is stored until it is acknowledged (Fig.3.4). Segment can be retransmitted because of expiration of retransmission [7] timer or after receiving of three duplicate acknowledgements for the first stored segment.

Retransmission time-out (RTO) is separated for each connection and it's value is dynamic based on round-trip time (RTT) of segments.

Fast retransmission allows to retransmit immediately after three duplicate acknowledgements are received. This method is called Reno.

Out-of-Order Segments
TCP always deliver in-order segment to upper layer but current TCP implementations store out-of-order segments temporarily [7].

Similarity to Automatic repeat request (ARQ) methods
Most of the current implementations of TCP are implemented without Selective acknowledgement (SACK) and only cumulative acknowledgements are sent. In this case, it looks like Go-Back-N ARQ method, but with temporary storing of out-of-order segments, it is mixed with behavior of Selective-Repeat ARQ. After SACK implementation, behavior of TCP is even closer to Selective-Repeat ARQ method [6].

3.7 Congestion Control in TCP

Congestion control uses congestion window and congestion policy to avoid congestion or detect and mitigate consequences, when it occurs.

3.7.1 Congestion Window

As was mentioned in previous section, flow control deals with overwhelmed receiver and controls the size of sender window. Network between sender and receiver can be overwhelmed too and there has to be a mechanism for network to tell the sender to slow down. Congestion Window [6] was implemented as a need to control congestion and together with window size, that advertises the receiver, change the window size of sender. Actual sender window size is the lowest value from congestion window and receiver advertised window.

3.7.2 Congestion Policy

Congestion policy contains slow start, congestion avoidance and congestion detection [6].

Slow start
Congestion window (cwnd) size starts with one maximum segment size (MSS), which is determined during connection establishment in TCP header options field. Congestion window increases one MSS each time acknowledgement segment is received.

The slow start algorithm starts slowly but increases rapidly until it reaches a threshold, which is called ssthresh (slow start threshold). Slow start increases the sender size slowly, when there are cumulative acknowledgements. The mechanism is shown in Fig. 3.7, where for simplicity, acknowledgement is send for each segment and receive window is longer than congestion window at any time.

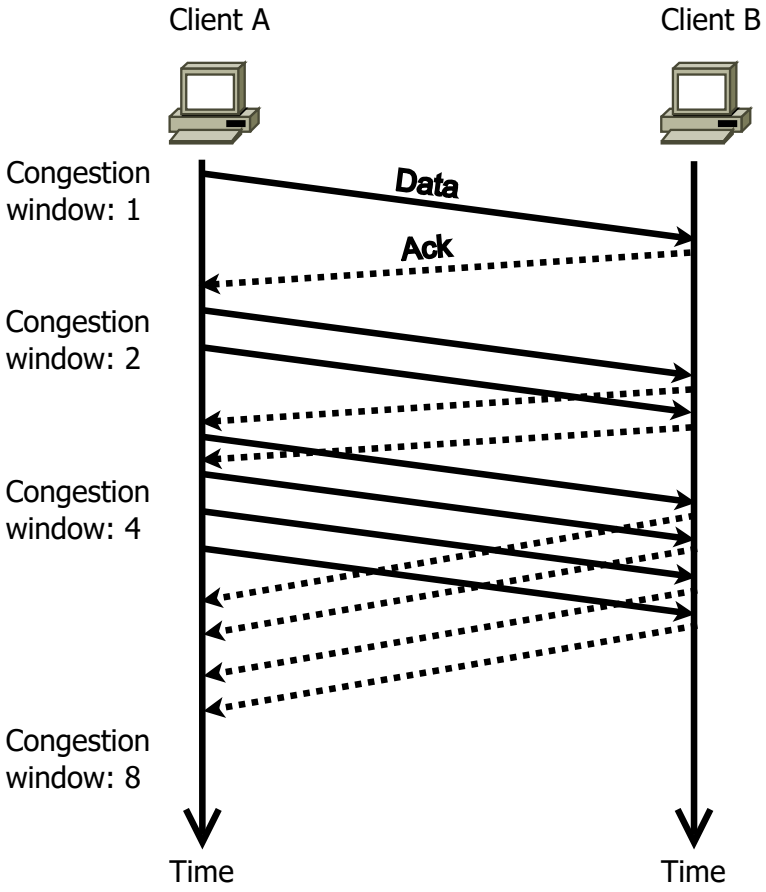


Fig. 3.7: Slow start mechanism with congestion window change

Congestion avoidance: Additive Increase

The slow start as increases rapidly the congestion window would eventually cause the congestion. Congestion Avoidance algorithm slow down the increase from (theoretical) exponential to additive growth. Congestion Avoidance algorithm starts when slow start algorithm ends, which is moment when congestion window reaches the slow start threshold. Then the additive increase begins until congestion is detected. Each time all segments in window are acknowledged, congestion window is increased by one. In this case, it is not important if cumulative acknowledgements are used but it depends on round-trip time (RTT) shown in Fig. 3.8.

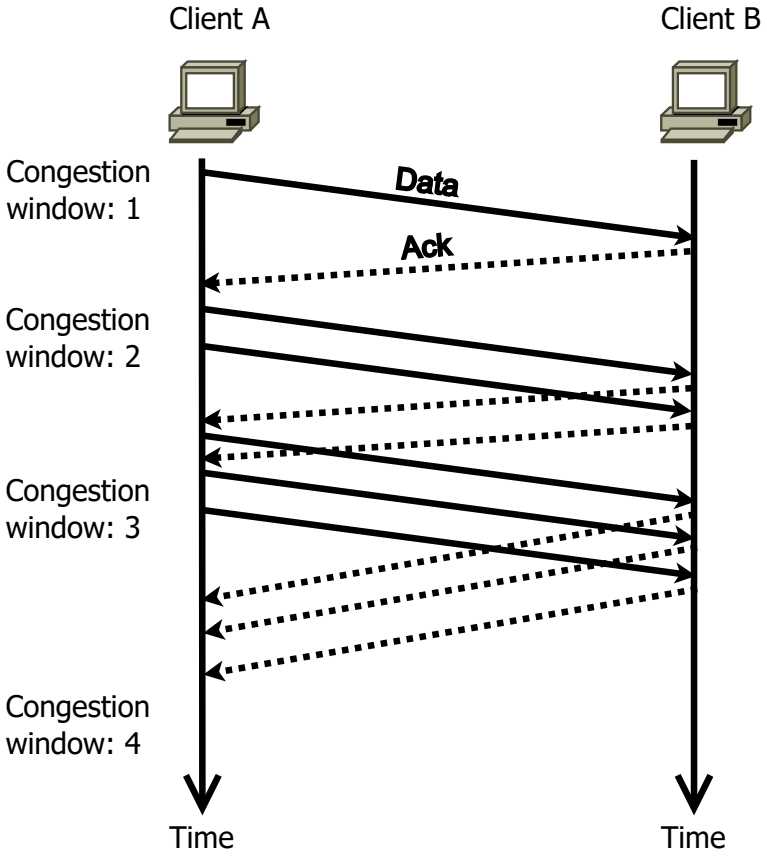


Fig. 3.8: Congestion avoidance with additive increase of congestion window

Congestion detection: Multiplicative Decrease

Congestion can be detected only based on need to retransmit a segment, which was dropped by some network part (most likely router) because of overload. Retransmission can happen when the RTO timer times out or when three duplicate acknowledgements are received, then the size of the threshold is in both cases decreased to half (multiplicative decrease). Depends of what was the reason for retransmission, there are two possible common reactions. Example of the reactions is shown in Fig. 3.9.

When the RTO timer times out and there is no informations about lost sent segments, there is a big possibility of congestion and the segments was probably dropped by some network part. Strong reaction begins with value of the threshold being set to half of the current window size. Congestion window is reduced to one segment and the slow start phase begins again (case A with blue line in Fig.3.9).

When three duplicate acknowledgements are received from the data receiver, a possibility of congestion is not too big as in previous case and some segments could have been dropped after receiving. Weaker reaction begins again with value of the threshold being set to half of the current window size. Congestion window is reduced

to the value of the threshold (case B with red line in Fig. 3.9) and the congestion avoidance phase begins again.

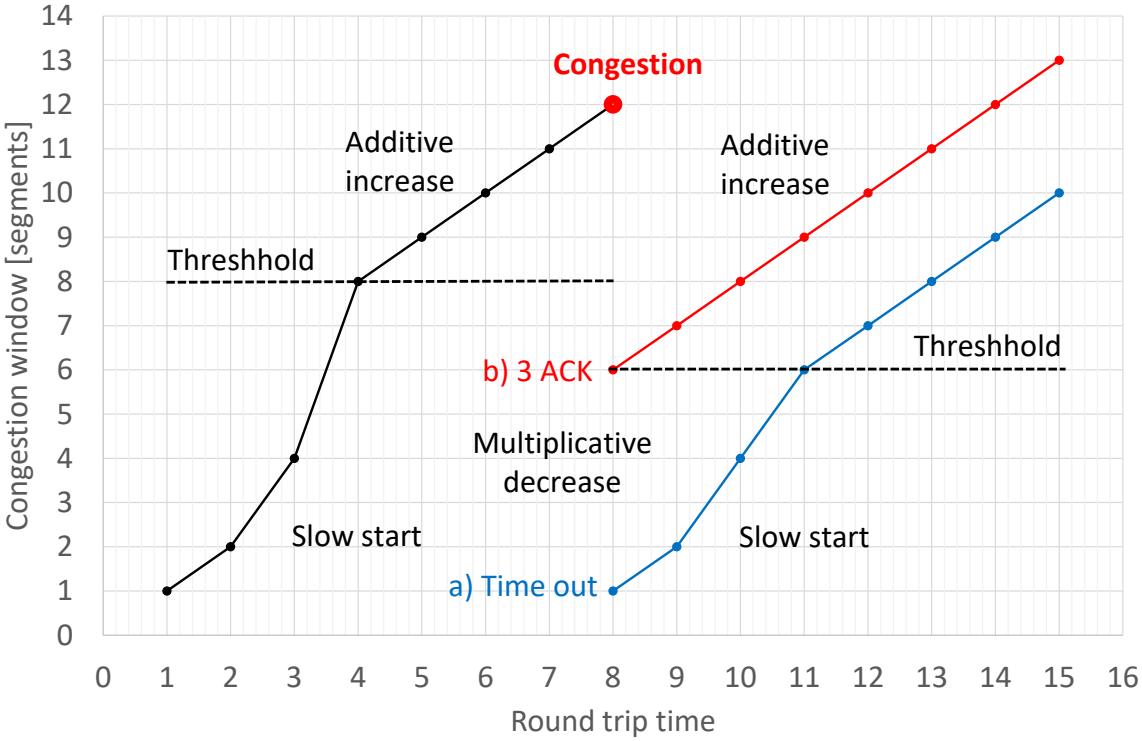


Fig. 3.9: Example of reactions after detecting congestion (Time out or 3 ACK)

3.8 Timers

TCP uses Retransmission, Persistence, Keep-alive and Time-wait timer [6].

Retransmission timer

Retransmission timer handles the retransmission time-out (RTO), which is the waiting time for an acknowledgement. For calculation of RTO, round trip time (RTT) has to be calculated first. There is a Measured RTT, Smoothed RTT and RTT Deviation. RTO is calculated based on the smoothed round-trip time and its deviation.

There can be defined some rules for Retransmission timer. The timer starts, when the segment in front of the sending queue is send. The first segment in front of the queue is resent and timer is restarted, if the timer expires. After cumulative acknowledgement, acknowledged segments are released from queue, because there is no need for storing them yet. Retransmission timer stops, when the queue is empty. If queue is not empty, timer is restarted after receiving acknowledgement.

Persistence timer
If the sender part receives segment with announced window size of zero, sender waits and do not send anything until receiving segment with announced window size of non zero size. If the second segment is lost, because of not acknowledging nor retransmitting of acknowledgement segments, both sides can wait for each other forever. Persistence timer for each connection, with special segment called a probe, solves this problem.

Keep-alive timer
It prevents a long idle connection without transfer of data. Without this timer, if client would crash during the data transfer, the connection would remain open forever. Keepalive timer is usually set to 2 hours. If the server has implemented this timer and there is no transfer during it's period, a probe segment is sent. With no response after 10 probes, connection is terminated.

Time-wait timer
This timer is used during connection termination.

4 AUTOMATIC REPEAT REQUEST (ARQ)

Automatic Repeat Request (ARQ) methods [8] use some form of sliding window technique, acknowledgements of messages with sequence numbers for correct order and time-outs. Flow and error control is basic purpose of ARQ.

Three basic ARQ methods include Stop-and-Wait ARQ, Go-Back-N ARQ and Selective-Repeat ARQ. ARQ methods are often implemented in the data link or transport layer of ISO/OSI model. Usually wireless data link protocols and reliable transport layer protocols such as TCP implements ARQ, but implementation on application layer is possible too (for example Trivial File Transfer Protocol).

4.1 Stop-and-Wait method

It is a very simple method [8] with receiver and sender sliding window of size 1. The sender sends only one segment and it is not allowed to send another until receives acknowledgement for sent segment. Receiver can detect corrupted segment because of checksum field, error notice can be sent or the segment is only discarded. The sender then retransmits the segment after time-out. Timer starts after sending a segment and ends after receiving an acknowledgement. Timer expiration is also possible after segment is lost and sender again retransmits duplicated stored unacknowledged segment.

Inefficiency is clear in Stop-and-Wait method (for example Fig. 4.1) and there is not many systems using this method for bigger data transfer. Other methods mentioned in next subsections provides more efficient options.

4.1.1 Numbering system

Because of only one segment can be sent until acknowledgement arrives, the sequence number field is only one bit (two possible numbers 0 and 1). If the segment is acknowledged, the sequence number is changed. If the packet is lost (Fig. 4.1) or the acknowledgement of segment (Fig. 4.2), segment is resent with the same sequence number. After acknowledgement is lost, the receiver knows that the same segment is resent again because the sequence number is not as was expected for the next one. Acknowledgement number is the sequence of the next expected segment by the receiver [6].

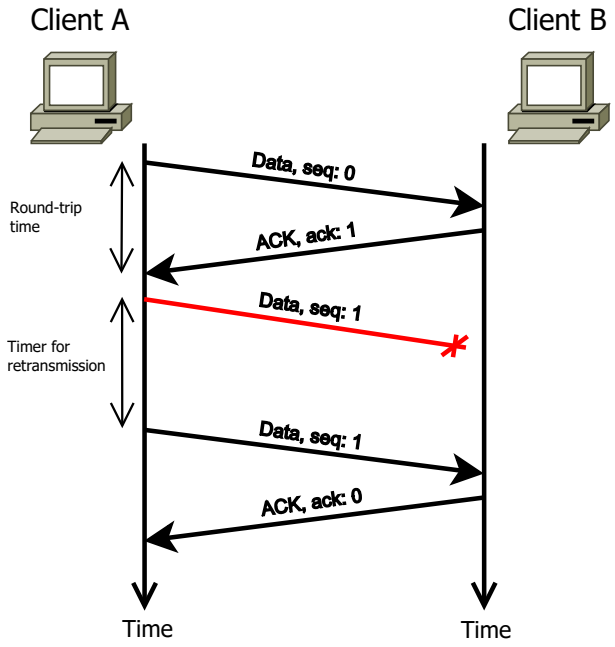


Fig. 4.1: Stop-and-Wait method: Solving problem with lost segment

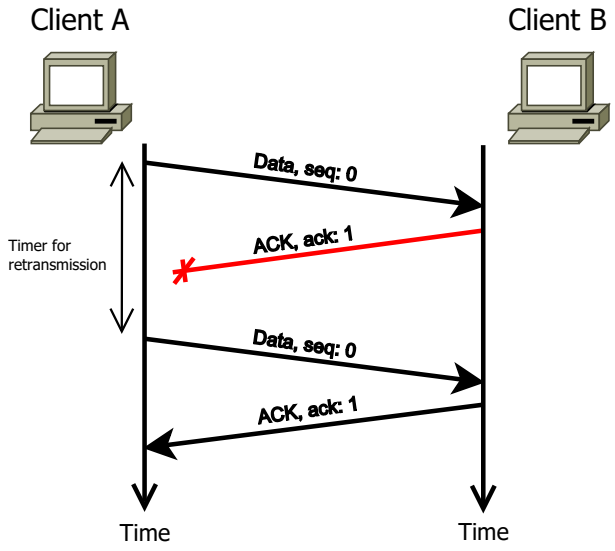


Fig. 4.2: Stop-and-Wait method: lost acknowledgement

4.2 Go-Back-N method

This method [6] allows to send more than one segment before receiving acknowledgements, but the receiver can still only buffer one segment same as in Stop-and-Wait method, example in Fig. 4.3. Send segments are also stored in sender part until acknowledgements are received.

4.2.1 Numbering system and windows

Acknowledgement number is cumulative [6] and the number is the sequence number of next expected segment. There is acknowledgement or negative acknowledgement (segment received with error) as a message for sender about the received segment. and Sequence number field size has to be higher because of more segments can be sent before acknowledgement arrives. Cumulative acknowledgement (acknowledgement sent after more segments in expected order arrives) has some advantages such as solving issue if ACK of the first segment would be lost but the second ACK would arrive to sender and the sender would assume the second ACK also acknowledges the first segment.

4.2.2 Windows and timers

Sender window size is usually fixed, but some protocols can have a variable size. Sender window has 3 basic parameters, which are send window size, sequence number of the first sent unacknowledged segment and sequence number of the first segment ready to send. The receive window size is always 1 and out of order segments are discarded. There is only one timer, after this timer expires, all unacknowledged segments are resent [7].

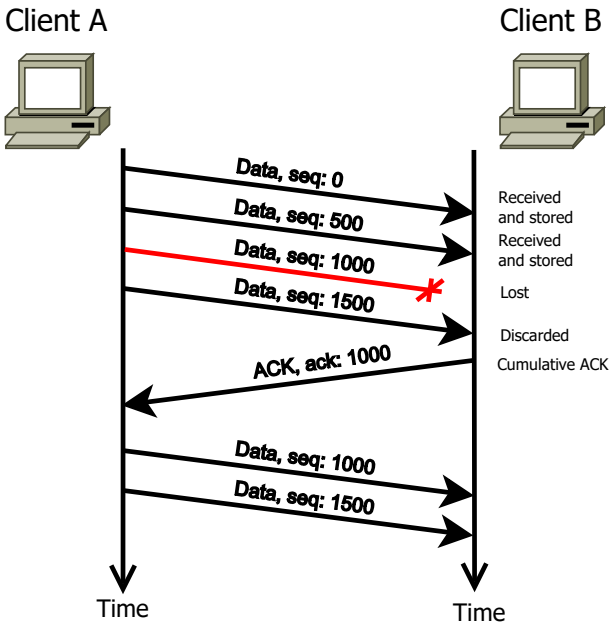


Fig. 4.3: Go-Back-N method: lost segment

4.3 Selective-Repeat method

Go-Back-N method is still quite inefficient, for example, if 5 segments are sent and the first is lost, other 4 are discarded on receiver side even if they arrived safely and sender has to send all 5 segments again. This inefficient behavior is solved by Selective-Repeat method [6], where only lost segments are resent again, , example in Fig. 4.4. Other side of comparing with previous methods is requirements for receiver, because it is not as simple operations as previous mentioned methods.

Acknowledgements in this method should confirm informations about only one segment and not confirm cumulatively all previous segments.

4.3.1 Windows and timers

Send and receive windows have the same size. This method allows to receive as many out of order segments as the size of the receive window, because the windows are the same size. Of course, segments has to be delivered to application layer in order and without errors.

Timer should be individual for each segment and the sender knows exactly what segment was lost. However, most implementations uses only one timer [7].

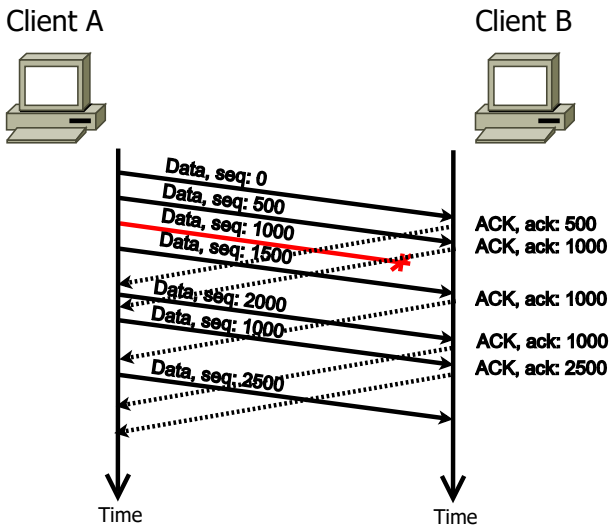


Fig. 4.4: Selective-Repeat method: lost segment

5 WAYS OF MESSAGE TRANSFER

There are some basic types of message transfer [7] and it usually depends on kind of transferred signal. Transfer of voice has different characteristics than transfer of data, for instance. This section will be mostly focused on packet and cell switching as widely known switching techniques in computer networks.

5.1 Types of message transfer and usage

These days, there are a lot of modern devices (many of them proprietary without possibility to know exactly how it works) and advanced protocols which allows wide range of possible behavioral in different situations. In this section, there will be comparison of basic model of switching techniques. More details will pro provided in next sections.

5.1.1 Circuit switching

Physical connection is created between hosts and even inside connecting nodes. Before transfer, connection needs to be set up which delays communication. This method is used usually in voice transfer, for example in telephone network.

5.1.2 Message switching

There is no physical connection between hosts. Message is stored and controlled in every node. The whole message has to be stored before control in interconnected device, which brings more requirements.

5.1.3 Packet switching

Packet switching [13] is similar to message switching but if the message is too long, it is fragmented to smaller parts (Fig. 5.1 - left side). Handling of packet on interconnected node is possible by more techniques (details in next sections), in Fig. 5.1 there is as an example check of whole packet on interconnected node. Separate packets are sent across network possible even in more different ways for one message, which brings possibility of receiving out of order packets. The most common way in data networks these days.

5.1.4 Cell switching

In cell switching [9], there is separating of message to smaller parts with exactly defined size. There is only check of header in transfer, which brings small amount of time spent in interconnected node. Control is done in host devices (Fig. 5.1 - right side). Usage is typically in Asynchronous Transfer Mode (ATM) networks. Advantage against circuit switching is saving resources of network. Exactly defined cell size brings advantages and disadvantages in different situations (more in next sections).

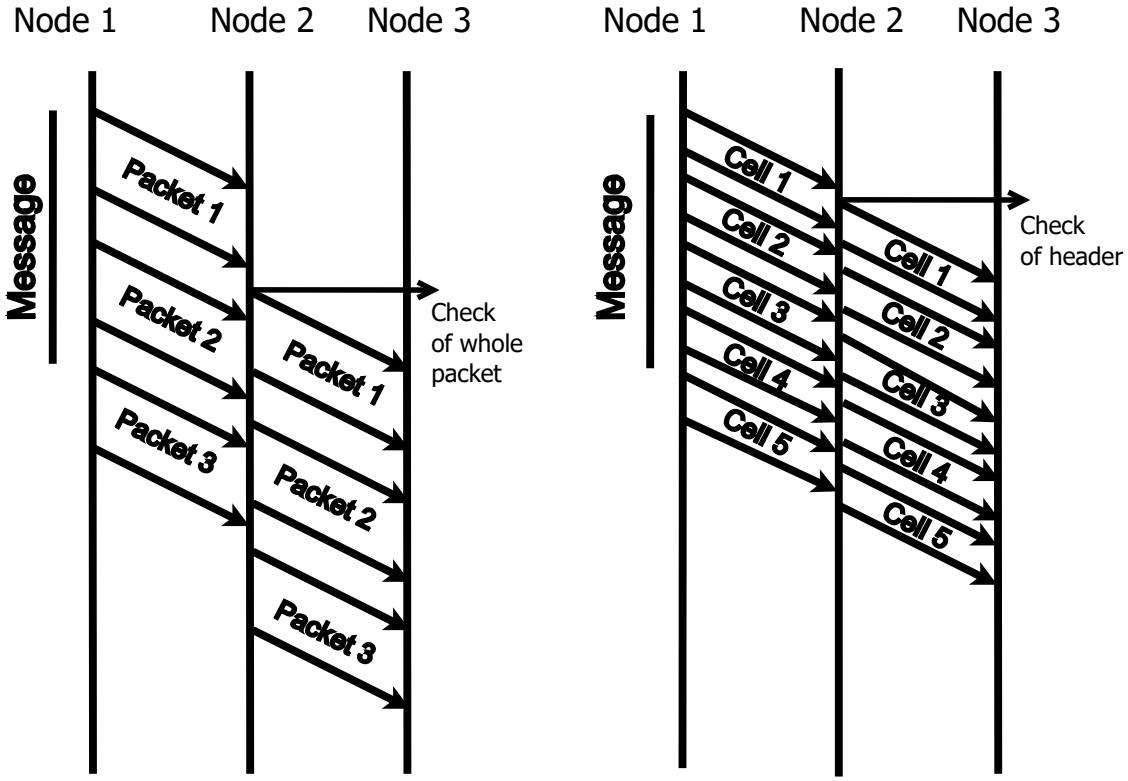


Fig. 5.1: Comparing packet switching and cell switching transfer of data

5.2 Processing of message on interconnected device

Network devices provide different options how to receive, process and forward packets. Some methods allow to lower latency, requirements for memory or processing power.

5.2.1 Store-and-Forward Switching

Switch waits for receiving whole frame (Fig. 5.2) which is copied into it's memory and count a Cyclic Redundancy Check (CRC). When counted CRC is the same as in frame, the frame is forwarded to destination. If counted CRC is not the same as CRC in frame, frame is discarded. Copying whole frame to memory and counting CRC causes delay and creates more requirements for switch memory. Because of modern hardware with optimized software, memory requirements is not problem and delay is not that higher than with other methods and that is the reason why this is the most used method [12] [13].

5.2.2 Cut-Through switching

Basic Cut-Through [13] method copies only the first 6 B of frame to switch memory, where is only MAC address (Fig. 5.2). This is the least needed information for finding destination for frame. This method reduces memory requirements, because not all frame is copied to memory. Latency is also reduced because switch can forward frame after reading destination MAC address and does not have to count CRC. Without checking CRC there can be situations when switch forwards corrupted frame.

Cut-Through method has some different options. One is called Fragment-Free switching [13]. It also reads only MAC address and then can make forwarding decision but it also waits for at least 64 B of frame because there can be runt frame, which is frame smaller than 64 B because according Ethernet specifications collision should be detected during first 64 B.

For example some data centre switches from Cisco called Nexus use Cut-Through method [12] with possibly more advanced behavioral. Not only that it checks destination MAC address but also field Type/Length (Fig. 5.2). When there is value 1536 and higher it means that it is Ethertype which defines used protocol encapsulated in date field. It can for example indicate IPv4 for which we use some Access Control List (ACL). With ACL there can be rule for investigating of network layer header and dropping the frame or checking QoS marking.

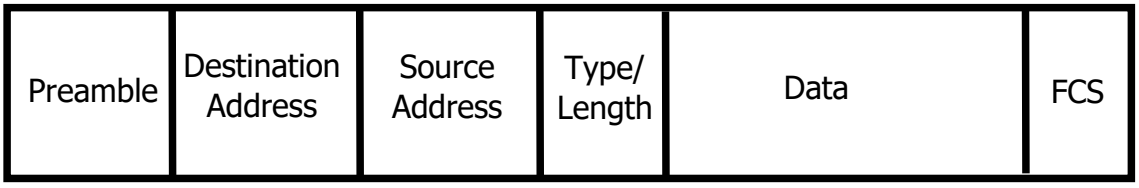


Fig. 5.2: Ethernet frame

5.3 Packet switching

Packet length can be variable, changing even during the transfer of one message. When packets are crossing network interconnected devices, there is process of receiving, storing and sending packets. After receiving packet, whole packet is buffered into device memory waiting for transmission (possible prioritizing with queues). There are other possibilities with no need of buffering whole packet into memory, mentioned in previous section. Different procedures on network devices means variable latency. Packet switching [10] allows travelling of packet in different ways in network even for one message. This results in optimizing of channel capacity usage, minimizing of latency and increasing robustness and flexibility in networks.

There are protocols/methods for connection-oriented or connectionless transfer of message [6]. Connectionless protocols are for example Ethernet, Internet Protocol (IP) and User Datagram Protocol (UDP). Connection-oriented protocol is for example Multiprotocol Label Switching (MPLS) and Transmission Control Protocol (TCP). Packet switching is connectionless with need of including addressing information in each packet, example of Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) packet in Fig. 5.3. It is needed because packets are routed individually possibly in different ways. This also brings possible out-of-order delivery of packets and higher requirements for receiver which has to store packets and put them into right order. For need of putting packets into right order there has to be a sequence number of packet (more information in header means bigger size). Interconnected network nodes provide connectionless services on network layer of International Standards Organization / Open Systen Interconnection (ISO/OSI) model, but there can be used for example connection-oriented protocol TCP which creates end-to-end connection-oriented transfer on transport layer of ISO/OSI model.

Connection-oriented transmission firstly has to set-up connection with each node in way involved [6]. After set-up phase, messages are delivered in order and headers can be smaller without need of information for individual routing decision. Connection-oriented switching is for example cell switching obtained in next section.

5.4 Cell switching

Cell switching [9] provides unreliable, connection-oriented switching with small fixed size of cells. Cell switching corresponds to physical and data link layer of ISO/OSI model with no flow or error control.

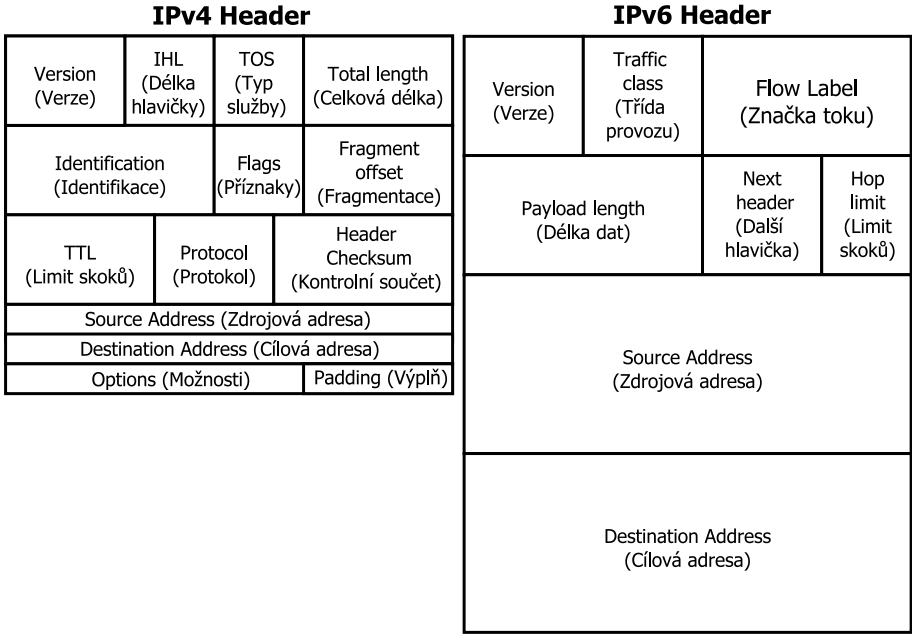


Fig. 5.3: IPv4 and IPv6 header

5.4.1 Asynchronous Transfer Mode (ATM)

Asynchronous Transfer Mode (ATM) is an example of cell switching [6]. ATM approximately works on network and lower layers of ISO/OSO model. ATM uses 53 B size of cells (48 B data and 5 B header) and is connection-oriented, which means virtual circuit is established before transmission. Virtual circuit can be permanent (dedicated link) or switched (set-up using signalling and disconnection after transmission). In packet switched networks, there can be problem with queuing of small packets of real-time data (voice, video) together with big packets usually used for other data. With fixed size of cell, jitter as a variance of delay is reduced which is important for voice or video traffic.

ATM cell is 53 B big as mentioned before with 5 B of header and 48 B of payload. Two different cell types [9] are used, User-Network Interface (UNI) and Network-to-Network Interface (NNI) (Fig. 5.4). Usually UNI cell format is used in ATM links. The Generic Flow Control (GFC) field is a 4-bit field originally created to negotiate multiplexing and flow control with various ATM connections but it was never standardized and field is always set to 0000. Virtual Path Identifier (VPI) is used as an identification of virtual path. Virtual Channel Identifier (VCI) is used as an identification of virtual channel. Virtual circuits and paths can be created statically or dynamically [11]. Virtual circuit routing is usually provided by Private Network-to-Network Interface (PNNI) protocol uses shortest-path-first algorithm. Payload Type (PT) can specify network management cell, explicit forward conges-

tion indication or ATM user-to-user option for indicating cell boundaries. Cell Loss Priority (CPL) is used for priority of cell and Header Error Control (HEC) is 8-bit CRC.

ATM uses different ATM Adaptation Layers (AAL) types [11] from AAL1 up to AAL5 for different kind of services. In set-up phase, each network device is informed about traffic type, which can be CBR (Constant Bit Rate), VBR (Variable Bit Rate), ABR (Available Bit Rate) and UBR (Unspecified Bit Rate).

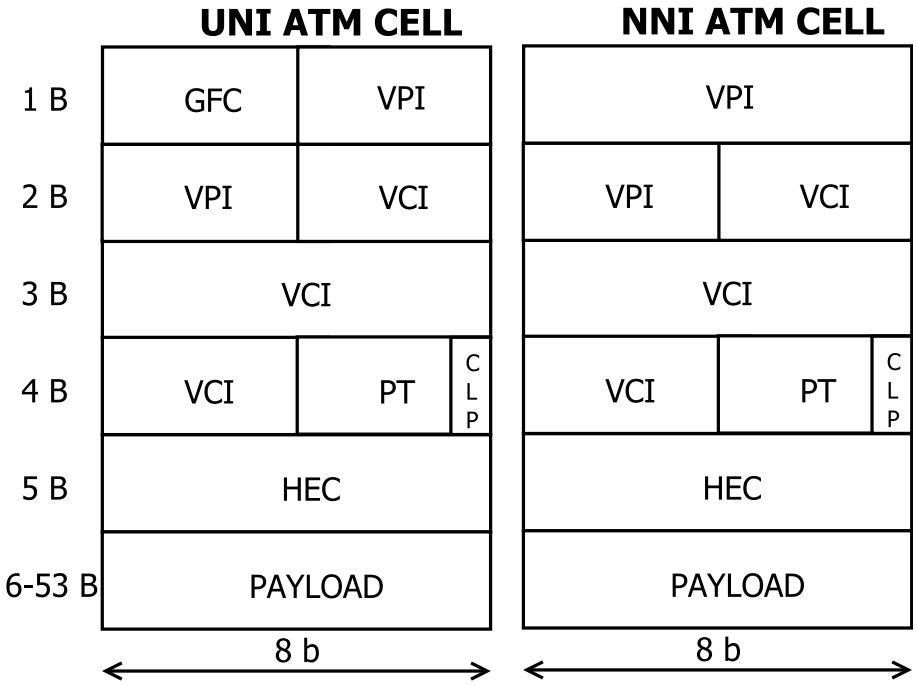


Fig. 5.4: UNI ATM and NNI ATM cell

6 AUTOMATIC REPEAT REQUEST (ARQ) SCENARIO

6.1 Creating scenario in ns-3, used methods and issues

Firstly, important was to find out, what is the default settings. For that, it was necessary to create some loss. I used *ErrorModel* class for creation of error rate of corrupted bytes. It is also possible to generate packet error rate for receiving messages on interface, which was used in other part.

Another task was to simulate Stop-and-Wait method for comparison. There is no possibility of changing it as a protocol, because it is a method used in protocols. According theoretical basics, the way could be by shrinking the receive and sender window. I found the possibility in class *TcpSocket*, where receive *(RcvBufSize)* and send *(SndBufSize)* buffer can be changed. Send buffer change was not useful, because the sender window was, after creating a connection, the minimum from congestion or receiver window (as is usually in real networks). So, when the receive window was decreased, the send window was the same. The size of data with header is 590 bytes (default ns-3 behavior), but the receive window has to be set higher or the flow will not work (receiver process delivered packet and report free window size, which has to be big enough for 1 other packet). It can not be much higher, because there could be sent more than one segment. The size of 1050 bytes was found as a working option.

After need of more nodes, there was needed to implement routing, I used static routing but in NS-3 there is possibility of dynamic routing too. I had to implement restriction of size for sent data so we can compare it for same sizes and not only for generated flow for some time without defined data size.

Biggest problem was with generating errors. I wanted to measure impact of packet error rate on needed time and data size with transmission of TCP and UDP communication. UDP was not possible to use, because I did not find any possible protocol from higher layer to deal with retransmission (for example TFTP did not work). With TCP there was problem with slow processing time of nodes and with higher packet error rate than 10% sender sends same packet more times and with every retransmission the waiting time is double size of previous one (1 s default) and another data are generated so graphs are not accurate. Fortunately even data from packet error rate lower than 10% shows differences.

6.2 Topology and scenario

Topology represents 2 point-to-point links for communication of firstly Node 0 with Node 1 and Node 0 with Node 2 in later parts (Fig. 6.1).

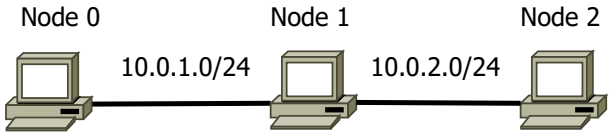


Fig. 6.1: Topology of ARQ scenario

Scenario is divided into three parts:

1. Implementing base of the scenario
2. Simulation tasks
3. Answers to control questions and assignment check

Scenario was created for understanding ARQ basic methods Stop-and-Wait, Go-Back-N and Selective-Repeat. By changing network and host parameters, there is shown the difference between methods. There is change of send and receive window size and behavior after loss.

First part is for creating basic simulation environment, where are simulated two hosts (Node 0 and Node 1) on point-to-point channel (Fig. 6.1). Second part is about analyzing of default behavior, adding another node, changing ARQ methods and comparing in different situations. There are included control questions and individual tasks. Last part contains answers to control questions and explanation or verification of individual assignments.

6.2.1 Implementing base of the scenario

Create new file and add modules

Create new document with name "arq_scenario.cc" in folder Scratch. There has to be only one document in Scratch folder for proper functionality.

Insert a modules needed for our scenario, allow logging and define naming space before using main function:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
```

```
#include "ns3/csma-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/gnuplot.h"
#include "ns3/tcp-socket-base.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("arq_scenario");
```

Create nodes

There are only two nodes (Node 0 and Node 1) for base scenario, we will use class *NodeContainer* and create mentioned two nodes. This and all other parts should be created inside main function. Code inside main function consists of:

```
int
main (int argc, char *argv[])
{
NodeContainer nodes12;
nodes12.Create (2);

}
```

Create channel between nodes

We have nodes connected using point-to-point link and for simplicity we can use class *PointToPointHelper*. Set data rate to 5 Mbps, delay to 3 ms and MTU to 1500 B. Data rate is low because of easier simulation tracking, delay is 0 s as an default value which is not showing real situation and it does not show time sequence in simulation (3 ms could be delay in local network including few switches). MTU 1500 B is standard value for Ethernet, we can set this by code:

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("3ms"));
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(1500));
```

Install TCP/IP stack

We can install TCP/IP protocol stack on our nodes:

```
InternetStackHelper stack;
stack.Install (nodes12);
```

Set IP addresses

Create network devices from current nodes. Set IP addresses to Node 0 and Node 1 from 10.0.1.0/24:

```
NetDeviceContainer devices;
devices = pointToPoint.Install (nodes12);
Ipv4AddressHelper address;
address.SetBase ("10.0.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = address.Assign (devices);
Ptr<Node> Node0 = nodes12.Get (0);
Ptr<Node> Node1 = nodes12.Get (1);
```

Generate TCP flow

We can use on/off application using TCP for generating a flow towards Node 1. Set TCP port 80 (http protocol) and packet size to 512 B, as an example of particular solution. For later error rate generation and lower simulation time, packet size is 512 B which is smaller than usual value. Application will start at 1.0 s and end at 10.0 s. Code for setting is:

```
OnOffHelper onoffTCP ("ns3::TcpSocketFactory",
Address (InetSocketAddress (interfaces.GetAddress (1), 80)));
onoffTCP.SetAttribute ("PacketSize", UintegerValue (512));
onoffTCP.SetAttribute ("OnTime",
StringValue ("ns3::ConstantRandomVariable[Constant=1]"));
onoffTCP.SetAttribute ("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0]"));
ApplicationContainer appTCP = onoffTCP.Install (Node0);
appTCP.Start (Seconds (1.0));
appTCP.Stop (Seconds (10.0));
```

Create receiver of data

There has to be a receiver of generated data, in our case, it is the second node (Node 1). End of our application receiver part should be later than at the sender side:

```
PacketSinkHelper sinkTCP ("ns3::TcpSocketFactory", Address
(InetSocketAddress (Ipv4Address::GetAny (), 80)));
appTCP = sinkTCP.Install (Node1);
appTCP.Start (Seconds (1.0));
appTCP.Stop (Seconds (11.0));
```

Logging data transfer for Wireshark

Enable capturing the data on point-to-point line to pcap document:

```
pointToPoint.EnablePcapAll ("arq_scenario");
```

Set start and end of simulation

```
Simulator::Run ();
Simulator::Destroy ();
return 0;
```

Setting run configuration

Firstly press Run -> Run for starting the project. It will not work, but now we can set up project in Run configuration menu (Run -> Run Configuration -> Search Project -> arq_scenario -> Run) same as in Fig. 6.2. After this step, configuration file is set and only Run -> Run or green button will work. After running configuration, there should be no error in console output, we can check if our communication works fine by opening generated pcap file in Wireshark (Fig. 6.3).

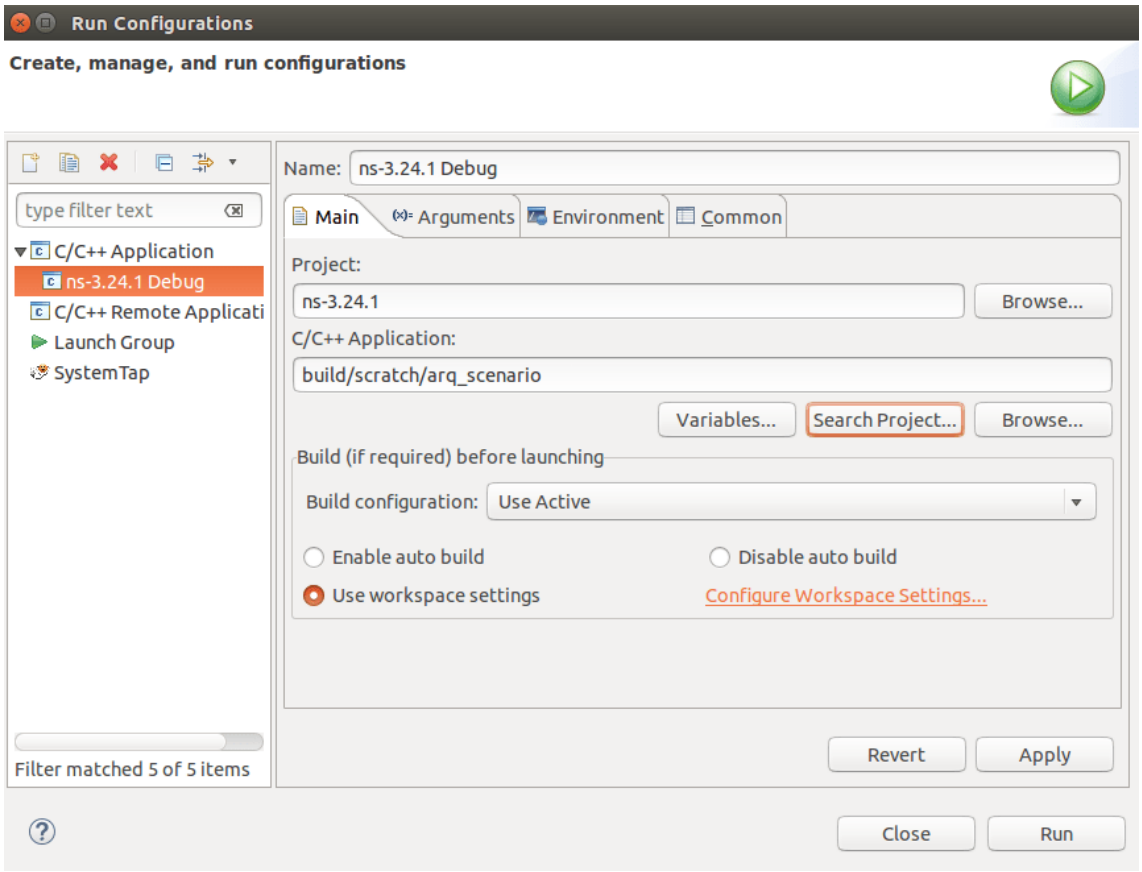
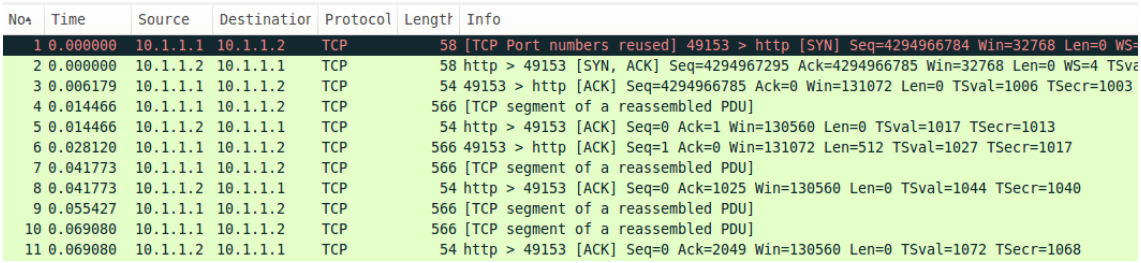


Fig. 6.2: Set-up of configuration file



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.1.1.1	10.1.1.2	TCP	58	[TCP Port numbers reused] 49153 > http [SYN] Seq=4294966784 Win=32768 Len=0 WS=
2	0.000000	10.1.1.2	10.1.1.1	TCP	58	http > 49153 [SYN, ACK] Seq=4294967295 Ack=4294966785 Win=32768 Len=0 WS=4 TSva
3	0.006179	10.1.1.1	10.1.1.2	TCP	54	49153 > http [ACK] Seq=4294966785 Ack=0 Win=131072 Len=0 TSval=1006 TSecr=1003
4	0.014466	10.1.1.1	10.1.1.2	TCP	566	[TCP segment of a reassembled PDU]
5	0.014466	10.1.1.2	10.1.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=1 Win=130560 Len=0 TSval=1017 TSecr=1013
6	0.028120	10.1.1.1	10.1.1.2	TCP	566	49153 > http [ACK] Seq=1 Ack=0 Win=131072 Len=512 TSval=1027 TSecr=1017
7	0.041773	10.1.1.1	10.1.1.2	TCP	566	[TCP segment of a reassembled PDU]
8	0.041773	10.1.1.2	10.1.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=1025 Win=130560 Len=0 TSval=1044 TSecr=1040
9	0.055427	10.1.1.1	10.1.1.2	TCP	566	[TCP segment of a reassembled PDU]
10	0.069080	10.1.1.1	10.1.1.2	TCP	566	[TCP segment of a reassembled PDU]
11	0.069080	10.1.1.2	10.1.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=2049 Win=130560 Len=0 TSval=1072 TSecr=1068

Fig. 6.3: Wireshark output of TCP data transfer with default settings in NS-3

6.2.2 Simulation tasks

In this part, we will analyze ARQ mechanisms in our version of ns-3 (3.24.1) and try to implement different mechanism for analysis and comparing.

Control questions and individual tasks are listed in italics with answers in next section.

Analyze current ARQ mechanism

After running configuration (done in previous part), there are generated two documents, one for each side of communication. Rename first one ("arq_scenario-0-1.pcap") for later comparison (after next simulation, file would we replaced) and open it in Wireshark.

Analyze TCP flow for determination of used ARQ mechanism (Ack, Seq, Win and other informations). Is it clear, which mechanism is used? In next steps, all signs will be explained and paired with specific mechanism. Example of expected output is in Fig. 6.3.

More segments are sent without receiving acknowledgement of the first one. Window (Win) is larger than one segment. There are cumulative acknowledgements. From this point of view, it is clear that it is not Stop-and-Wait method in any way. Cumulative acknowledgements are sign of Go-Back-N method. All segments are received in proper order and no loss occurred, but behavior after these occasions is crucial for our analysis. Errors will be generated in next step.

a) If the Selective-Repeat method would have cumulative acknowledgements and no loss would occur, what would be the difference with Go-Back-N method?

Generation of loss

We will generate errors on medium with using of class *RateErrorModel* for generating byte (default setting) and lately packet loss. Firstly we will generate byte error rate 0.0005 (generation of big amount of errors for purpose of finding default ARQ method, exact calculations and simulations in next parts) and implement on Node 1 for receiving bytes. We will increase data rate of TCP application from default 500 kb/s to 4 Mb/s so Node 0 can send segments quickly and we can see handling (on Node 1) of packets after losing previous one. Code below insert above code for running simulation (except of data rate, which insert after set-up of packet size):

```
Ptr<RateErrorModel> errorModel = CreateObject<RateErrorModel> ();
errorModel->SetAttribute ("ErrorRate", DoubleValue (0.0005));
devices.Get (1)->SetAttribute ("ReceiveErrorModel",
PointerValue (errorModel));
onoffTCP.SetAttribute ("DataRate", StringValue ("4Mbps"));
```

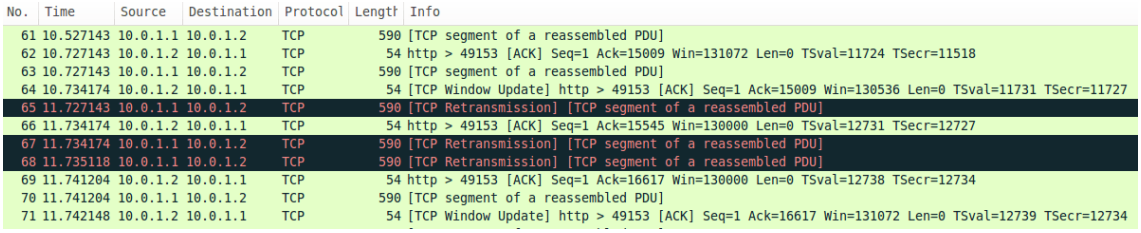
Run simulation and check pcap file from Node 0 in Wireshark ("arq_scenario-0-1.pcap"). There will be a lost, which can be determined by black row with red characters (example of lost segment is in Fig. 6.4). Analyze behavior in next segments after loss (Ack, Seq and Win). Are segments received and stored, if some segment is missing with lower sequence number? You can see receiving window shrinking after receiving next segments until lost segment is received. Also, after this segment is resent, receiver sent Ack with higher number than only for next segment after lost one (Fig. 6.5). This behavior matches Selective-Repeat method.

It is clear, this TCP implementation uses some parts from Go-Back-N mechanism (cumulative ack) and Selective-Repeat mechanism (storing out of order segments and resending only missing one).

b) Is there used any congestion control mechanism? If there is, where can you see it?
c) What is the congestion mechanism used in this ns-3 version? Compare it to SACK method.

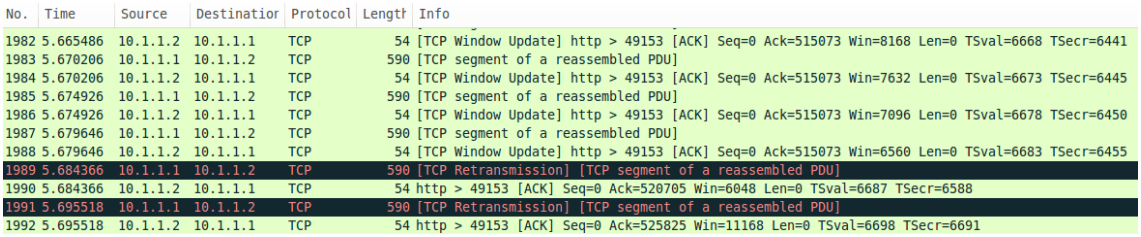
Creating Stop-and-Wait method

We have to realize that ARQ is a mechanism, not a protocol. We can not only switch some setting from Selective-Repeat to Stop-and-Wait. What could be the way for implementing Stop-and-Wait in NS-3? When the receive and send window would be decreased for only one segment size, there would be no option for sender to send another segment until it is acknowledged. This is possible in NS-3, we re-



No.	Time	Source	Destination	Protocol	Length	Info
61	10.527143	10.0.1.1	10.0.1.2	TCP	590	[TCP segment of a reassembled PDU]
62	10.727143	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=1 Ack=15009 Win=131072 Len=0 TSval=11724 TSecr=11518
63	10.727143	10.0.1.1	10.0.1.2	TCP	590	[TCP segment of a reassembled PDU]
64	10.734174	10.0.1.2	10.0.1.1	TCP	54	[TCP Window Update] http > 49153 [ACK] Seq=1 Ack=15009 Win=130536 Len=0 TSval=11731 TSecr=11727
65	11.727143	10.0.1.1	10.0.1.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
66	11.734174	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=1 Ack=15545 Win=130000 Len=0 TSval=12731 TSecr=12727
67	11.734174	10.0.1.1	10.0.1.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
68	11.735118	10.0.1.1	10.0.1.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
69	11.741204	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=1 Ack=16617 Win=130000 Len=0 TSval=12738 TSecr=12734
70	11.741204	10.0.1.1	10.0.1.2	TCP	590	[TCP segment of a reassembled PDU]
71	11.742148	10.0.1.2	10.0.1.1	TCP	54	[TCP Window Update] http > 49153 [ACK] Seq=1 Ack=16617 Win=131072 Len=0 TSval=12739 TSecr=12734

Fig. 6.4: Transfer of data in Wireshark in default settings with loss



No.	Time	Source	Destination	Protocol	Length	Info
1982	5.665486	10.1.1.2	10.1.1.1	TCP	54	[TCP Window Update] http > 49153 [ACK] Seq=0 Ack=515073 Win=8168 Len=0 TSval=6668 TSecr=6441
1983	5.670206	10.1.1.1	10.1.1.2	TCP	590	[TCP segment of a reassembled PDU]
1984	5.670206	10.1.1.2	10.1.1.1	TCP	54	[TCP Window Update] http > 49153 [ACK] Seq=0 Ack=515073 Win=7632 Len=0 TSval=6673 TSecr=6445
1985	5.674926	10.1.1.1	10.1.1.2	TCP	590	[TCP segment of a reassembled PDU]
1986	5.674926	10.1.1.2	10.1.1.1	TCP	54	[TCP Window Update] http > 49153 [ACK] Seq=0 Ack=515073 Win=7096 Len=0 TSval=6678 TSecr=6450
1987	5.679646	10.1.1.1	10.1.1.2	TCP	590	[TCP segment of a reassembled PDU]
1988	5.679646	10.1.1.2	10.1.1.1	TCP	54	[TCP Window Update] http > 49153 [ACK] Seq=0 Ack=515073 Win=6560 Len=0 TSval=6683 TSecr=6455
1989	5.684366	10.1.1.1	10.1.1.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
1990	5.684366	10.1.1.2	10.1.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=520705 Win=6048 Len=0 TSval=6687 TSecr=6588
1991	5.695518	10.1.1.1	10.1.1.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
1992	5.695518	10.1.1.2	10.1.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=525825 Win=11168 Len=0 TSval=6698 TSecr=6691

Fig. 6.5: Retransmission of data in Wireshark in default settings

duce receive window and by that, send window will be the same size by implemented mechanisms in NS-3. Delete generation of loss from previous part for now and delete set-up of data rate (better control and clearer output with slower communication). Firstly try to set 1000 B which should be fine for 1 our packet of size 512 B (with header) and ACK of size 54 B. Code with receive window size defined in bytes will be implemented in part after defining channel:

```
Config::SetDefault ("ns3::TcpSocket::RcvBufSize",
UintegerValue (1000));
```

Delete:

```
Ptr<RateErrorModel> errorModel = CreateObject<RateErrorModel> ();
errorModel->SetAttribute ("ErrorRate", DoubleValue (0.0005));
devices.Get (1)->SetAttribute ("ReceiveErrorModel",
PointerValue (errorModel));
onoffTCP.SetAttribute ("DataRate", StringValue ("4Mbps"));
```

Run configuration and open generated pcap file from Node 0 ("arq_scenario-0-1.pcap") in Wireshark. We can see (Fig. 6.6) that Node 0 received ACK of first packet but is not able to send another one and there is only end of TCP connection at the end of our simulation. The reason can be visible again in Fig. 6.6 in 5th packet where Node 1 sent ACK of previous packet with reported left window size of 488 B (1000 B window - 512 B of first message) and then as we already know sender

window size is set to the same size as receiver window size. With sender window size of 488 B, sender is not able to send 512 B packet. Solution is to set receiver window size at least for 2 packets of our size (2 x 512 B) but not that big that sender can send 2 packets in a row. To be sure about handling headers or Ack we can set size of receiver window a bit higher than 2 packets (for example 1050 B):

```
Config::SetDefault ("ns3::TcpSocket::RcvBufSize",
UintegerValue (1050));
```

Run configuration and check Wireshark output again. Now we can see (Fig. 6.7) that receiver is now reporting more than 512 B window size, sender sends every time only 1 packet and waiting for ACK so it is behavioral of Stop-and-Wait method.

d) Is there any difference with theoretical model of Stop-and-Wait method? If there is, what the change means?

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.1.1	10.0.1.2	TCP	58	[TCP Port numbers reused] 49153 > http [SYN] Seq=4294966784 Win=1000 Len=0 WS=1 TSval=1000 TSecr=0
2	0.006185	10.0.1.2	10.0.1.1	TCP	58	http > 49153 [SYN, ACK] Seq=4294967295 Ack=4294966785 Win=1000 Len=0 WS=1 TSval=1003 TSecr=1000
3	0.006185	10.0.1.1	10.0.1.2	TCP	54	49153 > http [ACK] Seq=4294966785 Ack=0 Win=1000 Len=0 TSval=1006 TSecr=1003
4	0.008192	10.0.1.1	10.0.1.2	TCP	566	[TCP segment of a reassembled PDU]
5	0.015183	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=1 Win=488 Len=0 TSval=1012 TSecr=1008
6	10.003086	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [FIN, ACK] Seq=0 Ack=1 Win=1000 Len=0 TSval=11000 TSecr=1008
7	10.003086	10.0.1.1	10.0.1.2	TCP	54	49153 > http [ACK] Seq=1 Ack=1 Win=1000 Len=0 TSval=11003 TSecr=11000

Fig. 6.6: Transfer of data in Wireshark with Stop-and-Wait method simulated by receive window 1000 B

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.1.1	10.0.1.2	TCP	58	[TCP Port numbers reused] 49153 > http [SYN] Seq=4294966784 Win=1050 Le
2	0.006185	10.0.1.2	10.0.1.1	TCP	58	http > 49153 [SYN, ACK] Seq=4294967295 Ack=4294966785 Win=1050 Len=0 WS
3	0.006185	10.0.1.1	10.0.1.2	TCP	54	49153 > http [ACK] Seq=4294966785 Ack=0 Win=1050 Len=0 TSval=1006 TSecr
4	0.008192	10.0.1.1	10.0.1.2	TCP	566	[TCP segment of a reassembled PDU]
5	0.015183	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=1 Win=538 Len=0 TSval=1012 TSecr=1008
6	0.016383	10.0.1.1	10.0.1.2	TCP	566	49153 > http [ACK] Seq=1 Ack=0 Win=1050 Len=512 TSval=1016 TSecr=1012
7	0.223375	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=513 Win=1050 Len=0 TSval=1220 TSecr=1016
8	0.223375	10.0.1.1	10.0.1.2	TCP	590	[TCP segment of a reassembled PDU]
9	0.430406	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=1049 Win=1050 Len=0 TSval=1427 TSecr=1223
10	0.430406	10.0.1.1	10.0.1.2	TCP	590	[TCP segment of a reassembled PDU]
11	0.637436	10.0.1.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=1585 Win=1050 Len=0 TSval=1634 TSecr=1430

Fig. 6.7: Transfer of data in Wireshark with Stop-and-Wait method simulated by receive window 1050 B

Add another node and enable communication

In this part we will add another node to our topology as in Fig. 6.1. We have to create another node same as previous ones and create same point-to-point connection between new Node 2 and Node 1 as with Node 0 and Node 1. For enabling of communication we have to set IP addresses on new interfaces and set-up routing so

Node 0 can reach Node 2 and opposite. We will use class *Ipv4StaticRoutingHelper* for easier creation of static routing. Code for creating pointer for Node 2 insert below pointer for Node 1 and created connections between nodes with IP addresses insert instead code already inserted for Node 0 and Node 1. Static routing with pointers for setting IP address insert before onoff application:

```
nodes123.Create (3);
Ptr<Node> Node2 = nodes123.Get (2);

NetDeviceContainer devices;
devices = pointToPoint.Install (Node0, Node1);
Ipv4AddressHelper address;
address.SetBase ("10.0.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = address.Assign (devices);

NetDeviceContainer devices12;
devices12 = pointToPoint.Install (Node1, Node2);
Ipv4AddressHelper address12;
address12.SetBase ("10.0.2.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces12 = address12.Assign (devices12);

Ptr<Ipv4> ipv4N0 = Node0->GetObject<Ipv4> ();
Ptr<Ipv4> ipv4N2 = Node2->GetObject<Ipv4> ();

Ipv4StaticRoutingHelper ipv4RoutingHelper;
Ptr<Ipv4StaticRouting>staticRoutingN0 =
ipv4RoutingHelper.GetStaticRouting (ipv4N0);
Ptr<Ipv4StaticRouting> staticRoutingN2 =
ipv4RoutingHelper.GetStaticRouting (ipv4N2);
staticRoutingN0->AddHostRouteTo (Ipv4Address("10.0.2.2"),
Ipv4Address ("10.0.1.2"), 1);
staticRoutingN2->AddHostRouteTo (Ipv4Address("10.0.1.1"),
Ipv4Address ("10.0.2.1"), 1);
```

Comparing communication with 2 and 3 nodes without errors between TCP with different ARQ methods and UDP

We will compare transmission of data with 2 and 3 nodes with using firstly default settings in NS-3 (Selective-Repeat method with cumulated Ack), Stop-and-Wait method simulated previously and UDP transmission without any ARQ method.

We added another node with static routing in previous task. Firstly we will send same amount of data with both ARQ methods and we will check differences with needed time and data for transmission. Then we will send the same amount of data with UDP protocol and check the same values. We will generate 50000 B in our communication in every case (quite small amount of data but simulation will be faster what we will appreciate especially with errors later) which is almost 100 of 512 B messages (for precise statistics would be better more and this is compromise for saving processing time but still with visible differences). Also we will change simulation time of client and server up to 300 s (needed later with errors) but it is not problem because there will be transmitted only 50000 B of data (plus headers, Ack and retransmission later). Insert maximum of our transmitted data (50000 B) under packet size set up:

```
onoffTCP.SetAttribute ("MaxBytes", UintegerValue (50000));
appTCP.Stop (Seconds (300.0)); appTCP.Stop (Seconds (300.0));
```

Firstly we can start with Stop-and-Wait because we have it implemented now. Run configuration and check size of transmitted messages (data + headers + Ack) for our data (50000 B) in pcap document for Node 0 ("arq_scenario-0-1.pcap"). You can find it as size of created pcap file. Also open this document in Wireshark and check needed time for transmission. Note this 2 values, we will than compare it with another methods (check your values with Fig. 6.9 and Fig. 6.10). Now we will change our server to Node 2 so we can communicate between 3 nodes and then compare values with 2 nodes in each communication:

```
appTCP = sinkTCP.Install (Node2);
OnOffHelper onoffTCP ("ns3::TcpSocketFactory",
Address (InetSocketAddress (interfaces12.GetAddress (1), 80)));
```

Run configuration and check folder with created pcap files. You can see that now for every of 4 used interfaces there was created pcap file with same amount of data. Again note size of Node 0 document and needed time for later comparison. We will also compare impact of sent data with 3 nodes in every method. Change size of sent message to 100000 B:

```
onoffTCP.SetAttribute ("MaxBytes", UintegerValue (100000));
```

Run simulation and again note needed time and data for transmission. You can compare your values with Fig. 6.11 and Fig. 6.12 which contains already other results and will be compared after all simulations.

Now we would delete code for creation of Stop-and-Wait method and do the same simulation again as a simulation of Selective-Repeat.

Individual task: run same simulations as with Stop-and-Wait simulations but with default settings of ARQ mechanism in NS-3 (delete code for Stop-and-Wait method). Results are listed for comparison in Fig. 6.9, Fig. 6.10 and Fig. 6.11, Fig. 6.12, which will be explained and compared after last simulation which is communication with UDP.

We will simulate the same statistics with UDP as a last simulation of this part. Code looks the same except of changing type of setting ***OnOffHelpder*** and ***PacketSinkHelper*** to UDP. For better naming (not TCP but UDP) there is listed code with better naming of classes and variables below (delete TCP communication and insert code below with UDP instead of TCP). We will **delete** (should be already done from previous task) code for simulation of Stop-and-Wait:

```
Config::SetDefault ("ns3::TcpSocket::RcvBufSize",
UintegerValue (1050));

OnOffHelper onoffUDP ("ns3::UdpSocketFactory",
Address (InetSocketAddress (interfaces12.GetAddress (1), 80)));
onoffUDP.SetAttribute ("PacketSize", UintegerValue (512));
onoffUDP.SetAttribute ("MaxBytes", UintegerValue (100000));

onoffUDP.SetAttribute ("OnTime",
StringValue ("ns3::ConstantRandomVariable[Constant=1]"));
onoffUDP.SetAttribute ("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0]"));
ApplicationContainer appUDP = onoffUDP.Install (Node0);
appUDP.Start (Seconds (1.0));
appUDP.Stop (Seconds (300.0));

PacketSinkHelper sinkUDP ("ns3::UdpSocketFactory", Address
(InetSocketAddress (Ipv4Address::GetAny (), 80)));
appUDP = sinkUDP.Install (Node2);
appUDP.Start (Seconds (1.0));
appUDP.Stop (Seconds (300.0));
```

Run configuration and again check Wireshark output (could be same as in Fig. 6.8). You can see that there is no initiation of connection or ending because UDP is not connection-oriented protocol as TCP. There is no acknowledgement of received messages and header is smaller in comparison to TCP. These all factors will play role

in later comparison.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http
2	0.008192	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http
3	0.016384	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http
4	0.024576	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http
5	0.032768	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http
6	0.040960	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http
7	0.049152	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http
8	0.057344	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http
9	0.065536	10.0.1.1	10.0.2.2	UDP	542	Source port: 49153 Destination port: http

Fig. 6.8: UDP communication between Node 0 and Node 2

Individual task: run same simulations as in previous two simulations but with UDP and not TCP protocol (code included in previous part). Results are listed for comparison in Fig. 6.9, Fig. 6.10 and Fig. 6.11, Fig. 6.12, which will be explained and compared after this simulation.

Now we can compare previous simulations. We did simulation without errors with TCP protocol and default ARQ mechanism (Selective-Repeat) in our version of NS-3, with Stop-and-Wait mechanism and with UDP protocol. We changed number of nodes (2 and 3) and value of send messages (50000 B and 100000 B).

Firstly we will check needed time and messages size (data + headers + Ack) for transmission with 3 previously mentioned simulations and impact of more nodes. As we can see in Fig. 6.9 the size on needed data is the same with more nodes, middle node is just passing messages between client and server. With UDP there is less additional data then with TCP, which is expected because UDP is connectionless protocol and does not even acknowledge packets. The most data is needed for Stop-and-Wait implementation, because there has to be acknowledged every packet not like with our default implementation of Selective-Repeat with cumulative acknowledgement of more packets. When we check Fig. 6.10 and compare time needed for transmission of our message, we can see that with additional node in the middle of communication UDP time is still the same. There will be of course some additional time but because middle node is not checking whole segment or any other activity with possible delay, the additional time is so small that is not even seen in our case. Because of this reason and less additional data then TCP transmission, UDP has again the best result. With Selective-Repeat method we can see value not that higher than UDP, there is additional connection establishment, connection

ending and cumulative acknowledgement but sender can send data almost all the time. We can see the difference between 2 and 3 nodes, where there has to be check of whole segment not like with UDP. With Stop-and-Wait method we can see more than 20 times higher needed time for transmission. It is mainly because sender has to wait for acknowledgement after every sent packet. The values would change with different processing time on nodes, link speed and other parameters.

We can also check message again for needed time and messages size (data + headers + Ack) for transmission with impact of different size of message. As we can see in 6.12, with twice more data we have twice higher needed time for transmission with every method. In Fig. 6.11 we can see that size of needed transmitted data is again twice higher for twice more data. These results were expected and later simulations will continue only with same size of data. We have to realize that UDP was better in our simulations than TCP because it was ideal environment with no errors and only one communication. In real situations, there are some corrupted/lost packets, more different transfers of data from different applications where TCP is able to dynamically change parameters of transfer for available bandwidth, amount of errors on medium and latency changes by congestion, error and flow control mechanisms.

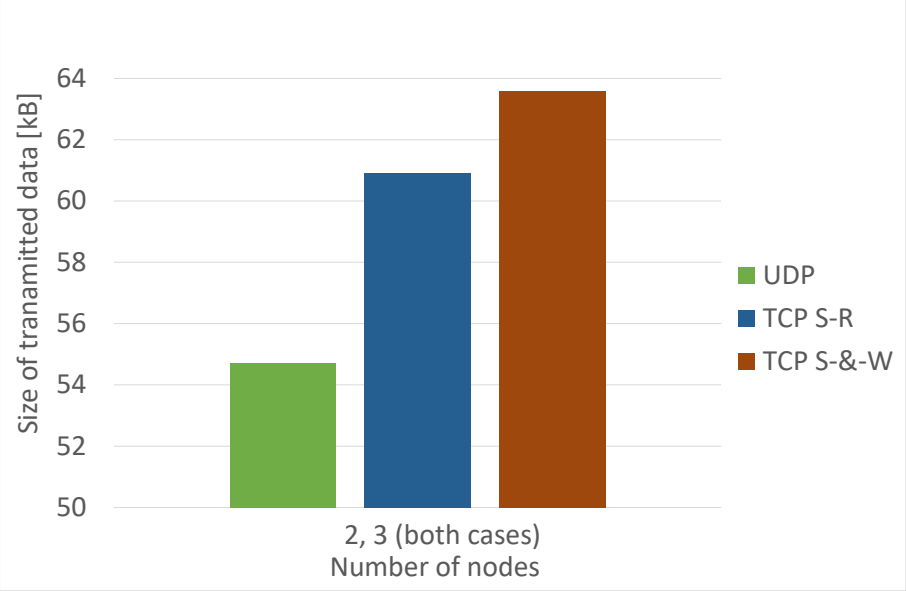


Fig. 6.9: Size of transmitted data with more nodes for 50000 B message

Comparison of ARQ methods in error communication

Now we can not compare UDP any more, it is possible to retransmit data with higher protocols (for example TFTP) which uses UDP in transport layer of ISO/OSI model

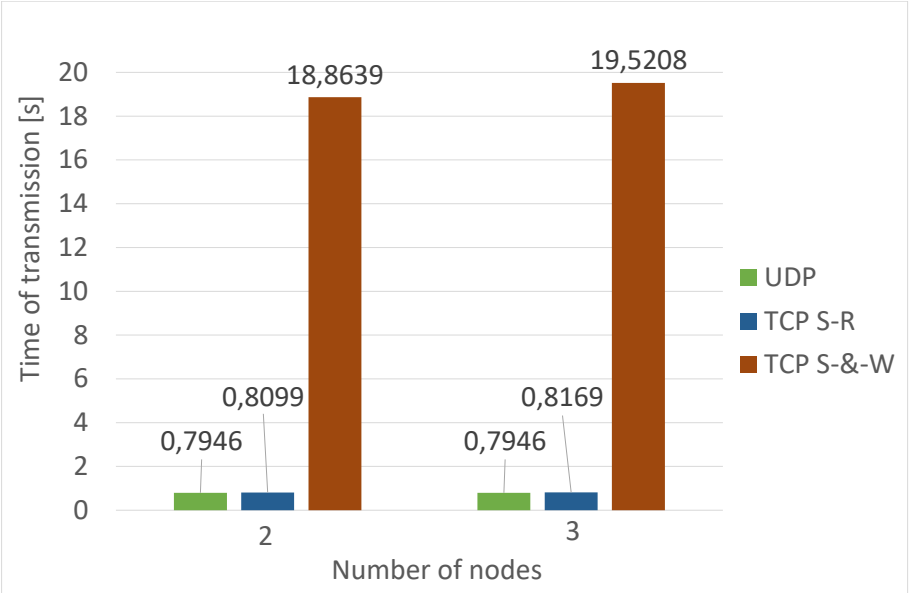


Fig. 6.10: Time needed for transmission with more nodes for 50000 kB message

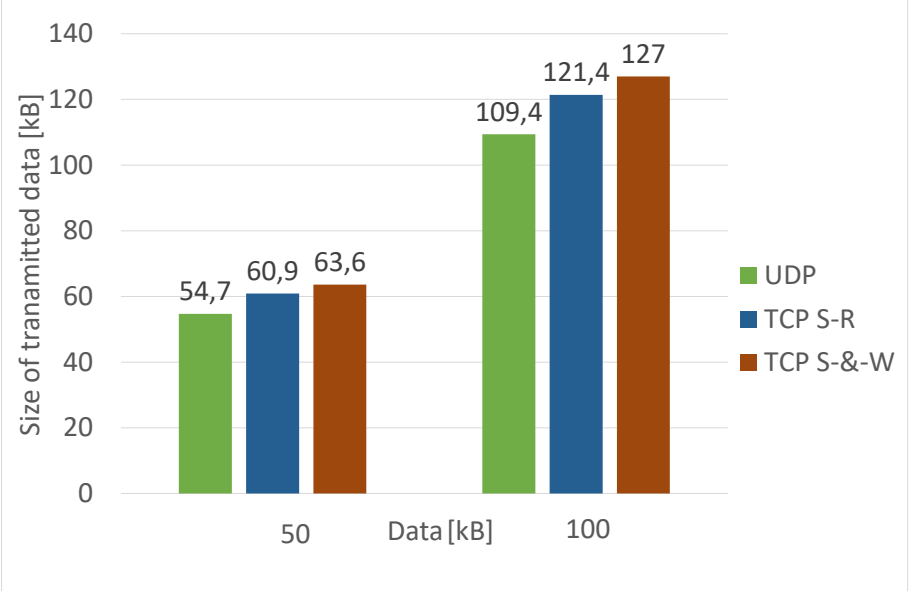


Fig. 6.11: Size of all transmitted messages for different size of data

but this is not implemented in NS-3. We will compare our two ARQ mechanisms in error communication. Now we will use packet error generation (not as previous byte error generation). We already checked behavioral in error communication with Selective-Repeat method in previous part.

We could make simulation of impact of increasing errors in communication to needed time and data for transmission with both ARQ methods. Unfortunately, NS-3 does not have implemented a lot of mechanisms for handling errors and with

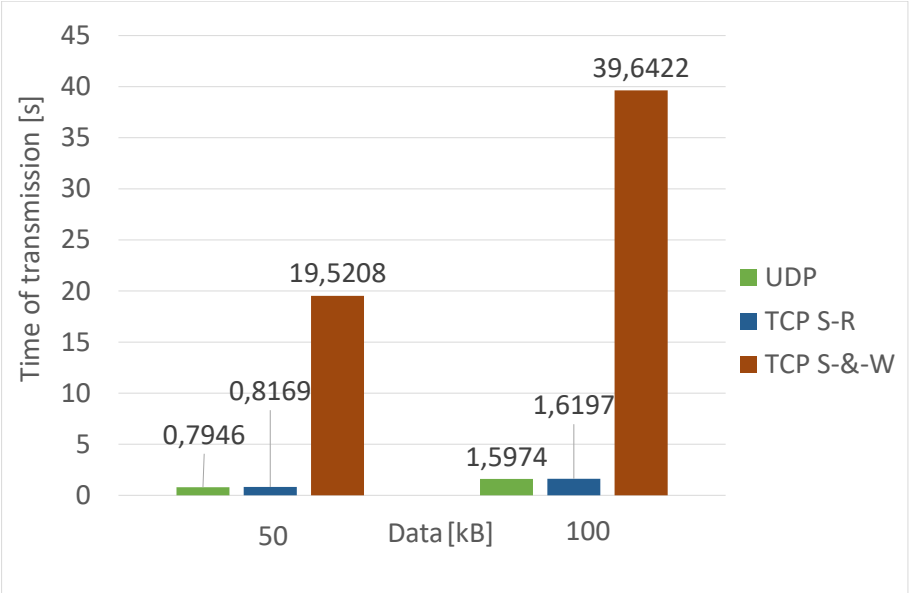
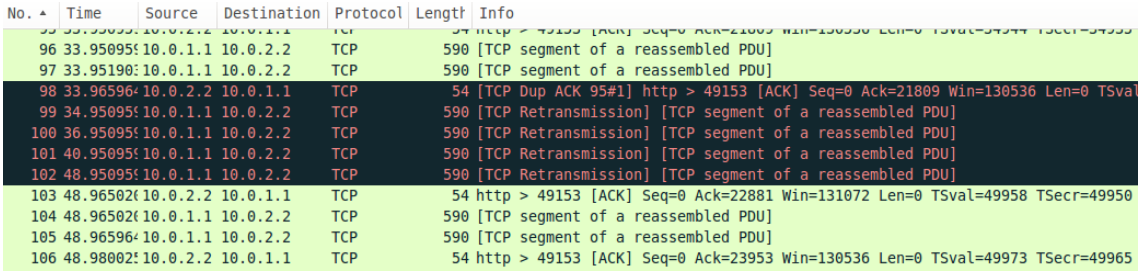


Fig. 6.12: Time needed for transmission of different size of data

higher error rate, the results are not according expectations. With higher error rate and default settings of ARQ method, there are parts where sender is resending the same packet a lot of times because receiver do not send Ack or sender is not reacting on Ack (can be caused by slow processing of nodes), which consumes time and creates huge outputs. Example of this behavioral can be seen in Fig. 6.13. With Stop-and-Wait simulation and higher error rate, there is not only the same problem as with default setting but another problem because of lower receiver window. If there is some error in beginning messages (Fig 6.14) where receiver's buffer is filled by creating TCP handshake and after resending of packet, receiver informs sender about his window size lower than for message, so sender can not send another packet and is just waiting for the end of communication to the end of TCP connection (same when we did not set receive window to the proper size without errors). If we try to set higher buffer, it is not possible to set it for behavioral of Stop-and-Wait and for recovering from this, when it starts to work with lowest size of window, then it sends later 2 packets without Ack.

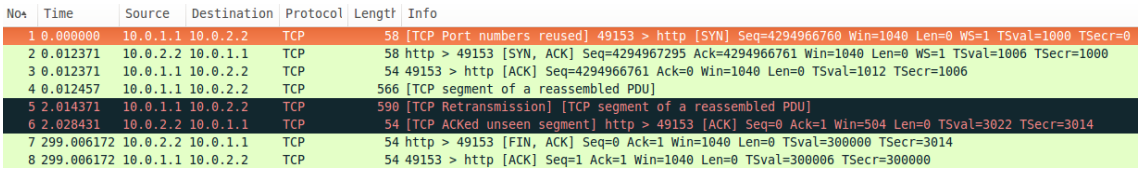
This happens with packet error rate higher than 10% which is extreme case and we can still simulate lower packet error rate and compare needed time and cumulated messages size (data + headers + ack) for same amount of data. Change again our application to TCP, set 100000 B of data and use Node 0 and Node 2 for communication through Node 1:

```
OnOffHelper onoffTCP ("ns3::TcpSocketFactory",
Address (InetSocketAddress (interfaces12.GetAddress (1), 80)));
onoffTCP.SetAttribute ("PacketSize", UintegerValue (512));
```



No.	Time	Source	Destination	Protocol	Length	Info
96	33.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
97	33.95190	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
98	33.96596	10.0.2.2	10.0.1.1	TCP	54	[TCP Dup ACK 95#1] http > 49153 [ACK] Seq=0 Ack=21809 Win=130536 Len=0 TSval
99	34.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
100	36.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
101	40.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
102	48.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
103	48.96502	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=22881 Win=131072 Len=0 TSval=49958 TSecr=49950
104	48.96502	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
105	48.96596	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
106	48.98002	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=23953 Win=130536 Len=0 TSval=49973 TSecr=49965

Fig. 6.13: Resending same packet more times because of missing acknowledgement



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.1.1	10.0.2.2	TCP	58	[TCP Port numbers reused] 49153 > http [SYN] Seq=4294966760 Win=1040 Len=0 WS=1 TSval=1000 TSecr=0
2	0.012371	10.0.2.2	10.0.1.1	TCP	58	http > 49153 [SYN, ACK] Seq=4294967295 Ack=4294966761 Win=1040 Len=0 WS=1 TSval=1006 TSecr=1000
3	0.012371	10.0.1.1	10.0.2.2	TCP	54	49153 > http [ACK] Seq=4294966761 Ack=0 Win=1040 Len=0 TSval=1012 TSecr=1006
4	0.012457	10.0.1.1	10.0.2.2	TCP	566	[TCP segment of a reassembled PDU]
5	2.014371	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
6	2.028431	10.0.2.2	10.0.1.1	TCP	54	[TCP ACKed unseen segment] http > 49153 [ACK] Seq=0 Ack=1 Win=504 Len=0 TSval=3022 TSecr=3014
7	299.006172	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [FIN, ACK] Seq=0 Ack=1 Win=1040 Len=0 TSval=300000 TSecr=3014
8	299.006172	10.0.1.1	10.0.2.2	TCP	54	49153 > http [ACK] Seq=1 Ack=1 Win=1040 Len=0 TSval=300006 TSecr=300000

Fig. 6.14: Problem with error communication with smaller receiver window

```
onoffTCP.SetAttribute ("MaxBytes", UintegerValue (100000));

onoffTCP.SetAttribute ("OnTime",
StringValue ("ns3::ConstantRandomVariable[Constant=1]"));
onoffTCP.SetAttribute ("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0]"));
ApplicationContainer appTCP = onoffTCP.Install (Node0);
appTCP.Start (Seconds (1.0));
appTCP.Stop (Seconds (300.0));

PacketSinkHelper sinkTCP ("ns3::TcpSocketFactory", Address
(InetSocketAddress (Ipv4Address::GetAny (), 80)));
appTCP = sinkTCP.Install (Node2);
appTCP.Start (Seconds (1.0));
appTCP.Stop (Seconds (300.0));
```

Set packet error rate to 10 % (1 = 100% of errors, 0 = 0% of errors) as our maximum packet error rate (explained in previous part), we will implement is on receiver interface for receiving packets and on sender interface for receiving packets (later we will discover impact). Insert code before start of simulation:

```
Ptr<RateErrorModel> errorModel = CreateObject<RateErrorModel> ();
errorModel->SetAttribute ("ErrorUnit",
StringValue ("ERROR_UNIT_PACKET"));
```

```
errorModel->SetAttribute ("ErrorRate", DoubleValue (0.1));
devices.Get (0)->SetAttribute ("ReceiveErrorModel",
PointerValue (errorModel));
devices12.Get (1)->SetAttribute ("ReceiveErrorModel",
PointerValue (errorModel));
```

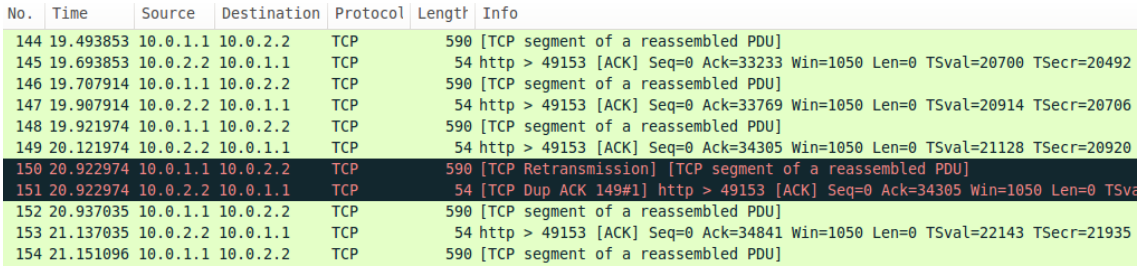
Run simulation and open Wireshark outputs (shown later) for Node 0 and Node 2 and analyze retransmission reasons and behavioral connected with retransmission. We implemented errors that packet in each side can be lost (corrupted). We generated errors on receiver interface for receiving bytes already in analysing of current ARQ mechanism part. In this situation, packet from sender is not delivered (without errors) to receiver. As we can see in Wireshark output from our simulation on receiver (Fig. 6.15) and sender interfaces (Fig. 6.16), the packet is not delivered, sender waits 1 s and resends packet again. We also added errors in other direction on sender's interface for receiving bytes (packets). This way is used for acknowledgement from receiver to sender. As we can see in Wireshark output, packet is delivered to receiver (Fig. 6.17) but sender did not receive Ack (Fig. 6.18). Sender has no chance to know if the packet arrived to receiver so waits again 1 s and resend packet. Receiver after receiving the same packet again sends the same acknowledgement.

No.	Time	Source	Destination	Protocol	Length	Info
28	2.382813	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
29	2.582813	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=6433 Win=1050 Len=0 TSval=3588 TSecr=3381
30	2.596874	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
31	2.796874	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=6969 Win=1050 Len=0 TSval=3803 TSecr=3595
32	2.810934	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
33	3.010934	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=7505 Win=1050 Len=0 TSval=4017 TSecr=3809
34	4.025995	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
35	4.225995	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=8041 Win=1050 Len=0 TSval=5232 TSecr=5024
36	4.240056	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
37	4.440056	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=8577 Win=1050 Len=0 TSval=5446 TSecr=5238
38	4.454117	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]

Fig. 6.15: Example of lost packet - receiver side is missing one sent packet

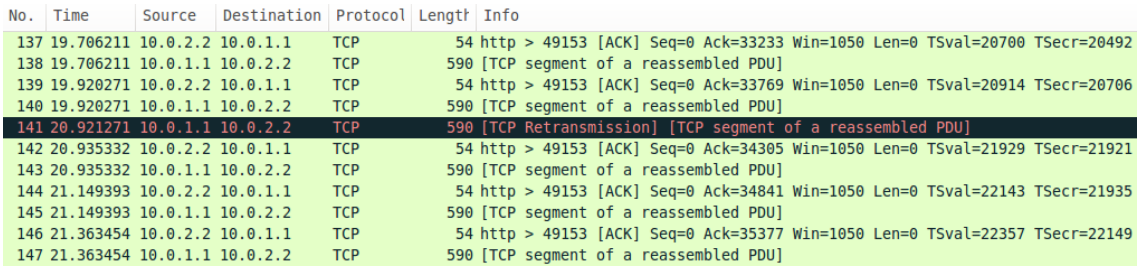
No.	Time	Source	Destination	Protocol	Length	Info
31	2.809231	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=6969 Win=1050 Len=0 TSval=3803 TSecr=3595
32	2.809231	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
33	3.023292	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=7505 Win=1050 Len=0 TSval=4017 TSecr=3809
34	3.023292	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
35	4.024292	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
36	4.238353	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=8041 Win=1050 Len=0 TSval=5232 TSecr=5024
37	4.238353	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
38	4.452414	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=8577 Win=1050 Len=0 TSval=5446 TSecr=5238
39	4.452414	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
40	4.666475	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=9113 Win=1050 Len=0 TSval=5660 TSecr=5452

Fig. 6.16: Example of lost packet - sender did not receive Ack to one packet and has to resent it again



No.	Time	Source	Destination	Protocol	Length	Info
144	19.493853	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
145	19.693853	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=33233 Win=1050 Len=0 TSval=20700 TSecr=20492
146	19.707914	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
147	19.907914	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=33769 Win=1050 Len=0 TSval=20914 TSecr=20706
148	19.921974	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
149	20.121974	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=34305 Win=1050 Len=0 TSval=21128 TSecr=20920
150	20.922974	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
151	20.922974	10.0.2.2	10.0.1.1	TCP	54	[TCP Dup ACK 149#1] http > 49153 [ACK] Seq=0 Ack=34305 Win=1050 Len=0 TSva
152	20.937035	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
153	21.137035	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=34841 Win=1050 Len=0 TSval=22143 TSecr=21935
154	21.151096	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]

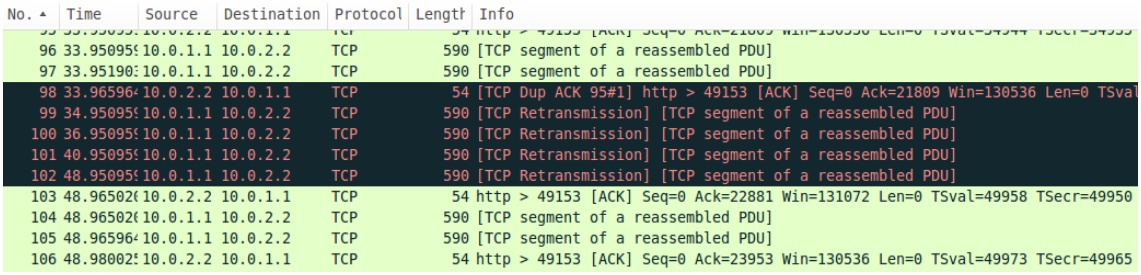
Fig. 6.17: Example of lost packet - receiver received all packets and sent Ack packet to sender



No.	Time	Source	Destination	Protocol	Length	Info
137	19.706211	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=33233 Win=1050 Len=0 TSval=20700 TSecr=20492
138	19.706211	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
139	19.920271	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=33769 Win=1050 Len=0 TSval=20914 TSecr=20706
140	19.920271	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
141	20.921271	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
142	20.935332	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=34305 Win=1050 Len=0 TSval=21929 TSecr=21921
143	20.935332	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
144	21.149393	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=34841 Win=1050 Len=0 TSval=22143 TSecr=21935
145	21.149393	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
146	21.363454	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=35377 Win=1050 Len=0 TSval=22357 TSecr=22149
147	21.363454	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]

Fig. 6.18: Example of lost packet - sender did not receive Ack for one packet

How is it with sender's waiting time for resending same packets without Ack (look again to Wireshark and look for more resent packets in a row)? We can see that NS-3 in this version changes waiting time for Ack before resending the packet to twice higher value starting with 1 s (and lowering in time without errors), we can see it in Fig. 6.19.



No. ▲	Time	Source	Destination	Protocol	Length	Info
96	33.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
97	33.95190	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
98	33.96596	10.0.2.2	10.0.1.1	TCP	54	[TCP Dup ACK 95#1] http > 49153 [ACK] Seq=0 Ack=21809 Win=130536 Len=0 TSval
99	34.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
100	36.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
101	40.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
102	48.95095	10.0.1.1	10.0.2.2	TCP	590	[TCP Retransmission] [TCP segment of a reassembled PDU]
103	48.96502	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=22881 Win=131072 Len=0 TSval=49958 TSecr=49950
104	48.96502	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
105	48.96596	10.0.1.1	10.0.2.2	TCP	590	[TCP segment of a reassembled PDU]
106	48.98002	10.0.2.2	10.0.1.1	TCP	54	http > 49153 [ACK] Seq=0 Ack=23953 Win=130536 Len=0 TSval=49973 TSecr=49965

Fig. 6.19: Example of changing waiting time for resending packet

Note values of data size needed for transmission of 100000 B (size of pcap file) and time needed for transmission (time of last packet in pcap file on Node 0). We can compare it with Fig. 6.20 and Fig. 6.21 (it can be a bit different because generation of errors can differ a little). Now we will change ARQ mechanism to

Stop-and-Wait and check these values as well for later comparison:

```
Config::SetDefault ("ns3::TcpSocket::RcvBufSize",
UintegerValue (1050));
```

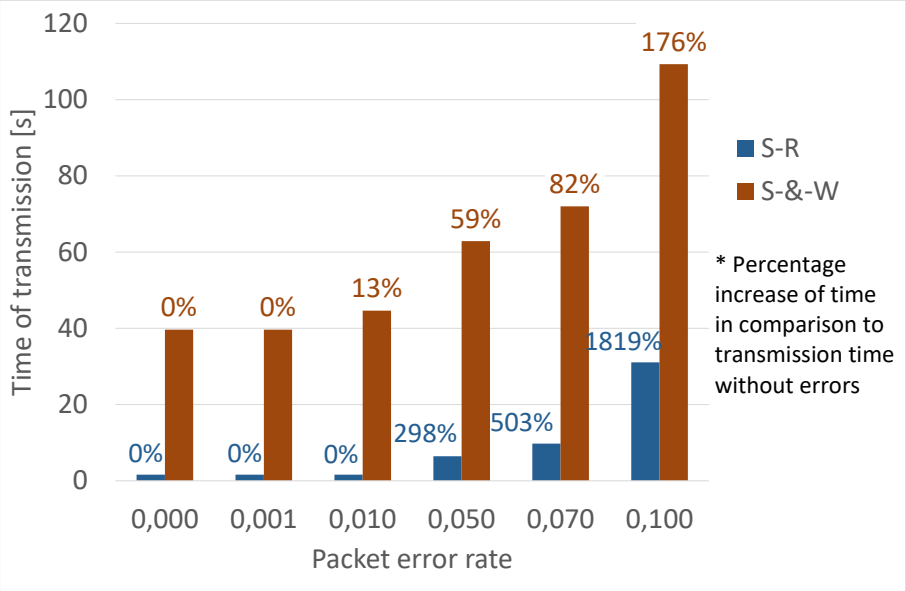


Fig. 6.20: Impact of packet error rate on time needed for transmission of 100000 B with different ARQ methods, percentage increase of needed time in comparison to transmission time without errors

Run configuration, check and note same values as in previous simulation (time and data size needed for 100000 B of data). We can compare it with Fig. 6.20 and Fig. 6.21 (it can be a bit different because generation of errors can differ a little).

Individual task: run the same simulations as now with Stop-and-Wait method and Selective-Repeat method for packet error rate 0.001, 0.010, 0.050, 0.070, 0.100 (this value we just simulate with both methods) and compare needed time and data size (data, headers, ack) needed for transmission of our 100000 B of data. You can compare values with Fig. 6.20 and Fig. 6.21.

d) Individual task: with our implementation of packet error rate 10% and Stop-and-Wait method, what is theoretical number of needed retransmitted packets? Compare it with real number from Wireshark output. Calculations explained in next section.
e) Are there any different angels in comparing methods? Is Stop-and-Wait better in some way?

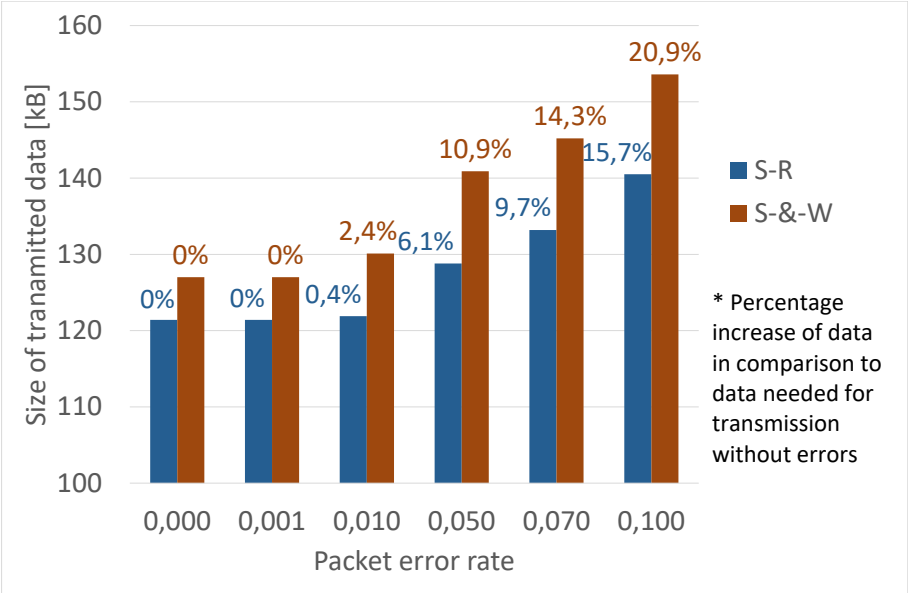


Fig. 6.21: Impact of packet error rate on transmitted data needed for transmission of 100000 B with different ARQ methods, percentage increase of needed data for trasmission in comparison to data for transmission without errors

As we can see in Fig. 6.21, size of transmitted data in is increasing almost the same with both methods. We implemented packet error rate and if we compare amount of sent packets without errors, Stop-and-Wait needs more because every packet is acknowledged. On the other hand Selective-Repeat in our implementation uses cumulative acknowledgements but receiver can process packets quite fast so it sends Ack usually after 2 or 3 packets. Because Selective-Repeat resends only lost (not acknowledged) packet, it generates the same amount of data as Stop-and-Wait method. Only difference is in already mentioned headers but they have smaller size (54 B versus 512 B of data message) in comparison to data messages so the difference is not that significant.

From Fig. 6.20 we can see bigger difference in methods with increasing time much faster with Stop-and-Wait method. The difference is obvious, Stop-and-Wait method waits 1 s for retransmission of packet and can not send any other packet in meantime. On the other hand, Selective-Repeat as it's big advantage can still send packets to the receiver if there is enough space in sliding window (Fig. 6.5) and then sender resends only lost packet and continues with next packet lastly send besides this resent one.

Conclusion

Firstly we implemented base scenario with 2 nodes and generated TCP flow between them. ARQ mechanism used in base scenario was found after generating errors. Receiver was storing out-of-order packets and after resending the missing one the last received packet was acknowledged and not only next after the resent one. It showed us Selective-repeat behavioral with cumulative Ack from basic Go-Back-And model. Stop-and-Wait model was created by lowering receiver window size. For other comparison there was added one other node, server moved to another node and allowed communication between edge nodes by static routing.

We compared communication with 2 and 3 nodes without errors between TCP with different ARQ methods and UDP. According expectations, UDP was the most effective, TCP with Selective-Repeat (and cumulative Ack) second and TCP with Stop-and-Wait least effective. However in real situation, TCP is able to dynamically change parameters of transfer for available bandwidth, amount of errors on medium and latency changes by congestion, error and flow control mechanisms. We also compared two mentioned ARQ methods in error communication with generated packet error rate up to 10%. There was not that bigger amount of all sent data by Stop-and-Wait because with one non acknowledge packet there is one resent packet in each method. Time of whole transmission was increasing much higher by Stop-and-Wait method with increasing generation of loss. It is caused because Selective-Repeat can send more packet at one time and then resent just the missing one but Stop-and-Wait not.

6.2.3 Answers to control questions and assignment check

There are answers to control questions and control of assignments with explanation.

a) It should not be different at all, that is one of the reasons, why we will generate loss. Later we will actually see, there was implemented Selective-Repeat method with cumulative acknowledgements from Go-Back-N model.
b) Usually, in Selective-Repeat method, the sender window is minimum from congestion and receiver window size. The slow start phase does not work in our simulation, because we have big receiver window and slow start phase of congestion mechanism should keep the sender window smaller at the beginning. In implementing Stop-And-Wait method later can be seen, that sender window size changed with receiver window size change. In our case, there was

eventually big loss, but the window of sender is actually in some stage bigger than receiver window (this looks like slow reaction on receiving messages).

c) In this NS-3 version, there should be implemented New Reno congestion control mechanism, which do not require change in configuration on both sides comparing to SACK implementation.

d) Sequence numbers are not only one bit numbers (0 and 1) as in theoretical model. We only forced the method by decreasing the window sizes to one segment, but numbering of sequence numbers stayed the same. Disadvantage could be the waste of needed size for large sequence numbers.

e) We set 0.1 packet error rate (10%) on 1 side and same packet error rate on other side of communication so now it should be lost about 10% of packets. For calculation how many packet will be lost we have to calculate how many packets we have without errors. We can not calculate is exactly with Selective-Repeat method because it sends cumulative Ack for more packets and after error it can send Ack more often. With Stop-and-Wait method we can count it precisely, we have 196 packets for data (size of all data divided by message size, 100000 B / 512 B) and Ack for each packet (196). We also have 6 packets for establishing and cancelling of TCP connection. With rising errors, numbers of packet rises too so the calculation will not be exactly precise with higher error rate. So we have about 400 packets sent in error free communication and 10% is 40 packets. It does not matter if Ack or data packet is lost (sender can not know), there is need for retransmission of packet in each case. We should run the simulation more times because we do not have that many packets for precise statistics but average packet error rate with 10 tried simulations was 44 retransmitted packets. If we compare it with theoretical 40 and note that it will be a bit higher because of more packets due to errors, practical simulation matches theoretical calculations.

f) Stop-and-Wait method do not need such requirements on both sides of communication. If you compare it with Selective-Repeat, when receiver has to store out of order segments, so receiver has larger window and has to deal with sophisticated retransmission methods and protocols.

7 PACKET AND CELL SWITCHING SCENARIO

7.1 Creating scenario in NS-3, used methods and issues

With more nodes, there was needed routing. In NS-3, we can use static or dynamic routing, I used static routing for easy control of direction, which was needed in load-balancing part. It is used UDP echo client and server for communication because there is no additional data like in TCP (initiation of connection, cancelling of connection and acknowledgements) and we can clearly see fragmentation in Wireshark output. Cell switching is simulated by dividing packets into smaller parts by lowering MTU size into 70 B because there is a minimum value in NS-3 with used UDP protocol (same as according to RFC 791). For instance in case of ATM cells are 48 B data and 5 B header and with our simulation 48 B data and 22 B header. There is no module for ATM or other cell switching technique for lowering the header and establishing virtual circuit between two endpoints. Option would be to create own module for it but differences are visible even in this configuration with theoretical comments so I decided not to create it.

7.2 Topology and scenario

Topology in this scenario represents ring topology of 4 point-to-point connections (Fig. 7.1). In first parts there will be used only Node 0, Node 1 and Node 2.

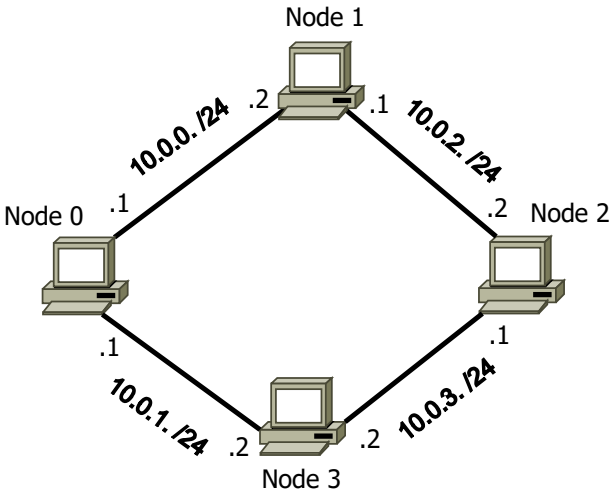


Fig. 7.1: Topology of packet and cell switching scenario

Scenario is divided into three parts:

1. Implementing base of the scenario
2. Simulation tasks
3. Answers to control questions and assignment check

Scenario was created for behavioral comparison of switching methods. It is not possible to generalize behavioral of message transfer with modern devices and advanced protocols. In this scenario there is a comparison of basic models of packet and cell switching with comments of possible advantages, disadvantages or modifications in real situations.

First part is for creating basic simulation environment where there are simulated three hosts on point-to-point channels (Fig. 7.1). Second part is about changing network attributes and comparing packet and cell switching. There are included control questions and individual tasks. Last part contains answers to control questions and explanation or verification of individual assignments.

7.2.1 Implementing base of the scenario

We will create base of the scenario as a preparation for later simulation. We will return to this base setting more times.

Create new file and add modules

Create new document with name "switching_scenario.cc" in folder Scratch. There has to be only one document in Scratch folder for proper functionality.

Insert a modules needed for our scenario (some modules used in later phase), allow logging and define naming space before using main function:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/gnuplot.h"
#include "ns3/ipv4-static-routing-helper.h"
#include "ns3/netanim-module.h"
#include "ns3/stats-module.h"
```

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("switching_scenario");
```

Create nodes

For our purpose we can firstly create only 3 nodes (Node 0, Node 1 and Node 2), we use class *NodeContainer* and create mentioned nodes. This and all other parts should be created inside main function. Code inside main function consists of:

```
int
main (int argc, char *argv[])
{
NodeContainer nodes012;
nodes012.Create (3);

}
```

Create channel between nodes

We have nodes connected using point-to-point topology and for simplicity we can use class *PointToPointHelper*. Set data rate to 5Mbps, delay to 3ms and MTU to 1500 B. Data rate is low because of easier simulation tracking, delay is 0 s as an default value which is not showing real situation and it does not show time sequence in simulation (3 ms could be delay in local network including few switches). MTU 1500 B is standard value for Ethernet, we can set this by code:

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("3ms"));
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(1500));
```

Install TCP/IP stack and create pointers to each node

We can install TCP/IP protocol stack on our nodes and create pointers on each node:

```
InternetStackHelper stack;
stack.Install (nodes012);
Ptr<Node> Node0 = nodes012.Get (0);
Ptr<Node> Node1 = nodes012.Get (1);
Ptr<Node> Node2 = nodes012.Get (2);
```

Set IP addresses

Create network devices from current nodes. Set IP addresses to Node 0 and Node 1 from 10.0.0.0/24 and IP addresses from 10.0.2.0/24 to Node 1 and Node 2 according Fig. 7.1. Also pair each network device with node:

```
NetDeviceContainer devices01;
devices01 = pointToPoint.Install (Node0, Node1);
Ipv4AddressHelper address01;
address01.SetBase ("10.0.0.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces01 = address01.Assign (devices01);

NetDeviceContainer devices12;
devices12 = pointToPoint.Install (Node1, Node2);
Ipv4AddressHelper address12;
address12.SetBase ("10.0.2.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces12 = address12.Assign (devices12);
```

Set static routing

We will use static routing for communication between Node 0 and Node 2. Firstly we will create pointers for mentioned nodes for using protocol IPv4. We will create IPv4 static routing using prepared helper where we can create pointer for static routing on every node (using previous pointer on protocol IPv4). Lastly we have to add static routes for every node. Firstly we define end address and then next hop address and outgoing interface:

```
Ptr<Ipv4> ipv4N0 = Node0->GetObject<Ipv4> ();
Ptr<Ipv4> ipv4N2 = Node2->GetObject<Ipv4> ();
Ipv4StaticRoutingHelper ipv4RoutingHelper;
Ptr<Ipv4StaticRouting>staticRoutingN0 =
ipv4RoutingHelper.GetStaticRouting (ipv4N0);
Ptr<Ipv4StaticRouting> staticRoutingN2 =
ipv4RoutingHelper.GetStaticRouting (ipv4N2);
staticRoutingN0->AddHostRouteTo (Ipv4Address("10.0.2.2"),
Ipv4Address ("10.0.0.2"), 1);
staticRoutingN2->AddHostRouteTo (Ipv4Address("10.0.0.1"),
Ipv4Address ("10.0.2.1"), 1);
```

Create UDP Echo server and client applications

We will create UDP communication between Node 0 and Node 2. We can use class `UdpEchoServerHelper` with setting of server port (ECHO protocol uses port 7).

UDP echo client is created by using class *UdpEchoClientHelper* with remote IP address and remote port. It is also possible to set size of packet, maximum of sent packets and interval between packets. We will use 6 packets with size of 1024B (this will increase later) and interval 1 second between them (clearer visualization with later cell switching simulation). We also use class *ApplicationContainer* for creating server (Node 2) and client (Node 0) applications.

```
UdpEchoServerHelper echoServer (7);
ApplicationContainer serverApplication = echoServer.Install (Node2);
serverApplication.Start (Seconds (1.0));
serverApplication.Stop (Seconds (20.0));

UdpEchoClientHelper echoClient (interfaces12.GetAddress (1), 7);
echoClient.SetAttribute ("MaxPackets",UintegerValue(3));
echoClient.SetAttribute ("Interval",TimeValue(Seconds (1)));
echoClient.SetAttribute ("PacketSize",UintegerValue(1024));
ApplicationContainer clientApplication = echoClient.Install (Node0);
clientApplication.Start (Seconds (1.0));
clientApplication.Stop (Seconds (20.0));
```

Logging data transfer for Wireshark

Enable capturing the data on point-to-point line to pcap document:

```
pointToPoint.EnablePcapAll ("switchingComparision");
```

UDP echo client and server communication displayed in console

It will display communication between UDP echo server and client in console. We can add this code at the beginning of main function:

```
LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
```

Set start and end of simulator

```
Simulator::Run ();
Simulator::Destroy ();
return 0;
```

Setting run configuration

Firstly press Run -> Run for starting the project. It will not work, but now we can set up project in Run configuration menu by Run -> Run Configuration -> Search Project -> switching_scenario -> Run (Fig. 7.2). After this step, configuration file is set and only Run -> Run or green button will work. Output of UDP ECHO

communication from console can be seen in Fig. 7.3, we can see 3 ECHO requests by client and reply for each request done by server.

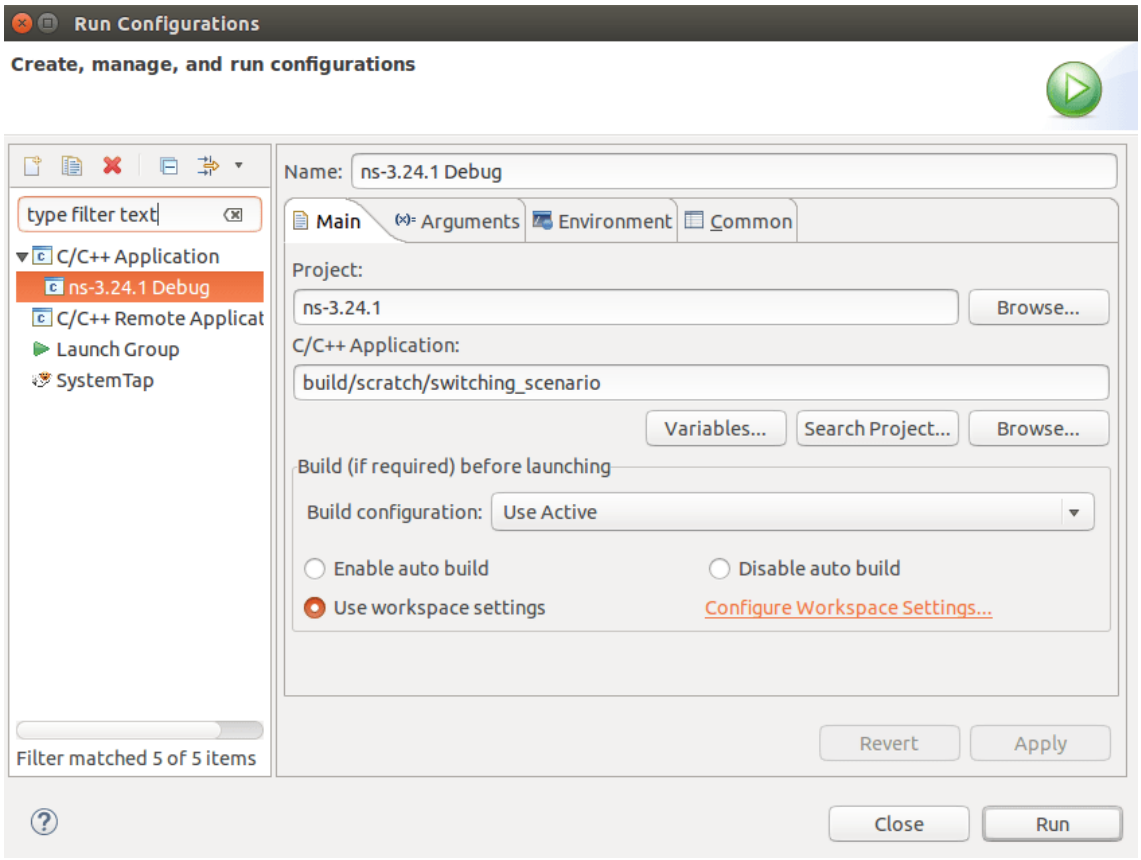


Fig. 7.2: Set-up of configuration file

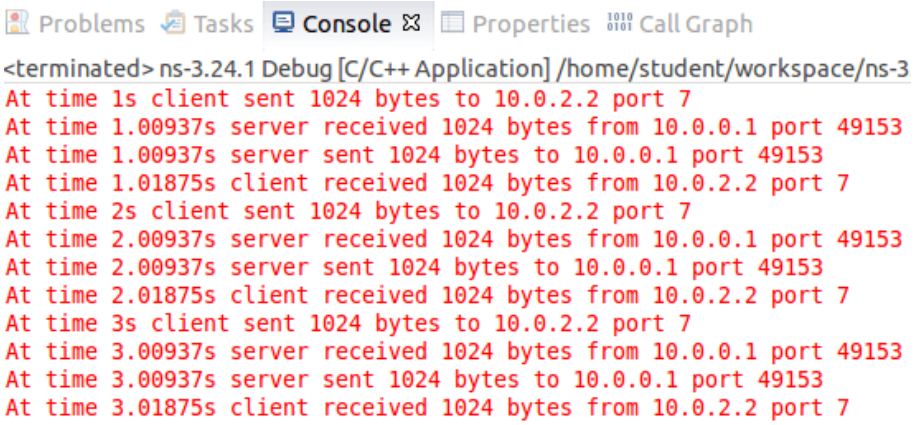


Fig. 7.3: Console output of UDP ECHO communication

7.2.2 Simulation tasks

In this part, we will compare switching techniques, especially packet switching and cell switching. Control questions and individual tasks are listed in italics with answers in next section.

Analyze current environment

Run our configuration for creating pcap file for analysis in Wireshark. There are generated four pcap files, one for each interface of node. After opening any of created pcap files, we can see a simple communication (UDP echo and reply) same as from console output (Fig. 7.3).

If we consider 4 basic ways of message transfer (Circuit switching, Message switching, Packet switching and Cell switching), which option/s does current behavior match and why? Because there is no initiation of connection (can be seen in Wireshark), it can not be circuit switching. Because of transferred packets are bigger than cell, it can not be cell switching either. Now because each message is delivered in one packet, we can not say if it is message switching or packet switching yet.

From Wireshark output, it is still not possible to decide which exactly from 4 mentioned ways of message transfer it is. We can change packet size in setting of echo client to 2048 B (code listed below) and check Wireshark output again. Now we can see fragmentation of message to more packets (Fig. 7.4) so it is clear our current environment uses packet switching way of message transfer. Changing of packet size is done by this code:

```
echoClient.SetAttribute ("PacketSize",UintegerValue(2048));
```

No.	Time	Source	Destinatior	Protocol	Length	Info
1	0.000000	10.0.0.1	10.0.2.2	IPv4	1502	Fragmented IP protocol (proto=UDP 17, off=0, ID=0000) [Reassembled in #2]
2	0.002403	10.0.0.1	10.0.2.2	ECHO	598	Request
3	0.022569	10.0.2.2	10.0.0.1	IPv4	1502	Fragmented IP protocol (proto=UDP 17, off=0, ID=0000) [Reassembled in #4]
4	0.023526	10.0.2.2	10.0.0.1	ECHO	598	Response
5	1.000000	10.0.0.1	10.0.2.2	IPv4	1502	Fragmented IP protocol (proto=UDP 17, off=0, ID=0001) [Reassembled in #6]
6	1.002403	10.0.0.1	10.0.2.2	ECHO	598	Request
7	1.022569	10.0.2.2	10.0.0.1	IPv4	1502	Fragmented IP protocol (proto=UDP 17, off=0, ID=0001) [Reassembled in #8]
8	1.023526	10.0.2.2	10.0.0.1	ECHO	598	Response

Fig. 7.4: Packet fragmentation with packet size bigger than MTU

Emulation of cell switched environment

There is no option how to switch to the cell switching transfer in NS-3 but it is

possible to change network attributes to emulate approximately behavioral. There is more differences between general model of cell and packet switched networks but modern devices (many of them proprietary without possibility to know exactly how it works) and protocols (for example error handling in TCP) often combine advantages of different mentioned models. First obvious difference is size of 1 packet/cell.

In our code, we can accomplish smaller fixed size of 1 cell (try to reach same size as in ATM which is cell switching technique) by lowering of Maximum Transmission Unit (MTU). When we try to change MTU to 53 B, communication is not working. NS-3 module regarding IP protocol is based on RFC standards, where there is (RFC 791) declaration of minimal size 68 B for forwarding datagram without fragmentation (header up to 60 B, minimum data 8 B). There is also declaration of minimum size of a datagram (576 B) which has to be every destination device able to receive in one piece or in fragments to be reassembled. In NS3, it is really according RFC 791 possible to have minimal size 68 B of datagram, MTU 1500 (usually used in Ethernet and also maximum for our scenario) is supported by nodes in NS3. According this behavioral we have to use at least 68 B MTU for our purposes. We will use 70 B MTU where we have 48 B of data (same as in ATM) and 22 B of header. This will be our first tool for comparison (used in following part).

a) How is cell switched transfer of messages different from packet switched?
b) What is usual size of MTU in packet switched networks and how big is size of cell?

Impact of different size of packet and cell

In this part, we will simulate current scenario environment with MTU 1500 B (as a packet switching simulation) and MTU 70 B (as a cell switching simulation) and compare outputs from Wireshark. We can let message size of 2048 B and interval between messages to 1 second (for not mixing requests and respond messages in cell switching so we can clearly see amount of cells per one message) and run simulation (Wireshark output shown in Fig. 7.4).

After running this first configuration, for possible comparison and not overwriting of pcap files, we can rename pcap file in code (code for renaming below). Finally for simulation of cell switched environment, change MTU to 70 B. Changes in code are:

```
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(70));
pointToPoint.EnablePcapAll ("cellSwitching");
```

Now simulate cell switching (with MTU changed to 70 B) and compare both methods in Wireshark, expected output from simulation of cell switching can be seen in

Fig. 7.5. In this simulation with sending of big message with no errors on medium, we can clearly see that cell switching is less efficient in amount of sent data, because a lot more headers is created. Graph of sent headers size for 1 message of 2048 B with packet switching and MTU 1500 B, emulated cell switching (MTU 70 B) and theoretical value of ATM as a cell switching technique (without possible headers from protocol of higher layer) is in Fig. 7.6. There is also included MTU 576 B which is minimal requirement for IPv4 nodes and links (RFC 791) not to fragment message same as 1280 B for IPv6 (RFC 2460). In real situation, there are different messages of different protocols and size, this was just first example of possible advantages/disadvantages, more comparison will be in next parts. For example error handling is crucial in switching methods and different size of packet and cells has big impact. In our simulation when 1 message is half size bigger with emulated cell switching (MTU 70 B) and even 1 error in 1 message means lost whole message, with same error rate the configuration with MTU 1500 B would be better because of less transmitted data and less chance of error. In real situation there are many possible used protocols and behavioral for retransmitting error packets (for example ARQ methods).

c) How big are headers in each case (MTU 1500 B and MTU 70 B) and how many Bytes were transferred in each method for data and headers for 1 message? Simulate also minimal required supported MTU without fragmenting for IPv4 (MTU 576 B) and IPv6 (MTU 1280 B), compare values with Fig. 7.6.

No.	Time	Source	Destinatior	Protocol	Lengtl	Info
36	0.003920	10.0.0.1	10.0.2.2	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=1680, ID=0000) [Reassembled in #43]
37	0.004032	10.0.0.1	10.0.2.2	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=1728, ID=0000) [Reassembled in #43]
38	0.004144	10.0.0.1	10.0.2.2	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=1776, ID=0000) [Reassembled in #43]
39	0.004256	10.0.0.1	10.0.2.2	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=1824, ID=0000) [Reassembled in #43]
40	0.004368	10.0.0.1	10.0.2.2	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=1872, ID=0000) [Reassembled in #43]
41	0.004480	10.0.0.1	10.0.2.2	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=1920, ID=0000) [Reassembled in #43]
42	0.004592	10.0.0.1	10.0.2.2	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=1968, ID=0000) [Reassembled in #43]
43	0.004704	10.0.0.1	10.0.2.2	ECHO	62	Request
44	0.017139	10.0.2.2	10.0.0.1	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=0, ID=0000) [Reassembled in #86]
45	0.017251	10.0.2.2	10.0.0.1	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=48, ID=0000) [Reassembled in #86]
46	0.017363	10.0.2.2	10.0.0.1	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=96, ID=0000) [Reassembled in #86]
47	0.017475	10.0.2.2	10.0.0.1	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=144, ID=0000) [Reassembled in #86]
48	0.017587	10.0.2.2	10.0.0.1	IPv4	70	Fragmented IP protocol (proto=UDP 17, off=192, ID=0000) [Reassembled in #86]

Fig. 7.5: Message is fragmented to more packets with MTU 70B

Animation of message transfer

We can use graphic animation tool called NetAnim which can display flow of packets between nodes. We will use class *`AnimationInterface`* for creating xml file, which can be later opened in NetAnim. We can also set-up horizontal and vertical position of nodes for animation. It is possible to enable meta data which can provide us information of IP addresses and port numbers. For not mixing messages in the same

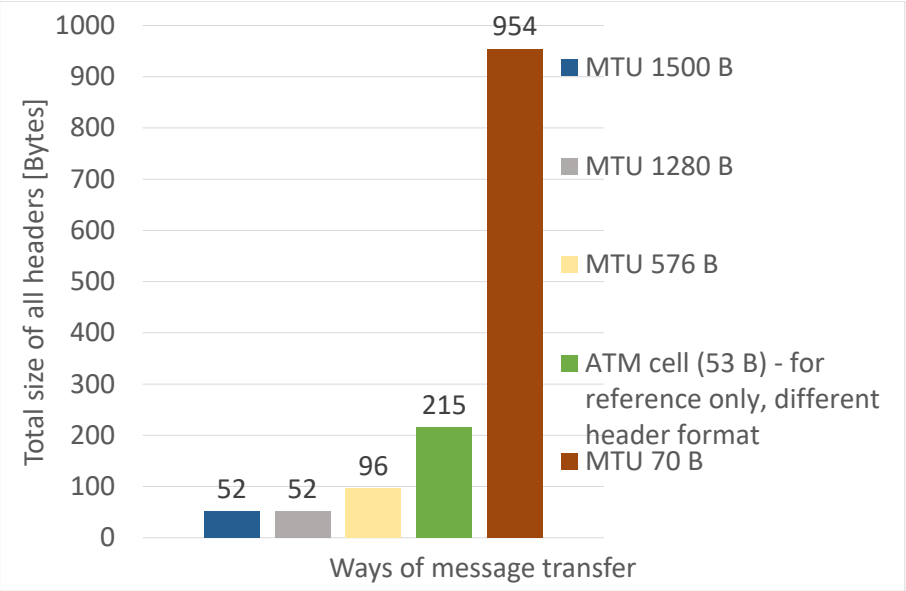


Fig. 7.6: Comparison of sent headers size for 2048 B message with packet switching with different values of MTU and cell switching

time so we can see clearly communication in NetAnim with information set MTU to 1500 B and message size to 1024 B. Code below should be inserted after code for creating pointers to specific nodes (MTU and message size change at current position) which is used in animation as an identification of each node.

```
AnimationInterface animation ("Switching_scenario_animation.xml");
animation.EnablePacketMetadata (true);
animation.SetConstantPosition (Node0, 0, 0);
animation.SetConstantPosition (Node1, 10, -10);
animation.SetConstantPosition (Node2, 20, 0);

echoClient.SetAttribute ("PacketSize",UintegerValue(1024));
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(1500));
```

Run simulation and use NetAnim to see our communication (Fig. 7.7). AnimationInterface WARNING is shown in console, you can ignore this warning. We will be using animation later in next section, you can delete or comment this code for now.

Latency comparison in switching methods

Basic simplified principle is that with packet switching, there is control of whole packet in interconnected device, which takes longer than simplified principle in cell switching, where there is only control of header of cell in interconnected router. Nowadays, there are advanced devices (many of them proprietary with no exact

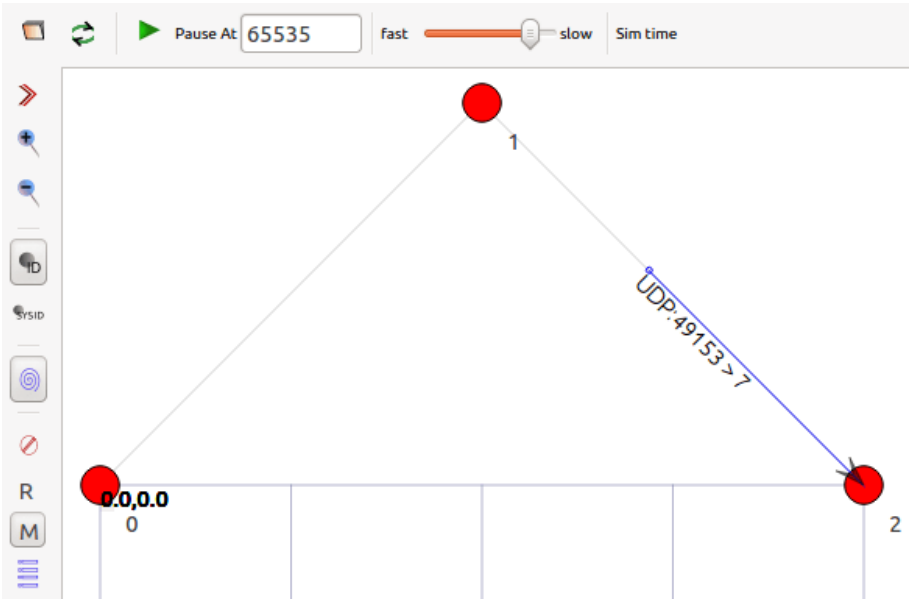


Fig. 7.7: Animation of message transfer in NetAnim

explanation how it works), protocols and techniques with changing behavioral, which can not be sort in basic switching models regarding latency in normal situation or with error handling.

However, in NS3, there is a possibility to simulate scenarios for understanding some latency issues with changing network parameters or topology. Set 3 messages with size of 2048 B each and MTU to 1500 B. Firstly we will simulate latency on only point-to-point connection between Node 0 and Node 1 (later with more) which means we have to change UDP echo server from Node 2 to Node 1:

```
ApplicationContainer serverApplication = echoServer.Install (Node1);
UdpEchoClientHelper echoClient (interfaces01.GetAddress (1), 7);
echoClient.SetAttribute ("MaxPackets",UintegerValue(3));
echoClient.SetAttribute ("PacketSize",UintegerValue(2048));
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(1500));
```

Run configuration and rename pcap files so we can later compare simulation with 70 B MTU. Change of code for changing MTU to 70 B:

```
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(70));
```

Run simulation with MTU 70 and then compare latency (from sending Echo request until receiving Echo reply to same message on client) of switching methods (from Wireshark output). Which method has better latency in this situation and why? We can see better latency with simulated packet switching (Fig. 7.8 combined with

later simulations). Reason is obvious because with simulated cell switching (MTU 70 B) the data with headers sent for 1 message is almost 1.5 times higher (3002 B versus 2100 B) than data sent with packet switching (MTU 1500 B).

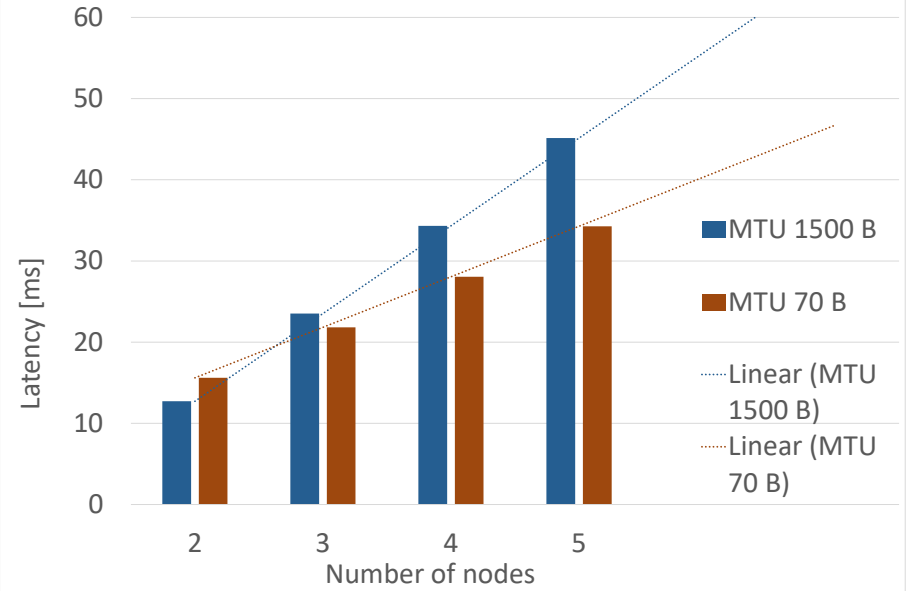


Fig. 7.8: Latency of sending UDP echo request until receiving echo reply with different number of nodes

Now we can try simulation with interconnected nodes. Firstly change UDP echo server back to Node 2 so Node 1 is interconnected node between client and server:

```
ApplicationContainer serverApplication = echoServer.Install (Node2);
UdpEchoClientHelper echoClient (interfaces12.GetAddress (1), 7);
```

We will again execute simulation with MTU 70 B and MTU 1500 B and explore again latency.

```
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(1500));
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(70));
```

Run configurations with both mentioned values of MTU and explore latency again in Wireshark. What is better and why in this situation? As we can see in Fig. 7.8, lower latency has now our emulation of cell switching (MTU 70 B). How is it possible when for 1 message of cell switching (70 B MTU) is needed almost half more transferred data (3002 B versus 2100 B) because of headers than for packet switching (MTU 1500 B)? According basic switching model this could be because of middle node only check header with cell switching and not all packet as with packet switching (not necessary with nowadays devices) but we know this is not our case because of using only lowering MTU. Another possible reason is

because of processing time of delivered big packet on interconnected device and possibility to send smaller messages immediately and not wait for receiving and check of bigger message. After exploring latency on interconnected node in Wireshark (time between receiving last packet and sending last packet of request/response) we can see that interconnected node could sent smaller 70 B messages to the next device and processing did not take that long as processing two bigger packets sent with MTU 1500 B. As we can see from Wireshark output from Node 1 with MTU 1500 B on both sides (Fig. 7.9 and Fig. 7.10), latency between received last packet of Echo request and sending last Echo request to Node 2 (0.002403 s - 0.000956 s) is 1.447 ms (same with Echo reply latency) but if we compare the same in Wireshark with MTU 70 B, latency is 0.013 ms which is significant difference. So even packets with configuration of MTU 1500 B are delivered sooner to interconnected node (Node 1) and it is almost 1.5 less data cumulated, cell switching (MTU 70 B) with message divided into smaller parts has better latency in this case.

No.	Time	Source	Destinatior	Protocol	Length	Info
1	0.000000	10.0.0.1	10.0.2.2	IPv4	1502	Fragmented IP protocol (proto=UDP 17, off=0, ID=0000) [Reassembled in #2]
2	0.000956	10.0.0.1	10.0.2.2	ECHO	598	Request
3	0.011763	10.0.2.2	10.0.0.1	IPv4	1502	Fragmented IP protocol (proto=UDP 17, off=0, ID=0000) [Reassembled in #4]
4	0.014166	10.0.2.2	10.0.0.1	ECHO	598	Response

Fig. 7.9: Wireshark output with UDP echo messages on Node 1 interface towards Node 0

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.1	10.0.2.2	IPv4	1502	Fragmented IP protocol (proto=UDP 17, off=0, ID=0000) [Reassembled in #2]
2	0.002403	10.0.0.1	10.0.2.2	ECHO	598	Request
3	0.011763	10.0.2.2	10.0.0.1	IPv4	1502	Fragmented IP protocol (proto=UDP 17, off=0, ID=0000) [Reassembled in #4]
4	0.012720	10.0.2.2	10.0.0.1	ECHO	598	Response

Fig. 7.10: Wireshark output with UDP echo messages on Node 1 interface towards Node 2

We can now try to add another node in the middle and see change of latency with more nodes. We can add Node 3 as in Fig. 7.1 and change UDP echo server to Node 3. Then, we have 2 interconnected nodes (Node 1 and Node 2).

d) Individual task: as mentioned above, add Node 3 to topology as it is in Fig. 7.1, change UDP echo server to Node 3 and set communication between devices so we have 2 interconnected devices between client and server (example of code can be seen in another section.

After completing previous task, set firstly MTU 1500 B and then MTU 70 B:

```
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(1500));
```

```
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(70));
```

Run configuration with both MTU and compare latency. We can see in Fig. 7.8 that only with 2 nodes there was better latency with MTU 1500 B but after adding another interconnected nodes the latency is increasing in linear way with both MTU sizes. We can see the latency with MTU 1500 B rising quicker than with MTU 70 B. This is only 1 example of comparing latency in specific scenario, there is a lot of techniques how to process and transfer message and it is not possible to generalize it for 1 switching method. With different switching technique on interconnected node, different size of message or used protocols the result could be different.

In general basic model of switching methods, cell switching establishes virtual circuit between source and destination node before start of transmitting and all cells for 1 connection are transmitted this way. On the other hand with packet switching, packets for 1 message can be sent in different ways. With modern devices and advanced protocols, load balancing is also possible in cell switching. What are advantages and disadvantages of sending packets for 1 message in different directions? Advantage could be dividing of necessary process performance to more devices and interfaces which can cause better latency a possible lower congestion of network. Disadvantage can be more requirements for receiving node, it has to put packets to the right order because it does not have to be in same order delivered.

We can try to simulate transmission of same amount of messages with and without load balancing and check latency. Now we will connect Node 3 to Node 0 so we have 2 ways how to reach our new server of UDP echo (Node 2). We have to set new interface between Node 0 and Node 3 and change static routing. We will set routing that from Node 0 if we want to reach Node's 2 IP address of interface towards Node 1 (10.0.2.2) we will go through Node 1 and if we want to reach Node's 2 other interface (10.0.3.1) we will go through Node 3 (and same routing on the way back). Firstly set server on Node's 2 interface towards Node 1. We will also use animation of transfer in NetAnim so we can see later that communication is load balanced, we can use the previous code with added Node 3:

```
NetDeviceContainer devices03
devices03 = pointToPoint.Install (Node0, Node3);
Ipv4AddressHelper address03;
address03.SetBase ("10.0.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces03 = address03.Assign (devices03);

ApplicationContainer serverApplication = echoServer.Install (Node2);
UdpEchoClientHelper echoClient (interfaces12.GetAddress (1), 7);
```

```
staticRoutingN0->AddHostRouteTo (Ipv4Address("10.0.2.2"),
Ipv4Address ("10.0.0.2"), 1);
staticRoutingN2->AddHostRouteTo (Ipv4Address("10.0.0.1"),
Ipv4Address ("10.0.2.1"), 1);
staticRoutingN0->AddHostRouteTo (Ipv4Address("10.0.3.1"),
Ipv4Address ("10.0.1.2"), 2);
staticRoutingN2->AddHostRouteTo (Ipv4Address("10.0.1.1"),
Ipv4Address ("10.0.3.2"), 2);

AnimationInterface animation ("Switching_scenario_animation.xml");
animation.EnablePacketMetadata (true);
animation.SetConstantPosition (Node0, 0, 0);
animation.SetConstantPosition (Node1, 10, -10);
animation.SetConstantPosition (Node2, 20, 0);
animation.SetConstantPosition (Node3, 10, 10);
```

Firstly we will simulate communication (with MTU 1500 B) from Node 0 to Node 2 without load balancing. Set 2 messages of 2048 B each and interval 0 (so we can clearly see cumulated latency for 2 messages since sending until receiving):

```
pointToPoint.SetDeviceAttribute("Mtu", UintegerValue(1500));
echoClient.SetAttribute ("Interval",TimeValue(Seconds (0)));
echoClient.SetAttribute ("MaxPackets",UintegerValue(2));
echoClient.SetAttribute ("PacketSize",UintegerValue(2048));
```

Run configuration and check console output how much time was needed for transfer (ignore animation warning). As we can see in console output (Fig. 7.11), the time needed for sending and receiving two 2048 B UDP echo requests and responses is 26.89 ms.

Now we will try to simulate load balancing. We can simulate load balanced same amount of data by dividing client request into two direction with each of half size of data previously in one way. Our static routing is prepared so if we try to reach Node's 2 one address, it will go different way then reaching second address. We have to set up two client applications with different destination interface on Node 2:

```
UdpEchoClientHelper echoClient (interfaces12.GetAddress (1), 7);
echoClient.SetAttribute ("MaxPackets",UintegerValue(1));
echoClient.SetAttribute ("PacketSize",UintegerValue(2048));
echoClient.SetAttribute ("Interval",TimeValue(Seconds (0)));
ApplicationContainer clientApplication = echoClient.Install (Node0);
clientApplication.Start (Seconds (1.0));
```

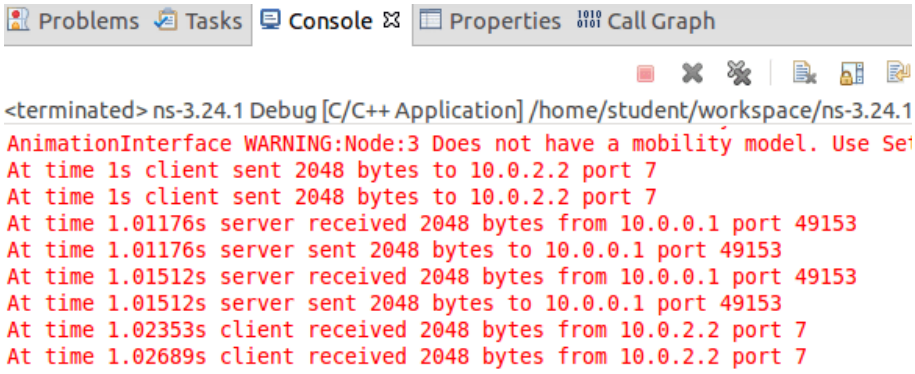


Fig. 7.11: Console output with 2 UDP echo messages transmitted via 1 way between Node 0 and Node 2

```
clientApplication.Stop (Seconds (20.0));

UdpEchoClientHelper echoClient2 (interfaces23.GetAddress (0), 7);
echoClient.SetAttribute ("MaxPackets",UintegerValue(1));
echoClient.SetAttribute ("PacketSize",UintegerValue(2048));
echoClient.SetAttribute ("Interval",TimeValue(Seconds (0)));
ApplicationContainer clientApplication2 = echoClient2.Install (Node0);
clientApplication2.Start (Seconds (1.0));
clientApplication2.Stop (Seconds (20.0));
```

Run simulation and check NetAnim with our generated xml file ("Switching_scenario_animation.xml") if there is really load balancing (Fig. 7.12). Check in console output (Fig. 7.13) how much time was needed now (Fig. 7.14). We can see that now we needed 23.25 ms which is really lower than without load balancing (26.89 ms), with slower link and higher amount of messages the difference would be even more significant. With errors in communication, latency would be very depend on technique used for retransmission if needed, for example with different ARQ methods and different communication type and protocols.

Conclusion

Firstly we created point to point topology with 3 nodes, set static routing between edge nodes and created UDP communication between edge nodes. We analysed basic scenario and discovered, packet are fragmented when message is bigger than 1500 B. Then, we emulated approximately cell behavioral by lowering MTU. We simulated scenarios with different size of packet and cell and calculated needed total size of all headers for one 2048 B message. Bigger MTU, less headers needed without errors

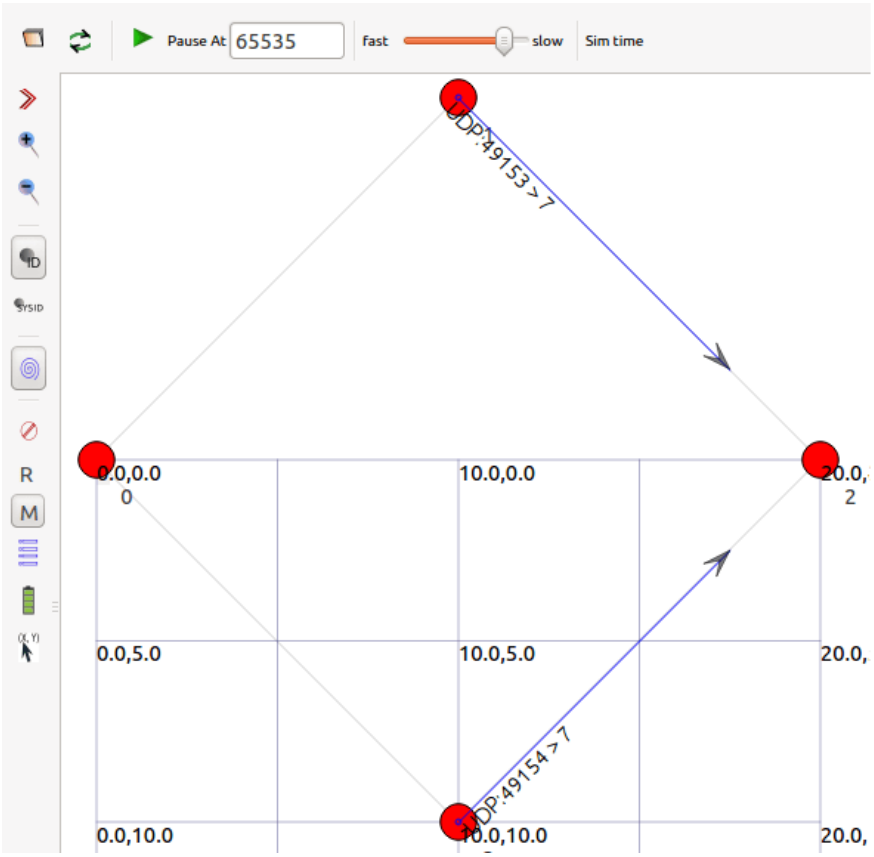


Fig. 7.12: NetAnim output of load balanced UDP messages between Node 0 and Node 2

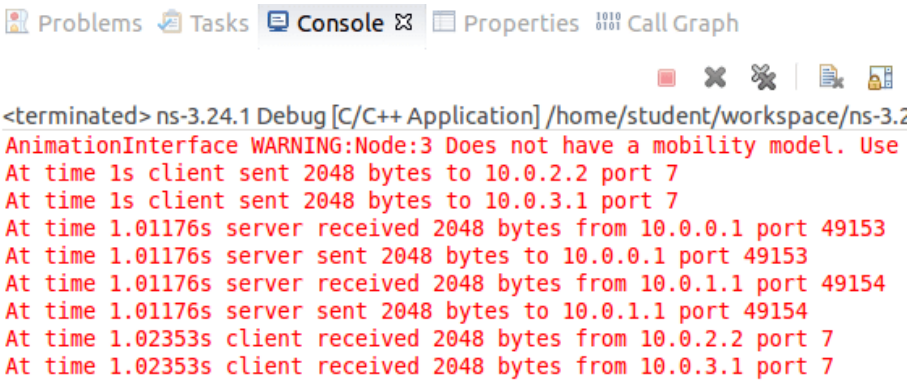


Fig. 7.13: Console output with 2 UDP echo messages transmitted each message different way between Node 0 and Node 2

of course. Real cells are small and with messages of big size, packet switching is more effective with total size of all headers needed in most cases and without errors. We also compared latency with switching methods with different number of nodes. Cell switching was worse with 2 nodes (more headers created, slower transmission)

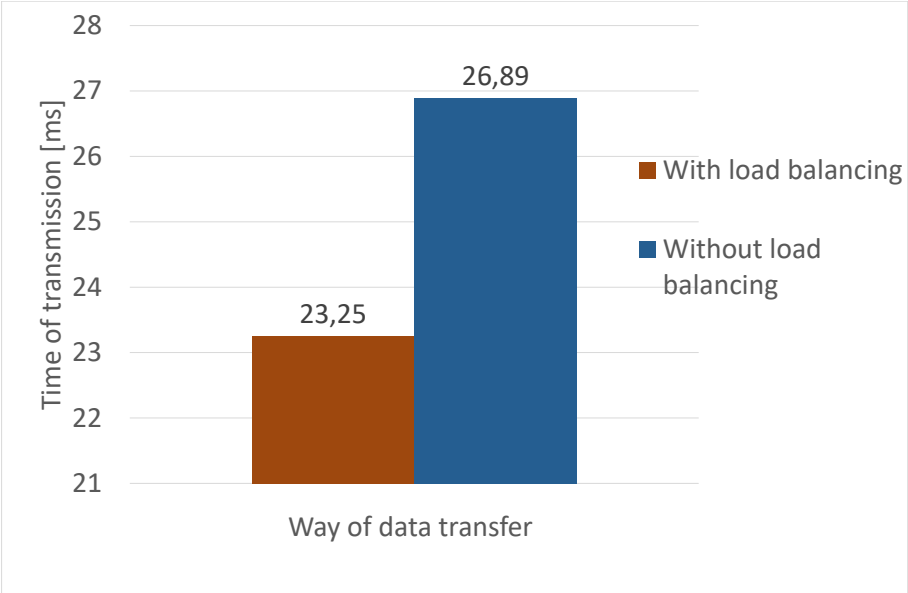


Fig. 7.14: Comparison of needed time for transmission of two 2048 B messages with and without load balancing

but better with more nodes because it is possible to sent small messages in middle nodes quicker than waiting for big message (in our case 1500 B). We simulated load balancing and check better latency than without because of split of needed processing for twice more interfaces and twice less loaded communication way.

7.2.3 Answers to control questions and assignment check

There are answers to control questions and control of assignments with explanation.

a) Cells have smaller and fixed length in comparison to packet switched network. In cell switching, on interconnected network devices, there is control only of header but in packet switched environment, there is control of whole packet (considered basic models, can be different with different devices/technologies). This behavioral has impact for delay same as possible delay caused by different sending and receiving procedure (more information later).
b) MTU is usually 1500 B in packet switched networks. Cell size for example in ATM is 53 B (48 B data, 5 B header).
c) If we have MTU 1500 B (packed switching simulation), we can see that there are 2 packets for 1 message. First message has 1502 B, where 1480 B is data and 22 B header. Next message is 598 B, where 568 B is data (2048 B - 1480 B) and 30 B is header (bigger than with first one because of echo request/replay addition in header). So for 1 message of 2048 B, we need to send 2100 B. If

we simulate cell switching (MTU is 70 B), for 1 message of 2048 B, there was used 42 cells of size 70 B, where 48 B is data and 22 B is header and 43th cell of size 62 B, where data is 32 B and header 30 B.
When we count size of sent packets for 1 message, it is 2100 B (52 B headers), size of sent cells of 1 message is 3002 B (954 B headers). From this results, it looks very inefficient for cells (954 B in comparison to 52 B headers). In real situation it would be different, because cells have only 5 B headers. We can calculate that for example in ATM (without counting headers of possible higher layer protocol) for 2048 B message it would be 43 of 53 B cells, which is 2279 B (215 B for headers), so it is still more than 4 times more transmitted Bytes for 1 message in this case in comparison to size of not message data with packet switching and MTU 1500 B.
When we run simulation with MTU 576 B, we can see 3 messages of 574 B (data 552 B and headers 22 B) and 1 message of 422 B (data 392 B and header 30 B). After running simulation with MTU 1280 we can see 1 message of 1278 B (data 1256 B and header 22 B) and 1 message of 822 B (data 792 B and header 30 B). We can see that with MTU 576 B there was needed 3 headers of 96 B together (1 whole message was 2144 B) and with MTU 1280 B there were 2 headers of 52 B together (1 whole message was 2100 B same as with MTU 1500 because it was possible to split it also in 2 messages). Calculations of data needed for transfer of 1 message of 2048 B are visualized in Fig. 7.6.

d) We have to create another node same as the others before, install IP stack, create pointer for node for later creation of point-to-point link between Node 2 and Node 3. We have to also set IP address for communication between Node 2 and Node 3, set static routing for communication between Node 0 and Node 3 and change UDP echo server to Node 3:

```
NodeContainer nodes0123;
nodes0123.Create (4);


stack.Install (nodes0123);
Ptr<Node> Node0 = nodes0123.Get (0);
Ptr<Node> Node1 = nodes0123.Get (1);
Ptr<Node> Node2 = nodes0123.Get (2);
Ptr<Node> Node3 = nodes0123.Get (3);


NetDeviceContainer devices23;
devices23 = pointToPoint.Install (Node2, Node3);
```

```
Ipv4AddressHelper address23;
address23.SetBase ("10.0.3.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces23 =
address23.Assign (devices23);


Ptr<Ipv4> ipv4N3 = Node3->GetObject<Ipv4> ();
Ptr<Ipv4StaticRouting>staticRoutingN3 =
ipv4RoutingHelper.GetStaticRouting (ipv4N3);


staticRoutingN0->AddHostRouteTo(Ipv4Address("10.0.3.2"),
Ipv4Address ("10.0.0.2"), 1);
staticRoutingN2->AddHostRouteTo(Ipv4Address("10.0.0.1"),
Ipv4Address ("10.0.2.1"), 1);
staticRoutingN1->AddHostRouteTo(Ipv4Address("10.0.3.2"),
Ipv4Address ("10.0.2.2"), 2);
staticRoutingN3->AddHostRouteTo(Ipv4Address("10.0.0.1"),
Ipv4Address ("10.0.3.1"), 1);


ApplicationContainer serverApplication =
echoServer.Install (Node3);
UdpEchoClientHelper echoClient
(interfaces23.GetAddress (1), 7);
```

8 CONCLUSION

This thesis provides an option to create simulations in NS-3 environment, which includes large scale of possibilities with community creating a lot of new modules.

Theoretical parts include everything needed for understanding of each scenario. Scenarios were created with focus on individual work but every part is detailed explained. Individual tasks for understanding problematic and NS-3 implementation covers additional parts and code is always included in next section.

In first scenario, there is an option to investigate TCP communication with various ARQ mechanisms used. In used NS-3 version, there is used Selective-Repeat method with cumulative acknowledgement in default settings, which is shown after generating loss. Stop-and-Wait method was created after decreasing of send and receive window. Methods are practically and theoretically compared. UDP was also used for comparison of transfer without loss. For generating of loss, I used class Errormodel with byte error rate and packet error rate. There was an option to simulate UDP transmission in error environment with higher layer protocol ensuring retransmission of lost/corrupted packet, for example Trivial File Transfer Protocol (TFTP). Unfortunately, there is not any implementation of retransmission in higher layer protocols without own module in current NS-3.

Second scenario compares ways of message transfer, mostly Packet and Cell switching. Cell mode was simulated by lowering MTU. Size of cell is 53 B but MTU in NS-3 is restricted to the same value as defined in RFC791 for IP protocol, which is 68 B. After simulating cell, there was finding of impact of different size of packet and cell, where I used big size of one message (2048 B) in comparison to one cell for bigger difference between needed cumulated data including headers for transfer of one message. For latency comparison, there was needed to add more nodes, because the important difference (as an latency impact) is on interconnected device and it's receiving, processing and forwarding of packet/cell.

NS-3 provides wide scale of options but it also brings a lot of possible limitations. In first scenario with generation of packet and byte error rate, NS-3 was not behaving according theoretical expectations with higher error rate. Lowering transfer speed or higher latency as possible slow reaction time did not work. Eventually, I had to use Wireshark for finding when these errors start to occur, which showed to be 10% of packet error rate which was still possible to use for creating useful graph showing differences.

Because there are not many documented scenarios in NS-3, this could not only help with understanding of obtained topic in scenarios but also as an example how to create simulations in NS-3 and work with different modules. Scenarios could be used in university computer network courses.

BIBLIOGRAPHY

[1] *What is ns-3. Ns-3* [online]. © 2011-2015 [cit. 2016-11-06]. Available from URL: <https://www.nsnam.org/overview/what-is-ns-3/>.

[2] *Ns-3 tutorial. Ns-3* [online]. © 2011-2015 [cit. 2016-12-05]. Available from URL: <https://www.nsnam.org/tutorials/geni-tutorial-part1.pdf>.

[3] *Introduction. Ns-3* [online]. © 2011-2015 [cit. 2016-12-05]. Available from URL: <https://www.nsnam.org/docs/tutorial/html/introduction.html>.

[4] *Ns-3 tutorial, release ns-3.21. Ns-3* [online]. © 2011-2015 [cit. 2016-12-05]. Available from URL: <https://www.nsnam.org/docs/release/3.21/tutorial/ns-3-tutorial.pdf>.

[5] *RFC 793: TRANSMISSION CONTROL PROTOCOL* [online]. September 1981 [cit. 2016-12-05]. Available from URL: <https://tools.ietf.org/html/rfc793>.

[6] FOROUZAN, Behrouz A. *TCP/IP protocol suite. 4th ed. Boston: McGraw-Hill Higher Education,* c2010. ISBN 00-733-7604-3.

[7] JEŘÁBEK, J. *Komunikační technologie. Skriptum FEKT Vysoké učení technické v Brně,* c2016, s. 1-172

[8] *RFC 3366: Advice to link designers on link Automatic Repeat reQuest (ARQ)* [online]. August 2002 [cit. 2016-12-06]. Available from URL: <https://tools.ietf.org/html/rfc3366>.

[9] KABELOVÁ, Alena a Libor DOSTÁLEK. *Velký průvodce protokoly TCP/IP a systémem DNS. 5., aktualiz. vyd. Brno: Computer Press,* c2008. ISBN 978-80-251-2236-5.

[10] *How Packet Switching Works on Computer Networks. INTERNET & NETWORK* [online]. c2017 [cit. 2017-07-25]. Available from URL: <https://www.lifewire.com/packet-switching-on-computer-networks-817938>.

[11] *Asynchronous Transfer Mode (ATM) Switching. Cisco* [online]. c1999 [cit. 2017-07-25]. Available from URL: <https://www.cisco.com/cpress/cc/td/cpress/fund/ith2nd/it2420.htm#xtocid4900>.

[12] *Cut-Through and Store-and-Forward Ethernet Switching for Low-Latency Environments. Cisco Nexus 5000 Series Switches, White Papers* [online]. c2017 [cit. 2017-07-25]. Available from URL: <http://www.cisco.com/c/en/us/products/collateral/switches/nexus-5020-switch/white_paper_c11-465436.html>.

[13] CASTELLI, Matthew. *LAN switching first-step. Indianapolis, IN: Cisco Press,* c2005. ISBN 1-58720-100-3.

LIST OF SYMBOLS, PHYSICAL CONSTANTS AND ABBREVIATIONS

AAL	ATM Adaptation Layers
ABR	Available Bit Rate
ARQ	Automatic Repeat Request
ATM	Asynchronous Transfer Mode
CBR	Constant Bit Rate
CPL	Cell Loss Priority
CRC	Cyclic Redundancy Check
DNS	Domain Name System
GFC	Generic Flow Control
HEC	Header error control
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISO/OSI	International Standards Organization / Open System Interconnection
LTE	Long-Term Evolution
MPLS	Multiprotocol Label Switching
NIC	Network Interface Card
NNI	Network-to-Network Interface
PNNI	Private Network-to-Network Interface
POP3	Post Office Protocol version 3
PT	Payload Type

SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
UBR	Unspecified Bit Rate
UDP	User Datagram Protocol
UNI	User-Network Interface
VBR	Variable Bit Rate
VCI	Virtual Channel Identifier
VPI	Virtual Path Identifier
WiFi	Wireless Fidelity
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6

LIST OF APPENDICES

A Content of included CD **90**

A CONTENT OF INCLUDED CD

There are these files on included CD:

- MT_Bures.pdf - Electronic version of this thesis
- Scenario_ARQ.pdf - First scenario including introduction, comments and code
- Scenario_Switching.pdf - Second scenario including introduction, comments and code