UNIVERSITY OF LATVIA

FACULTY OF COMPUTING

Leo Truksans

NETWORK VIRTUALIZATION BASED ON EFFECTIVE PACKET TRANSFORMATIONS

Doctoral Thesis

Area: Computer Science

Sub-Area: Data processing systems and computer networks

Scientific advisor:

Dr.sc.comp., prof.

GUNTIS BARZDINS

Riga, 2014

Scientific advisor:

Dr.sc.comp., prof. Guntis Barzdins
University of Latvia

Referees:

____

____

____

The defence of the thesis will take place in an open session of the Council of Promotion in Computer Science of the University of Latvia

____

____

The thesis and its summary are available at ____

Head of the Council Janis Barzdins

Abstract

Two original technologies are proposed in this thesis that improve network virtualization effectiveness: ZERO tunneling protocol, and Packet Transformation Language (PTL) that can formally describe ZERO and other tunneling protocols in a compact language.

ZERO is the proposed Ethernet over IP tunneling protocol, which divides all Ethernet frames to be tunneled into NICE and UGLY frames. The UGLY frames are tunneled by traditional methods, such as UDP or GRE encapsulation, resulting in substantial overhead due to additional headers and fragmentation typically required to transport long Ethernet frames over IP network traditionally limited to MTU=1500 bytes. Meanwhile the NICE Ethernet frames are tunneled without any overhead as plain IPv4 packets due to non-traditional reuse of “fragment offset” field in the IP header. It is shown that for typical Internet traffic transported over Ethernet, the proposed ZERO tunneling protocol classifies 99.94% of Ethernet frames as NICE and thus results in nearly zero-overhead, no fragmenting Ethernet over IP tunneling. The proposed tunneling method extends also to Ethernet frames containing VLAN and MPLS tags, as well as IPv6 packets – all of these also can be classified as NICE and transported with zero-overhead over Internet or private IPv4 transport network. Unprecedented efficiency of the proposed tunneling protocol enables wide use of L2 transparency across existing L3 infrastructures thus leading to new network design patterns essential for modern applications such as Internet of things or cloud infrastructures.

Real world tests of a Linux kernel ZERO protocol implementation proved practicality of the new protocol and also uncovered a new observation – even the rare channel synchronization packets get fragmented only at 2-6% rate on burst traffic like Web browsing.

Based on the effective tunneling technology, the architecture of next generation Scientific Cloud and real time stream processing of astronomical data systems are proposed. The ZERO protocol is proposed as effective tunneling solution for joining federated IaaS clouds.

Keywords: computer networks, Internet, tunneling, Ethernet, stream processing, cloud computing

Acknowledgements

The author expresses many thanks to the supervisor of this thesis professor Guntis Barzdins for help in search of innovative ideas and for help in preparing publications that are the basis of this theses.

Also, the author expresses many thanks to the director of Institute of Mathematics and Computer Science of the University of Latvia (IMCS UL) Rihards Balodis-Boluzs, the executive director Inara Opmane, and other staff for the possibility to engage in academic groups and projects and use the Institutes' IT infrastructure during work on this theses.

All the people that supported author's work and path through the formal steps associated with this work are appreciated, as well. Most notably: Anita Ermusa and Ella Arsa and other staff of Computer Science faculty of University of Latvia (UL CSF) bachelor and major secretariats, UL Academic department, Elmira Zarina and other staff of UL Development and planning department.

Kind regards are expressed to the authors' colleagues at the UL CSF and UL Linux Center, most notably – Inga Medvedis, Rihards Rumnieks, Kaspars Krampis, Martins Balodis. Frequent discussions and exchanges of ideas and knowledge have often inspired the author for new goals.

The research in this theses is partially supported by:

- the European Regional Development Fund within the project Nr. 2010/0325/2DP/2.1.1.1.0/10/APIA/VIAA/109 and by the Latvian National Research Program Nr. 2 „Development of Innovative Multifunctional Materials, Signal Processing and Information Technologies for Competitive Science Intensive Products" within the project Nr. 5 „New Information Technologies Based on Ontologies and Model Transformations".
- European Union via European Regional Development Fund Project No.2010/0206/2DP/2.1.1.2.0/10/APIA/VIAA/011

Contents

List of figures

List of tables

Glossary

C-fields	PDU fields that are copied to the transformed packet without modification
Channel	An established transformation associated with a unique S-field list
CT	Channel Table: lists the transformations of Channels established in a Tunnel
DC	Direct Channel: a Channel in a Direct Tunnel.
DT	Direct Tunnel: a tunnel that is directed towards branch networks, optimal for "many-to-few" communications
Entrance	A tunnel entrance entity (device or software) that transforms a frame from the Transported network into a packet usable in the Transport network
Exit	A tunnel exit entity that transforms a packet used in the Transport network back into the frame used in the Transported network
Hederata	The encapsulation headers of IP packet – besides native Ethernet header may include VLAN tags, MPLS headers etc.
IC	Indirect Channel: a Channel in an Indirect Tunnel
IT	Indirect Tunnel: a tunnel that is directed towards central and global networks, optimal for "few-to-many" communications
MEPS	Maximum Encapsulated Packet Size: maximum NICE packet size that can be encapsulated in SYN and still produce a packet no larger than MTU of the transport interface
NICE	A frame that fulfills all criteria set in Definition 1 and thus is eligible for zero-overhead tunneling
Pseudowire	A logical networking channel that emulates properties of a physical wire

	connecting two nodes disregarding the actual physical topology
S-fields	Also “Saved fields”: Layer 2-3 information in the tunneled frame that needs to be preserved for accurate recreation on tunnel Exit
T-fields	Packet fields that must incorporate values necessary for tunneling the packet over transport network towards the Exit
Transport network	The network carrying the tunneled packets, e.g., Internet
Transported network	The network from which network traffic is to be transported into a tunnel, e.g., a corporate network
Tunnel	A simplex logical connection from one Entrance host to one Exit host that is used for ZERO protocol operation
UGLY	A frame that does not fulfill all criteria set in Definition 1 and thus is not eligible for zero-overhead tunneling
X-fields	Also “Index fields”: often unused fields in the packet's Layer 3-4 PDUs
ZERO ENC	IP packet with a UDP datagram that encapsulates the transported frame
ZERO IP	IP packet into which a NICE frame is transformed for zero-overhead forwarding in the Transported network
ZERO SYN	IP packet with a UDP datagram that encapsulates the transported frame and has one extra field for index value that allows to syncronise the S-fields of encapsulated frame to the tunnel Exit
ZERO server	A ZERO protocol prototype implementation in a userland process

1 General description of the theses

The presented theses has been worked on from 2008 to 2014 in the Institute of Computer Science and Mathematics of the University of Latvia (IMCS UL), and the Computer Science faculty at the University of Latvia (CSF UL). The thesis elaborates further the early prototype of ZERO protocol proposed by IMCS UL professor Guntis Barzdins and extends the architecture and technologies of the existing Scientific Cloud system designed and implemented at IMCS UL by the author and colleagues.

1.1 Relevance of the thesis

With advent of concepts of modern cloud computing and Internet of things the need to stretch Ethernet local area networks (LANs) outside boundaries of physical premises has greatly increased. Existing solutions try to satisfy this need in two ways: providing wide area coverage of the existing Ethernet protocol as a service; or tunneling Ethernet frames encapsulated in higher layer protocol data units (PDUs). Either approach is a form of network virtualization that separates the logical use of network channel from the physical one allowing to build a hierarchy of logical channels disregarding the physical infrastructure.

The architecture of modern IaaS cloud services heavily relies on the network virtualization. Multiple new network virtualization protocols have emerged in just few recent years [1][2][3]. All of these and the earlier known ones [4][5][6][7] solve the problem in the traditional encapsulation way. This theses shows a novel Ethernet-over-IP tunneling protocol called ZERO that tunnels Ethernet through the public Internet without the encapsulation overhead.

The ZERO protocol is new and probably it's key technique of adding redundancy with indexing will be debated. The choice of design details and parameters may be revisited and experimented upon. For easier design, development and implementation of the ZERO and similar protocols a new Packet Transformation Language (PTL) and its execution engine is proposed in this theses.

The ZERO protocol and the PTL expands the possibilities of effective network virtualization and serves as a basis for new effective wide area applications. Two such applications are described in this theses: A cloud architecture that effectively interconnects multiple clouds in a cloud exchange; and a system for effective streaming of astronomical data between remote astronomical sites and data centers.

1.2 The aim and tasks of the research

The main aim of the thesis is:

- To propose new technologies that would improve effectiveness of data flows in network virtualization applications across heterogenous infrastructures like tunneling or virtual networks in cloud computing installations and WANs.

To reach for that aim the following goals have been set in this thesis:

- To propose a language that allows to describe transformations imposed upon data packets in tunneling protocols.
- To propose and implement a prototype of new Ethernet-over-IP tunneling protocol that would not fragment packets on the Internet as transport network.
- To propose the architecture and services of the next generation Scientific Cloud at IMCS UL.
- To propose a system for effective real time streaming and processing of astronomical data between remote astronomical sites and data centers.

Tasks for reaching these goals:

- To study existing network tunneling protocols, paying special attention to the tunneling overhead.
- To propose the concept of a new Packet Transformation Language (PTL) that allows to describe transformations imposed upon data packets in tunneling protocols. It would allow to formally describe the transformations imposed upon data packets in existing tunneling protocols as well as in the new tunneling protocol.
- To describe the syntax of PTL.
- To describe the functionality of PTL engine.
- To propose the initial set of PTL functions.
- To propose the theoretical foundation of new Ethernet-over-IP tunneling protocol.
- To implement a prototype of the new Ethernet-over-IP tunneling protocol, consider building a stable high performance kernel mode prototype.
- To test the prototype/-s for conformity to the proposed concept and to observe the zero-

overhead feature in practice.

- To propose the concept of system for effective real time streaming and processing of astronomical data between remote astronomical sites and data centers, keeping in mind the possible uniqueness of the raw data.
- To propose the architecture, functionality and possible implementation of the stream processing system.
- To describe the file transfer protocol performance study done by author and colleagues that gives recommendations on protocol choice and tuning for file transfer over WANs.

1.3 Research methods used

Standards and case study research in the field of computer networks virtualization, performance and cloud computing has been used to widen the theoretical bases of this thesis. Logical reasoning has helped to elaborate on the proposed ideas.

Modeling method is widely used throughout the thesis to describe existing design concepts as well as to propose new ones. Also, several proposed procedures and algorithms have been visualized in form of block diagrams or pseudo-code.

It must be noted that all the networking protocols that are at the bases of Internet are published as open standards. It allows high quality research of the networking processes, interfaces and structures used in the Internet. Also, the effect of the proposed innovations often could be precisely calculated and modeled.

Computer simulation has been used to do the initial evaluation of ZERO protocol effectiveness.

Actual ZERO protocol prototypes have helped to observe in practice and empirically prove several statements about its functionality and effect on network traffic proposed in this thesis.

Study of the open source Linux operating system kernel source code and four other networking protocols was done to improve understanding of the networking processes happening in this modern network operating system and to find an arguably best model for incorporating ZERO protocol into the Linux kernel. Details of this particular research are left outside this thesis.

Authors experience in designing, building and maintaining the first generation Scientific Cloud at IMCS UL since 2008 has helped to reason about and innovate on the next generation Scientific Cloud proposed in this thesis.

1.4 Main results of the thesis

- A new Packet Transformation Language (PTL) for network traffic evaluation and mangling procedures. The syntax of PTL, initial set of PTL functions, functionality of PTL engine and Reverse Transformation Generator (RTG) are described.
- Major redesign of the ZERO Ethernet-over-IP tunneling protocol (patent number LV 14784, 20.01.2014) based on PTL, with significant functionality improvements and practical use over Internet. The redesigned ZERO protocol is implemented as a high performance Linux kernel mode driver and is used practically in Internet. The effectiveness of ZERO protocol is empirically proved with measurements.
- The architecture and services of the next generation Scientific Cloud at IMCS UL is proposed.
- A system for effective real time streaming and processing of astronomical data between remote astronomical sites and data centers is proposed. The architecture, functionality and possible implementation of the system are also described in this thesis.
- File transfer protocol performance study that gives recommendations on protocol choice and tuning for file transfer over WANs. The results encourage federated and heterogenous cloud systems to be set up even on data centers distributed on global scale.

1.5 Validation of the results

The high performance kernel mode prototype implementation of the ZERO protocol has been developed and evaluated by the author. The practicality and effectiveness of ZERO protocol is empirically proved.

The baseline performance of the ZERO protocol implementation has been determined on dedicated computers as well as on IaaS virtual machines in the Scientific Cloud at IMCS UL. This cloud system has been particularly convenient for kernel mode development since the virtual machines can be easily cold rebooted and the kernel trace messages right before crashes can be observed and documented.

A new factor in behavior of the ZERO protocol has been proposed and observed in real life – the “small sync factor”. It showed that almost never (detected only once in a synthetic test) fragmentation happens for TCP sessions shorter than 11 seconds, which is rather significant considering that the mandatory SYN packets of the ZERO protocol are fully encapsulated.

Several ideas and key technologies of the proposed next generation Scientific Cloud at IMCS UL have already been implemented in the existing Scientific Cloud (real time monitoring technology) and the High Performance Computing as a Cloud installation (MPI delegation technology) also done by the author and colleagues for the Faculty of Physics and Mathematics of the University of Latvia (FPM UL).

Two studies on currently available open standards file transfer protocols for TCP/IP networks has been done by the author and colleagues for the European meteorological union (EUMETSAT). The studies provided the background experimental material for the selection of a file transfer architecture in the EUMETSAT upcoming next generation Meteorological weather satellite system to be launched in 2014.

1.6 Publications of the research results

The list of publications by the author includes 8 titles, of these 3 are included in the SCOPUS or Thomson Reuters ISI Web of Science international scientific databases.

- *ZERO: an efficient Ethernet-over-IP Tunneling Protocol*, L.Truksans, G.Barzdins, A.Haidens, I.Opmane, R.Balodis, 2013, Springer, "*Inter-cooperative Collective Intelligence: Techniques and Applications*" in the "*Studies in Computational Intelligence*" book series, Volume 495, p.349-374, ISBN 978-3-642-35015-3 (SCOPUS)
- *Unified computing facility design based on open source software*, R.Balodis, I.Opmane, L.Truksans, p.337- 342 in 2012 International Conference on Systems and Informatics (ICSAI 2012) 19.-20. May 2012, Yantai University, IEEE Catalog Number: CFP1273R-CDR, ISBN: 978-1-4673-0197-8, IEEExplore Digital Library DOI: 10.1109/ICSAI.2012.6223629 (SCOPUS)
- *Real time batch processing of streamed data using Lustre*, Rihards Balodis, Kaspars Krampis, Inara Opmane, Leo Truksans, Baltic Applied Astroinformatics and Space Data Processing (BAASP) conference, Engineering Research Institute „Ventspils International Radio Astronomy Centre" of the Ventspils University College, published in conference proceedings, ISBN 978-9984-648-20-0
- *Real time batch processing of streamed data using Lustre*, Rihards Balodis, Kaspars Krampis, Inara Opmane, Leo Truksans, Space Research Review, Volume 1, 2012, ISBN-978-9984-648-23-1

- *File Transfer Protocol Performance Study for EUMETSAT Meteorological Data Distribution*, Leo Truksans, Edgars Znots, Guntis Barzdins, 2011, Scientific Papers, University of Latvia, Volume 770, p.56-67.
- *ICT aspects in Latvian educational system*, J.Miķelsons, A.Andžāns, Ē.Ikaunieks, A.Niedra, U.Straujums, L.Trukšāns, V.Vēzis. Proceedings of LatSTE international conference 2005, Rīga, 2005, lpp. 55–60.
- *ICT in Latvian Educational System – LEIS Approach,* J.Miķelsons, A.Andžāns, J.Bicevskis, I.Medvedis, A.Niedra, U.Straujums, V.Vēzis, L.Trukšāns. Proceedings of EISTA'05, 2005, vol. 2, p. 94-98. (Thomson Reuters, ISI Web of Science)
- *Internet infrastructure in Latvian education informatization system (LEIS)*, I.Medvedis, U.Straujums, L.Trukšāns. 2 lpp., Sakaru pasaule, 4(20) – 2000.

1.7 Presentations in conferences

- *Lielu datu apstrādes risinājums zinātniskajā mākonī (Big Data processing solution in the Scientific Cloud)*, L.Trukšāns, Latvian Open Technologies Association (LATA) conference, 2014., Riga.
- *Unified computing facility design based on open source software*, R.Balodis, I.Opmane, L.Truksans, 2012 International Conference on Systems and Informatics (ICSAI 2012) 19.-20. May 2012, Yantai University.
- *Real time batch processing of streamed data using Lustre*, R.Balodis, K.Krampis, I.Opmane, L.Truksans, Baltic Applied Astroinformatics and Space Data Processing (BAASP) conference, 2012., Ventspils.
- *Zinātniskā datu glabātuve un atvērtās tehnoloģijas (The Scientific Data Storage and open technologies)*, R.Balodis, I.Opmane, L.Trukšāns, Latvian Open Technologies Association (LATA) conference, 2008., Riga.
- *ICT aspects in Latvian educational system*, J.Miķelsons, A.Andžāns, Ē.Ikaunieks, A.Niedra, U.Straujums, L.Trukšāns, V.Vēzis. LatSTE international conference 2005, Rīga.
- *ICT in Latvian Educational System – LEIS Approach*, J.Miķelsons, A.Andžāns, J.Bicevskis, I.Medvedis, A.Niedra, U.Straujums, V.Vēzis, L.Trukšāns. EISTA'05 conference, 2005.

2 Packet Transformation Language (PTL)

The language described here can be used in a scenario in which two devices in a network make an agreement that certain network protocol data units (PDUs) exchanged between them may be subjected to certain alterations that are defined on these two hosts. The alterations would transform a PDU into a temporary state to be sent to the other host which would restore the original PDU state. The host that transforms PDU into temporary one is called *Sender* and the host that restores the original PDU would be called *Receiver*. This scenario is widely known and used in computer networks as tunneling or network virtualization. A universal language for defining transformations happening in such scenarios is proposed here – Packet Transformation Language (PTL). The functionality of a PTL engine and Reverse Transformation Generator (RTG) are also proposed here.

To author's knowledge PTL language is original what is confirmed by the patent-search of the original ZERO protocol.

2.1 The concept of PTL

Senders use *forward transformations* (*FTs*). *Receivers* use *reverse transformations* (*RTs*). For simplicity a common term *peers* is used for both a *Sender* and a *Receiver*. Also, the term *agreement* denotes a shared understanding between a *Sender* and a *Receiver* about the transformations they may use. In practice *agreements* most likely will be a matter of configuration. Note that a *Sender*/*Receiver agreement* is conceptually a simplex one. It works in one way. For one *peer* to receive and also send transformed PDUs with another *peer* it will take two *agreements* – one in each direction.

PTL is designed with the following abilities:

1. to transform a given OSI layer PDU into a PDU of different OSI layer;
2. to provide a rather universal language to define PDU transformations;
3. to provide reversibility of PDU transformations.

For the 3rd ability to work it is a prerogative that any forward transformation (*FT*) in a *Sender* is matched by a functionally reversed transformation (*RT*) in *Receiver*. This ensures that any PDUs transformed by *Sender* can be reconstructed to the original content on its *Receiver*. This principle is called here and later as *Reversibility principle*. It can be described with the following formulas, where *P* is the original PDU and *P′* is the transformed PDU.

$$FT(P) = P' \quad \text{and} \quad RT(P') = P$$

$$\text{hence,} \quad RT(FT(P)) = P$$

One real world example of such transformation is the GRE tunneling protocol [9]. The *FT* of GRE would be to encapsulate a given packet *P* into a GRE packet *P′*. Then, the *RT* of GRE would be to decapsulate the *P′* into the original *P*.

In order to ensure the *Reversibility principle* the transformation definitions are applied by the master/slave principle, from here on meaning *Sender* is the master and *Receiver* is the slave:

- only *FT*s are defined and only on *Senders*;
- *FT*s are propagated from *Senders* to *Receivers* in form of service messages;
- On *Receivers*, *RT*s are derived from acquired *FT*s, while honoring *Reversibility principle*.

Any FT is defined as a chain of *functions*. A *function* can have a set of arguments. The arguments may point to regions in the current PDU upon which the *function* is going to have effect. Thus, any function is allowed to change the contents of PDU. In the process of going through chain of functions the PDU is transformed.

The *functions* in chain are executed sequentially from the beginning. All *functions* ether succeed or fail. And success of the whole transformation is determined in „execute until failure“ manner:

- If a *function* succeeds, pass on to next *function*;
- If a *function* fails, the whole transformation stops and fails;
- If no more *functions* are defined, the whole transformation succeeds.

Figure 1 illustrates the rule processing.

For transformation reversibility to work all *functions* need to be reversible. In the context of proposed concept this means that for every *function F* changing PDU from *P* to *P′* there needs to be a reverse *function R* for any possible *P* values, that:

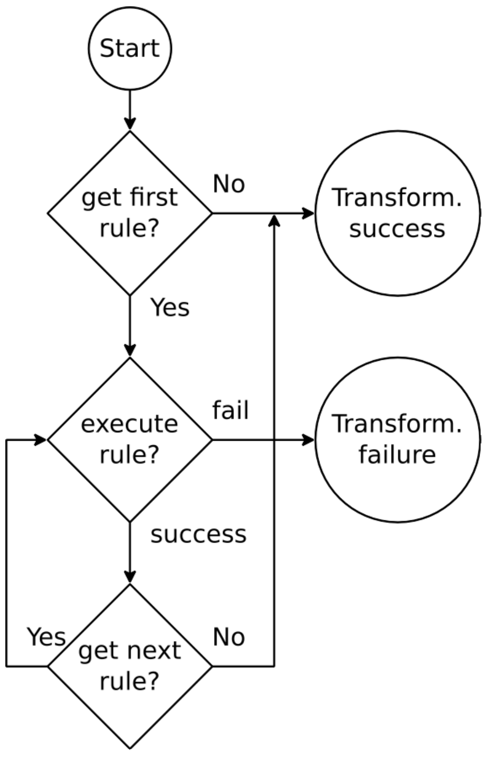

Figure 1: Rule processing diagram

- changes *P′* into exactly *P*;
- succeeds whenever the *F* has succeeded.

Such functions are called *reversible*.

Lemma 1: *A forward transformation is reversible if all its functions are reversible.*

Assumption: the transformation *functions* chain is ordered, it has determined and linear execution order (no loops, deviations or parallelisms).

Proof. Given the assumption it can be deduced that the *functions* of any forward transformation *FT* form a fixed chain in which any *function* F_i takes PDU content P_i and produces new content P_{i+1} that would be passed as content for next *function* F_{i+1}.

If any *function* of the *FT* is reversible, it means:

- any F_i has a reverse *function* R_i that would transform P_{i+1} into P_i. Thus, for any *Forward Transformation* $[T_1 ,..,T_n]$ that transforms P_1 into P_{n+1} a *Reverse Transformation* $[R_n,..R_1]$ can be found that transforms P_{n+1} into P_1.
- if the whole *FT* has completed successfully for a PDU, the *RT* will also successfully complete and transform back to the original PDU.

Lemma 1 is proved.

Figure 2 illustrates Lemma 1. It shows a fragment of forward transformation, matching fragment of the reverse transformation, and how the PDU content traces back in the reverse process.

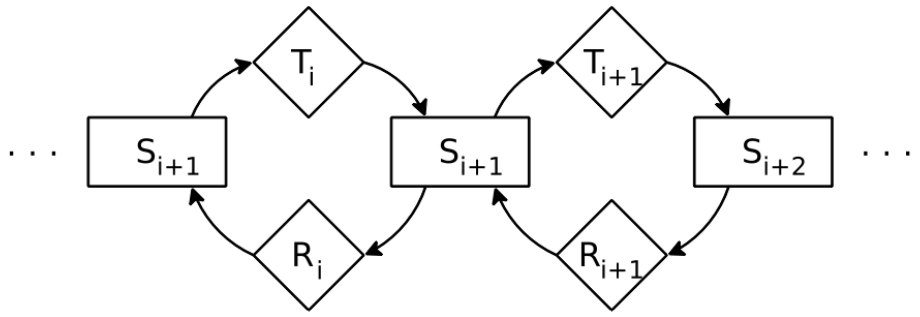

Figure 2: Forward and reverse functions changing packet contents

Finding an *RT* for any given *FT* may prove itself to be an interesting endeavor. The author omits this task for now and proposes a simple approach:

- a user can choose *functions* only from a given set of predefined *functions*;
- all the predefined *functions* are valid in the sense of Lemma 1, most notably: all *functions* are *reversible*.

The predefined *functions* later in this chapter are already provided with known reverse *functions*. Such a constrained approach serves another purpose – it will be easy to see that a user can not configure an *FT* that would not work in reverse.

2.2 The Reverse Transformation Generator

The PTL engine would have a Reverse Transformation Generator (RTG) that can construct an *RT* from a given *FT* with the following algorithm:

- it starts with an empty *RT* list [];
- it takes *functions* one by one from *FT* list [T_1,..,T_n], starting with the first (T_1);
- for any *function* T_i it finds its reverse *function* R_i, and appends R_i to *RT* list.

Given RTG always finds correct reverse *function*, it is easy to see RTG will construct *RT* list [R_1,..R_n] such that R_i (for any i from[1..n]) is a reverse transition of T_i. With Lemma 1 this means the *RT* constructed by RTG will be an exact reverse of the given *FT*. While *FT* will be executed forward (T_1,..,T_n) and produce the temporary PDU content P_{n+1}, the *RT* will be executed backward – [R_n,..R_1] producing the original PDU content P_1 from P_{n+1}. Thus, *Reversibility principle* is honored.

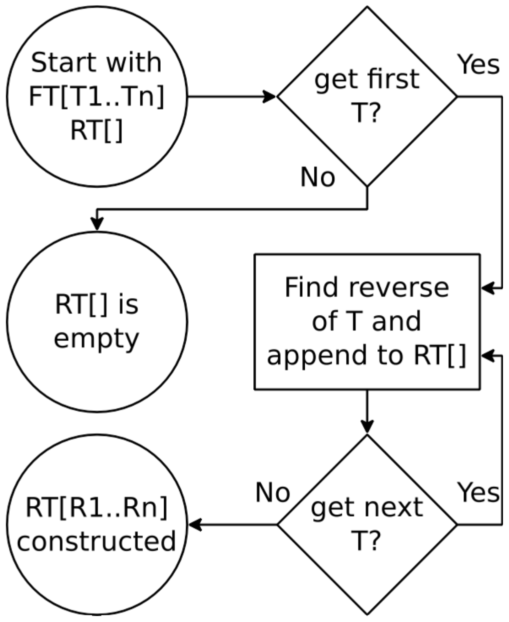

Figure 3: Reverse Transformation Generator algorithm

2.3 Practical aspects of PTL

The practical need to know the success of the whole transformation would be to decide what to do further with the PDU. For instance, whether to drop the PDU or pass to the next stage of networking work flow. And if pass, then which one: the original PDU (perhaps, if unsuccessful transformation) or the transformed one (perhaps, if successful transformation).

One might notice similarity between the proposed concept and the one of existing packet filtering and mangling solutions, [8] for instance. However, several differences are present:

- The rules are all executed until one fails, which is opposed to logic of traditional stateless packet inspection systems [8], [10], [11].
- Possibility to easily define custom transformations from a range of functions seldom found together in one solution.
- Perhaps, most distinguishably – the automatic reversibility. The authors research into the existing packet filtering and mangling solutions doesn't show an existence of such a system that would generate reverse transformations from user defined forward transformations of network PDUs.

PTL imposes a requirement on *peers*: any *Sender* must have means to propagate service information to the agreed *Receiver*. The details of such service protocol or technique are outside the PTL concept. Many networking solutions already have some service protocols that might incorporate PTL service messages.

A word of caution must be mentioned. With PTL user can create transformations that compromise integrity of a user PDU or its chances of reaching intended destination. At least for now PTL is not meant to be an end user tool, rather to serve as a platform for feature rich, perhaps, intelligent traffic channeling solutions that would integrate new transitions and pay attention to correctness of their actions.

2.4 PTL syntax

As mentioned earlier, a PTL transformation is a list of *functions*.

```
<Transformation> := <function 1>
                    ...
                    <function n>
```

A *function* definition consists of *function* name and optional list of parameters. A syntax similar to C programming language is used in function calls.

<function> := <function_name>([<parameter_1>...])

A parameter may itself be a braced list of elements separated by comma. Elements can be: bits, regions of PDU content or constants. Besides numbers and strings, two more forms of constants exist for convenience: MAC addresses (in hexadecimal form) and IP addresses (in dotted decimal form).

<parameter> := <element>|([<element>...])

<element> := <bit>|<region>|<constant>

<constant> := <number>|<string>|<MAC_address>|<IP_address>

A region is described with a single or a pair of bit offsets in the outer layer of the original PDU: starting bit, and ending bit. The pair is put into brackets.

<region> := <start_bracket><bit>–<bit><end_bracket>

Some PDUs may contain variably sized data structures. Because of that the absolute bit offset consists of two parts: a prefix tells in which level PDU to seek, and the relative offset there. This allows to correctly calculate absolute offsets for any variably sized PDU.

<bit> := <layer_prefix><layer_offset>

The PTL engine is mandated to look for actual offsets of the prefixes in the original PDU before transformation. Three prefixes are proposed:

- *„L“* – Link layer header start;
- *„N“* – Network layer header start; if not found – equal to frame header end + 1

- *„T“* – Transport layer header start; if not found – equal to packet header end + 1

The <*layer_offset*> can be a number or a reserved symbol “E” that means the last correct offset of that PDU. Choice of symbol for any bracket tells if the offset is inclusive or non-inclusive.

<*start_bracket*> *:= “[” | “(”*

<*end_bracket*> *:= “]” | “)”*

The bracket symbols mean:

- *“[”* and *“]”* means inclusive offset;
- *“(”* and *“)”* means non-inclusive offset.

The brackets may be mixed.

Some region examples:

- [N0-T0) – bits from network layer header start inclusive up to transport start non-inclusive, meaning – the whole packet header.
- ([P128-P159],[T0-TE]) – Destination IP address and whole transport layer PDU.
- [F96-P0) – region right after Ethernet frame *SrcMAC* up to end of frame header (not including layer 3 PDU start) – the variably sized region that holds Ethertype and may hold VLAN tags.
- [P0-F0] – logically impossible, thus, empty region.

2.5 Functions

An initial set of *functions* that allow a PTL system to mimic some existing network tunneling or labeling solutions is proposed here. PTL framework can be used in wider range of scenarios. More *functions* can be defined and implemented for particular needs.

Note:

- all *functions* fail if the number of parameters is wrong;
- all *functions* by default are successful, until noted otherwise;
- if any *function* mangles content, then any compromised checksums will get recalculated after the transformation.

2.5.1 Evaluation functions

Name:

`equal` – compares two parameters.

Syntax:

```
equal(p1, p2)
```

Description:

Useful for restricting to certain content in a PDU.

Impact on PDU:

None.

Outcome:

Fails if:

- parameters are of different size;
- parameters are not equal.

Reverse function:

None.

Example:

To check if Ethertype field of an untagged frame is 0x0800 (meaning it contains an IPv4 packet):

```
equal([F96-F112], 0x0800)
```

Name:

`exists` – checks existence of the parameter.

Syntax:

```
exists(p1)
```

Description:

Useful for checking presence of optional or variably sized fields.

Impact on PDU:

None.

Outcome:

Fails if:

- at least one of the regions can not be found.

Reverse function:

None.

Example:

To check if IPv4 packet has options:

```
exists([F160-T0))
```

2.5.2 Mangling functions

Name:

`swap` – swaps content of both parameters.

Syntax:

```
swap(p1, p2)
```

Description:

Useful for exchanging contents of fields.

Impact on PDU:

p1 ↔ p2

Outcome:

Fails if:

- any parameter contains constants (can not guarantee *reversibility*);
- parameters are of different size.

Reverse function:

Symmetric.

Example:

To swap UDP ports:

```
swap([T0-T15], [T16-T31])
```

Name:

`insert` – inserts contents of p2 before p1.

Syntax:

```
insert(p1, p2)
```

Description:

Useful for inserting labels or tags.

Impact on PDU:

The regions starting with p1 are offset by length of p2. Inserts the p2 at p1. Adjusts length of all touched PDUs.

Outcome:

Fails if:

- p1 can not be found.

Reverse function:

Cuts out the inserted region. Adjusts length of all touched PDUs.

Example:

To insert a VLAN 10 tag into an Ethernet frame:

```
insert(F96, (0x8100, 0x000A))
```

Name:

`encapsulate` – encapsulates p1 into new p2.

Syntax:

```
encapsulate(p1, p2)
```

Description:

Creates an “envelope” PDU whose type and content is defined in p2 and inserts its header before PDU pointed to by p1. Useful for encapsulating current PDU into another PDU.

Impact on PDU:

Encapsulates content. Sets length of the new PDU. Adjusts length of PDUs outside p1, if any.

Outcome:

Fails if:

- p1 does not point to a start of existing PDU;
- p2 does not describe a correct PDU.

Reverse function:

Removes the envelope. Adjusts length of all outer PDUs.

Example:

Encapsulate current IPv4 packet into another IPv4 packet with the given Source and Destination IP addresses and IP protocol number (4 – IP-in-IP); adjust outer Ethernet:

```
encapsulate(P0, (“ipv4”, 10.0.1.2, 10.0.2.2, 4))
```

Name:

`index` – maps contents of sr into dr.

Syntax:

```
index(dr, sr, db, dbs)
```

Description:

Maps contents of source range (sr) into an index; saves the mapping into a Mapping DataBase (MDB) identified by db where dbs number of mappings can be stored; saves the index into destination range (dr). The mapping is saved together with exact absolute pattern offsets for safe reversal.

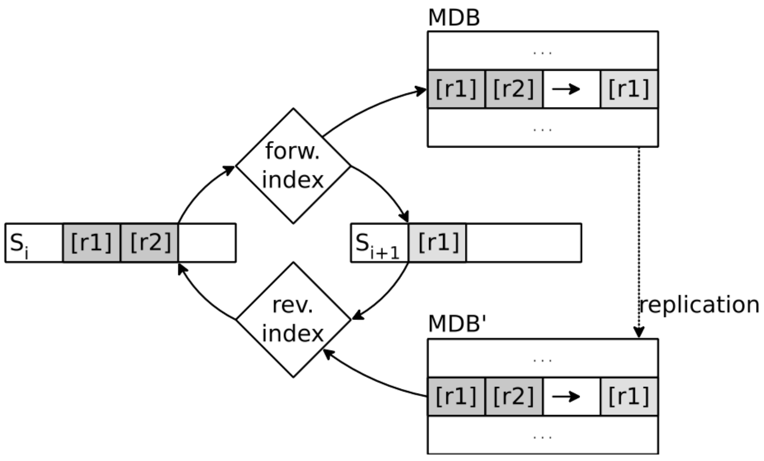

Figure 4: forward and reverse index function

Note that a transformation may utilize multiple MDBs. For sure reversibility dr regions must also be included in the sr regions. Thus a larger region sr can "collapse" into a sub-region dr in form of index, thus, leaving other sr sub-regions redundant or unnecessary.

Impact on PDU:

The "collapsing" allows to leave the unnecessary sub-regions behind and further "travel lighter". This idea in one form is known as header compression [rfc1144]. In PTL this transition can be used on any regions, in multiple settings, with multiple MDBs, even recursively.

Outcome:

Fails if:

- any region of dr is missing or partial;
- any region of dr is different length or positions than saved in MDB.

Reverse function:

Using index value from dr, finds contents of the full sr regions from the replicated MDB.

Restores the original sr. The reverse function can fail if the MDB replica is not complete yet and the index is not found or is corrupt.

Example:

Map IPv4 Source and Destination IP addresses into the Source IP (the Destination IP becomes eligible for new content), using MDB number 1 that can store 1024 entries:

```
index([P96-P127],[P96-P159],1,1024)
```

Name:

`DES` – encrypts p1 with key.

Syntax:

```
DES(p1, key)
```

Description:

Uses DES encryption with given key to encrypt p1 regions.

Impact on PDU:

Replaces p1 regions with encrypted content.

Outcome:

Fails if:

- key is not a valid DES key.

Reverse function:

Uses DES decryption with the key.

Example:

Encrypt the data field of IPv4 packet:

```
DES([P160-PE], “3b3898371520f75e”)
```

Name:

`flag` – marks p1 with info from p2.

Syntax:

```
flag(p1, c1, c2)
```

Description:

Checks if p1 regions contain c1. Then replaces p1 with constants from c2. Useful for setting flags in PDUs while ensuring the flags have not been set already.

Impact on PDU:

Sets p1 regions to values of c2.

Outcome:

Fails if:

- p1, c1 and c2 don't match in size;
- p1 contains constants;
- c1 or c2 contains regions;
- p1 contains values not equal to c1.

Reverse function:

Reverses to **`flag`**`(p1, c2, c1)`. The reverse can fail but only if p1 has been corrupt after this function and is not equal to c2 any more.

Example:

Check and mark the Evil bit in IPv4 packet:

```
flag(P48, 0, 1)
```

2.6 A PTL transformation example

The following transformation first checks if:

- the layer 3 PDU is IPv4 packet with standard size header;
- the packet fields MF and FO are all equal to 0;

… then does the following mangling:

- checks and marks the Evil bit in IPv4 packet;
- maps IPv4 Source and Destination IP addresses into the Source IP, using MDB number 1 that can store 1024 entries.

```
equal(([P0-P7],[P50-P63]), (0x45, 0))
flag(P48, 0, 1)
index([P51-P63], ([P51-P63], [P96-P159]), 1, 1024)
```

2.7 Conclusions on PTL

Packet Transformation Language (PTL) is a universal language for defining reversible transformations of protocol data units (PDUs) for tunneling and network virtualization scenarios. PTL concept, theoretical groundwork, syntax and a set of initial functions has been described in this chapter.

The PTL engine is proposed in this chapter. It processes *forward transformations* (*FTs*) on egress PDUs and *reverse transformations* (*RTs*) on ingress PDUs, thus restoring the original content of PDUs that were before the *FTs*. The *reversibility principle* is proposed and reasoned about. A lemma is proved: *A forward transformation is reversible if all its functions are reversible.* Upon this a Reverse Transformation Generator (RTG) is proposed that can construct an *RT* from any given *FT*, given that all functions in the *FT* have known reverse functions.

PTL is primarily meant as a tool to describe PDU transformations happening in tunneling protocols and to simulate new protocols on existing infrastructures. Still, this language stands out from existing PDU transformation designs like the ones found in tunneling or network address translation (NAT) for several key principles:

1. The transformations can fail. This means a framework of transformations may be used that defines “primary” and “fall-back” transformations for intelligent traffic engineering.
2. The rules are all executed in sequence until one fails, which is opposed to logic of traditional stateless packet inspection systems that execute only the matching rule and then quit

inspecting the reminder of policy.

3. Possibility to easily define custom transformations from a range of functions seldom found together in one protocol.
4. The *reverse transformations* are generated automatically by RTG.

This chapter describes completely individual work of the author.

3 ZERO: an efficient Ethernet-over-IP Tunneling Protocol

3.1 Introduction

Internet of things might fail to take off in near future for low-level technical constraints, such as IP addressing and routing inflexibility of current Internet protocols.

Therefore a Layer 2 (L2) tunneling method is proposed here, which in terms of robustness is comparable to NAT (which is a staple of current IPv4 Internet overcoming the addressing limitations of the early Internet), yet provides a remedy to a different current Internet limitation. The problem addressed by the proposed approach is that current Internet is a Layer 3 (L3) network, while many "private network" or “partially private network” applications (such as VPN, tunneling, distributed server farms, cloud computing, high throughput computing, Internet of things, etc., as illustrated in chapters 4, 5 and 6) would benefit, if Internet would have been able to provide robust and efficient L2 connectivity transparent to dynamic routing, auto discovery and multicast within overlaid “private” L2 clouds. By efficient the author means nearly non-fragmenting tunneling.

This problem has partially been addressed by MPLS technology widely deployed in carrier networks and providing Ethernet-over-MPLS Layer 2 VPN service [12][13] in combination with VLAN [14] technology. But due to its cost, complexity, and reliance on extended MTU Carrier Ethernet [15], MPLS is not practical for use in existing access networks.

The current solutions for tunneling L2 traffic over access networks and Internet are highly inefficient, because in most cases they relay on extensive encapsulation, packet fragmentation, and reassembly – take OpenVPN [7] as example. This might be justified in cases where encryption of VPN traffic is additionally applied, but for mere transport of L2 traffic over L3 Internet an original and highly efficient tunneling method with nearly zero overhead is proposed here.

Three categories of popular PDU processing protocols and methods that have some of the features of interest in the proposed context have been identified:

1. The tunneling protocols – encapsulate one PDU inside another of the same or different OSI layer.
2. Translation technologies – mangle the transported PDUs, do not change or append their data structures.

3. Traffic compression protocols – compress some of the PDU fields and achieve more compact traffic flows.

The first two categories are presented in Tables 1 and 2. The third category is not investigated deeper in this work since those techniques perform rather complex data processing on each packet with unpredictable timing – dependent on a packet content.

Table 1: Popular tunneling protocols

Protocol	**Layers**	**Overhead**	**Short description and features of interest**
VLAN (802.1q)	L2-over-L2	4B	Tags Ethernet frames with a new field. Does not distinguish content differences and does not optimize overhead. Needs switched Ethernet transport network with larger (1522B+) frame support. The access/trunk port configurations are predefined as configuration.
Ethernet-over-MPLS	L2-over-L2	4B	Needs specific transport network – MPLS, user owned or leased. Point-to-point topology.
VPLS	L2-over-L2	8B	Switches for learned MAC addresses, like Ethernet bridging. Needs specific transport network – MPLS, user owned or leased. Broadcast, Multi Access topology.
IP-within-IP (RFC 2003)	L3-over-L3	34B (20+14)	Works through Internet. Encapsulates into IP packets with own protocol number (4). Has no additional fields.
EtherIP (RFC 3378)	L2-over-L3	36B (22+14)	Works through Internet. Tunnels VLAN tagged frames, as well. Can also tunnel different L3 protocols, since it encapsulates without checking frame contents. Able to operate with single interface to local network. Encapsulates into IP packets with protocol number 97. Additional 16 bit field immediately after IP header is partially used.
L2TP (RFC 2661)	L2-over-L4	42B (28+14)	L2TPv3 supports pseudo-wire concept. May have multiple sessions (for each protocol) in one tunnel. Encapsulates into UDP packets. Exchanges specific in-band control messages. Complex protocol with multiple states and events.

Protocol	Layers	Overhead	Short description and features of interest
VXLAN	L2-over-L4	50B (28+8+14)	Encapsulates into UDP packets. Establishes discrete “segments”. Encapsulated VLAN numbers and MAC addresses may be reused for every segment. Unicast is addressed directly. Broadcast frames are addressed to segment's multicast group.
NVGRE	L2-over-L4	50B (20+8+14)	Encapsulates into GRE protocol and adds a field for tenant number.
STT	L2-over-L4	54B (20+20+14)	Encapsulates into TCP segments, but does not support TCP session semantics. That allows to use hardware Segmentation Offload, even depends on it by intentionally sending and receiving larger segments that only once in the beginning has STT protocol fields. Utilizes SEQ and ACK fields of the encapsulation segment for protocol specific data. Uses 64b “Context ID”.
OpenVPN	L2-over-L4	42B (28+14)	Works through Internet. Encapsulates into UDP packets.

Table 2: Popular PDU transformation techniques

Protocol	Layers	Overhead	Short description and features of interest
NAT, PAT	L3	0	Modifies address or ports fields. Initial packet header content is lost.
RAT	L3	0	Establishes a tunnel with NAT features between mobile device and special gateway in home network. The tunnel is 1-to-N. Only part of the traffic between two nodes goes through the established tunnel – that is initiated from CN to MN. MN sends data to CN directly using mobile IP addresses.
CARP	L3	0	Identified traffic states are synchronized between the configured firewalls. If one firewall fails another can

			take over the packet filtering task using current state table. The table is used for direct and reverse traffic inspection as mandated by the stateful filtering principle. A new state is noted for every new session of most popular protocols: TCP, UDP, PING, etc.

The research shows that none of the existing protocols satisfy all the following criteria: Ethernet over IP tunneling protocol; works through the public Internet; non-fragmenting. All popular Ethernet-over-IP protocols such as OpenVPN, L2TP [6] are encapsulating tunnelled Layer 2 frames and thus require fragmentation.

This chapter describes the proposition – the ZERO tunneling protocol that is simplex, in most cases does not fragment Ethernet frames carrying even maximum size IP packets, works over public Internet, and its synchronization is resilient to packet loss.

3.2 The ZERO protocol concept and design

Observing a typical Internet user traffic one may notice a set of fields that usually have identical content in a burst of Ethernet frames. Also, some fields are rarely used at all. Consider, a computer is using SMTP protocol to send a large e-mail to a remote server. Every outbound frame during the whole session will have the same source and destination MAC addresses. Also, the *Fragment Offset* (FO) field will probably be 0. The idea behind ZERO protocol is to distinguish the often unused fields in the packet's Layer 3-4 PDUs (here and later called U-fields as in “Unused fields”) and fill those with the Layer 2-3 information that needs to be preserved during tunneling (here and later called S-fields as in “Saved fields”) over the Internet. Essentially, the same packet gets forwarded on the tunnel entrance, just some fields are substituted. On the other end of the tunnel the contents of the substituted fields are restored and the destination computer receives the same frame that was sent from the source computer.

For the rest of this chapter term *Entrance* will be used for the tunnel entrance entity (device or software) and term *Exit* for the tunnel exit entity. The simplex nature of this protocol must be emphasized and with *Entrance* and *Exit* only one direction is really meant. The reverse direction (if necessary) is identical in functionality but fully separate data structure, really – a different tunnel.

And since ZERO is an Ethernet-over-IP tunneling protocol it is about transforming Ethernet frames into IP packets on *Entrance* and IP packets to Ethernet frames on the *Exit*.

Here and later the term *Transported* network is used for the network to be transported into the tunnel. An example would be a company network. And the network that is used to carry the tunneled traffic is *Transport* network. Internet is an example of *Transport* network. Figure 5. shows the terms associated with a simplex tunnel (in one direction).

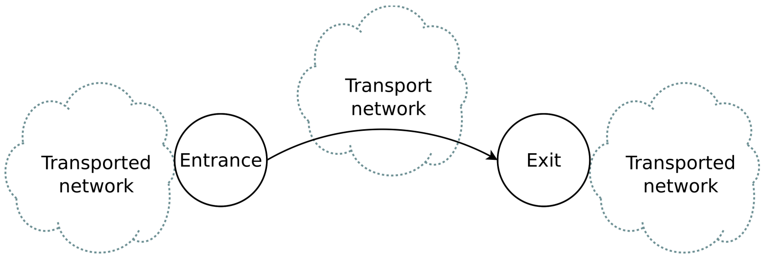

Figure 5: Tunneling terms

The author proposes the *Fragment Offset* (FO, 13 bits) field of IP header to be actually used seldom enough that it can be used as a U-field. Choosing the U-field set is discussed later in this chapter.

A kind of labeling mechanism is necessary to mark the ZERO IP packets on the *Transport* network so the *Exit* will recognize those from the other traffic. The *evil bit* (EB, 1 bit) IP field is unused in the public Internet as described in [18]. The author proposes to use it as a flag to mark the transformed packets. In case EB will be used for other purposes other ZERO labeling mechanisms may be employed. Looking for a combination of null Identification field and non-null FO field may be another way to distinguish ZERO IP packets.

Presumably, no Ethernet or IP fields are universally unused. Any may contain a non-null value in some scenarios.

Definition 1. *Ethernet frame is said to be NICE if it fulfills the following criteria:*

- *it includes a legal IPv4 packet as the last part of Ethernet data field;*
- *the header of the included IPv4 packet is 20 bytes long (has no options field such as Source Routing);*
- *the TTL value of the included IPv4 packet is higher than certain minimum (ttl_min,*

described further);

- *the U-fields of the included IPv4 packet contain null data.*

All these criteria are typically met for Internet traffic sent over Ethernet (natively or in VLAN or in MPLS). NICE frames can be tunneled with zero-overhead.

Definition 2. *Ethernet frame is said to be UGLY if it is not NICE.*

UGLY frames will be tunneled in less optimal, traditional full encapsulation.

Upon entering tunnel the destination IP address of a NICE frame will change to the other end of the tunnel to be routed through Internet. Hence, destination IP address (32 bits) is an S-field. All other IP header fields may go into *Transport* infrastructure unmodified and will remain so during the transportation. The TTL field is an exception and is discussed further.

Leaving unmodified Source IP address in some cases may be a policy issue. In that case, masquerading or other security feature may be used before the tunnel *Entrance*.

Also, whole frame header is an S-field since it has meaning only to local link and will inevitably be lost in transit through Internet. It must be noted here that Ethernet frame header is not fixed in length. Besides Source and Destination MAC addresses and Type fields other fields of significance to the switching equipment may be added to the header. Examples are VLAN tags and MPLS labels. Those even may add up to create a hierarchy of labels. Those all are S-fields, too. And it is presumed the whole frame header will be fixed for any combination of two hosts. If the labeling between two hosts will change at some point in time the protocol would treat the new header as another data set. The frame Preamble field is left outside S-fields because it is a constant label for marking frame start, and also the Checksum field which will be regenerated on every link anyway.

To sum up, two elements of the S-field set are identified:

- Packet destination IP field (4 bytes);
- Whole frame header up to frame data field start (14+ bytes).

3.2.1 NICE tunneling

The transformation of the NICE on *Entrance* is performed by substituting the S-fields for U-fields, setting the destination IP address to that of the *Exit* host, and setting EB flag. Here and later the transformed packet is called ZERO IP. On the tunnel *Exit* the S-fields are restored into a recreated frame and the U-fields and EB flag are blanked.

Here and later an established transformation associated with a unique S-field set is called *Channel*. Thus for multiple S-field sets a tunnel will contain multiple *Channels*.

Replacing a destination IP address in an otherwise unmodified packet will lose it's integrity in regards to its header checksum. So, during both transformations the packet also gets new header checksum. The frame checksum will be recalculated during reconstruction on the *Exit*.

The proposed S-field set (DstIP, Frame header) is at least 144 bits large, hence larger than the proposed U-field set of only one 13 bit field (FO). A *Channel Table* is used here to list the transformations of *Channels*. It has 2 columns: *s_fields* (S-field values); *index*. The *index* is the same size as the U-fields. During *Entrance* transformation the associated *index* is written into the U-fields to represent an S-field set. See Figure 6. The reserved width of the *s_fields* field is not specified here, that decision is left to implementation.

If the S-field values for another frame in *Entrance* are already present in the *Channel Table*, the *Channel* is reused. For a new set of S-field values the *Entrance* makes a new *Channel Table* entry with an unused *index*. The *Channel Table* synchronization is described further.

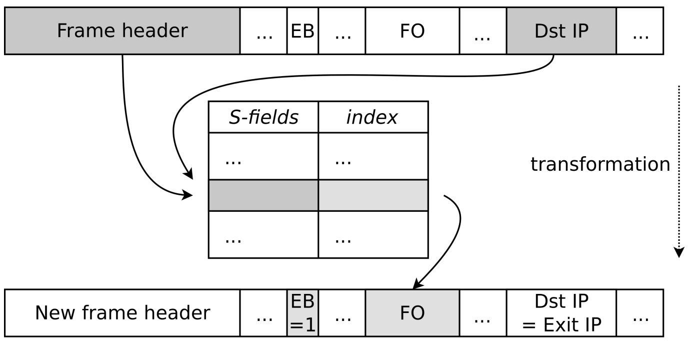

Figure 6: Transformation of NICE packets on *Entrance* using Channel Table

3.2.2 UGLY tunneling

The transformation of UGLY frame on *Entrance* is performed by encapsulating the whole frame into a special IP packet with a UDP datagram that here and later is called ZERO ENC. The format of ZERO ENC packet is shown in Figure 7. It is a rather usual UDP packet with UDP Data field carrying the whole original frame. ZERO ENC is addressed to the *Exit* IP address and to a dedicated UDP port. The port value is irrelevant at this point. It's an arbitrary number in the prototype implementation. In this chapter it is assumed that ZERO ENC packets are targeted (Destination IP and port) at the ZERO server process on the *Exit* host.

Naturally, the resulting ZERO ENC packet may exceed the outbound MTU and get fragmented. That will not disrupt the protocol since the U-fields are not used in ZERO ENC.

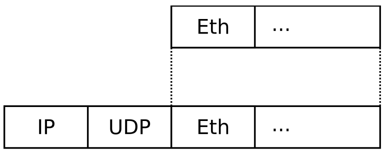

Figure 7: ZERO ENC transformation

3.2.3 Synchronizing the Channel Table

When a new *Channel Table* entry is made on the *Entrance*, the NICE frame (first in it's “session”) is tunneled in the ZERO SYN format. Figure 8 illustrates the ZERO SYN format which is similar to ZERO ENC format in that it encapsulates the original frame allowing *Exit* to extract S-fields for the new *Channel*. However, ZERO SYN has one extra field immediately following the UDP header – the *Z-index* field. It's length is equal to U-field set length rounded up to byte boundaries. The field contains the *Channel Table index* to be used with the encapsulated S-field information.

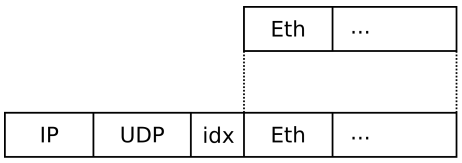

Figure 8: ZERO SYN transformation

Upon receiving ZERO SYN the *Exit* saves the S-field set data into it's *Channel Table* into the given *index*. This allows *Exit* to start using the new *Channel* already with it's first packet. The *Exit* must update the appropriate *Channel Table* entry upon receiving any ZERO SYN packet. This convention allows same format to be used for establishing new *Channels* as well as updating existing ones.

Since the proposed S-fields are not fixed in length different *Channels* may have different bit patterns to check in frames. A design choice is not to synchronize the patterns. Instead, the same S-field recognition logic is used at *Entrance*, as well as *Exit*. That way it is guaranteed that whatever S-field pattern will be determined for a frame at *Entrance*, the same S-field pattern will be determined at *Exit* in the encapsulated frame coming in ZERO SYN format, assuming the determining logic is common for both tunnel ends.

Since the ZERO protocol is simplex, the tunnel endpoints do not use any confirmations or requests for the synchronization. The ZERO SYN packet for a new *Channel* may be lost resulting in *Exit* unable to reconstruct S-fields for following ZERO IP packets. A simple redundancy technique is used to remedy this problem. Every 10 seconds *Entrance* will send a NICE frame again in the ZERO SYN form, allowing the *Exit* to update the entry in it's *Channel Table*. If the *Exit* has missed a ZERO SYN packet it will have a chance to acquire that information in 10 seconds. See Figure 9.

It should be noted that update ZERO SYN is sent in no less than 10 seconds, but only when a frame of that *Channel* is about to be tunneled. No updates are sent for inactive *Channels*. More in that further.

Two specific UDP destination ports are dedicated for both ZERO ENC and ZERO SYN packets to be accepted on *Exit*.

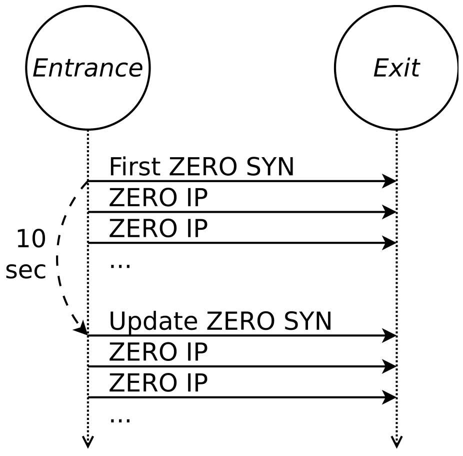

Figure 9: ZERO SYN update interval

3.2.4 Managing the Channel Table

When not explicitly stated this section talks about the processes to manage the *Channel Table* in only the *Entrance*. It is the *Entrance* that does the logic of building and updating the table. The *Exit* is just following the updates brought by ZERO SYN packets.

In case the *Channel Table* is full and a new *Channel* needs to be added, a Least Recently Used (LRU) mechanism is used to replace the oldest record with a new one. Then, the ZERO SYN will be sent for the updated *Channel* which will make *Exit* update the entry in it's *Channel Table*.

For the LRU mechanism to function a new *last_used* column is added to the *Channel Table*. It shows the time when the last packet was sent for a *Channel*. The time value is stored in Unix time format. To make lookup of the oldest entry in *Channel Table* efficient the author proposes a linked list with pointers to *Channel Table* entries. The list complements the LRU mechanism and is called *LRU list*.

The LRU procedure is as follows. For every NICE frame on *Entrance* whenever an existing *Channel index* is found or created in the *Channel Table* the *last_used* time is updated. Also, pointer to a *Channel* entry is moved to the head of list. By doing that the other list entries are falling towards the end of list. The list's head would represent most recently synchronized *Channels*, the tail – least recently (or never) synchronized. Thus, looking for oldest *Channel* becomes trivial – looking at *LRU list*'s tail.

Also, a new *last_synced* column is added to the *Channel Table*. It shows the time when the last ZERO SYN packet was sent for a *Channel*. The time value is stored in Unix time format. It is used to manage *Channel* updates the following way. For every NICE frame on *Entrance* whenever an existing *Channel* is found in the *Channel Table* the *last_synced* time is checked. If it is older or equal to 10 seconds, the necessity to encapsulate the frame into ZERO SYN is triggered. Also, the trigger updates *Channel*'s *last_synced* field with current time. See Figure 8.

So far the S-fields lookup algorithm in *Channel Table* has not been detailed. For the sake of simplicity a full seek of the table looking for *index* of given S-fields is assumed. A realistic approach might be to consider a method of hashing S-fields into an “S-hash” that could be looked up in logarithmic time [16].

Note that the presented algorithm has no locking issues because each frame to be tunneled is completely processed by a single thread without any helper or background processes.

The *Exit* has significantly simpler *Channel Table* management functions: lookup of *Channel* in the *Channel Table* by a given *index*; and updating a given *Channel* with a given S-fields data. Both are rather trivial.

3.2.5 TTL compensation

Most applications set up or default to rather large TTL for new IP packets disregarding if the target is in local or corporate network or in the public Internet. Often used values are in range between 64 and 255. The naive behavior would be to leave the TTL for NICE packet as is. The value will decrease during propagation over the *Transport* network, the target will receive a frame with a lower IP TTL value. Two problems emerge:

- Some applications depend on TTL correctness in their logic. Traceroute is an example.
- If a packet will have lower TTL value than necessary to traverse the *Transport* network, it will be lost in transit.

To solve these problems the following TTL compensation mechanism is used. The idea is to simply add a certain number *ttl_delta* to the TTL field. *ttl_delta* would represent the number of hops between the tunnel endpoints. However, there are three issues with this idea:

- If *ttl_delta* is added at *Entrance*, for NICE packets with TTL already at the maximum 255 or close (TTL + *ttl_delta* > 255) the method would lose correctness.
- If *ttl_delta* is added at *Exit*, for NICE packets with low TTL (TTL < *ttl_delta*) the packet would be lost in transit.
- The path from *Entrance* to *Exit* may change at some point in time due to routing change in Internet. In that case a constant *ttl_delta* would not represent the actual hop count between tunnel ends and the method would lose correctness.

The author choses not to fight the second issue and tunnel all NICE frames with low TTL in the ZERO ENC form. Author's practical experience leads to believe such frames to be rare for typical Internet applications. Both other issues are solved by two functions:

- Compensating at the *Exit*, meaning the *Exit* always increases TTL of the received ZERO IP packet by *ttl_delta* value.
- Using a dynamic *ttl_delta* update process.

The *ttl_delta* update process is a part of the earlier described *Channel* synchronization process. The TTL of ZERO SYN packet will always be set to 255 at *Entrance*. The original TTL altogether with the original frame is encapsulated inside the ZERO SYN packet. Thus, upon receiving a ZERO SYN the *Exit* is synchronizing not only S-fields but also the *ttl_delta*. It is looking for the incoming ZERO SYN packet TTL (*syn_ttl*) and calculating the new *ttl_delta*:

$$ttl\_delta = 255 - syn\_ttl$$

If the new *ttl_delta* differs from previously used, it is updated. This simple check is done for every received ZERO SYN. But the update of *ttl_delta* will be done in those rare situations when it actually changes. See Figure 10.

Note that the proposed TTL compensation mechanism ensures correct *ttl_delta* only at *Exit*. The Entrance does not know that number and so it can not correctly evaluate the “too low TTL” criteria. If the *Entrance* also has the reverse tunnel from the *Exit* side, it could look for detected *ttl_delta* in reverse direction and hope the path in both directions is the same or at least with the same hop count. The author proposes a different approach that suits the asymmetric infrastructures like satellite communications. The delta numbers will have different semantics on each tunnel end. The *Exit* calculates the *ttl_delta*, as described. But the *Entrance* has a different number – *ttl_min*. It is proposed to be a configuration parameter and to serve as the “too low TTL” criteria. Network administrators may choose to set that number little lower than the lowest default TTL for the common applications in tunneled infrastructure, e.g., 63.

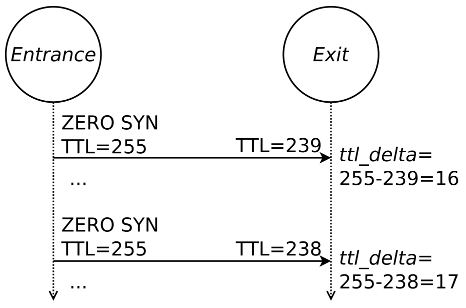

Figure 10: ttl_delta update example

3.2.6 Tunneling the Internet

While Destination IP address is one of the S-fields the ZERO protocol mandates to create new *Channel* for any IP host connected to. That would be acceptable for communication among two parts of small to medium sized network (more on this further). But it does not scale well if the number of IP destinations is large or unpredictable. It would be so if a branch office is going to access Internet through a ZERO tunnel to central office, for instance.

Note that the returning traffic in the opposite tunnel would scale well because the packets in there would have limited set of Destination IP addresses (back to branch office) and the highly variable Source IP addresses would be carried along unchanged in ZERO IP packet. In other words, the scalability problem is with “local to global” *Channels*, not with “global to local” or “local to local” *Channels*. Stub or branch IP networks are meant by “local” that have small number of inside subnets and reach out to Internet through one gateway.

To address this scalability issue the following terminology and solution is proposed. The tunnels that are directed towards branch networks are called *Direct Tunnels* (*DT*) and it's *Channels* are called *Direct Channels* (*DC*). The tunnels that are directed towards central and global networks are called *Indirect Tunnels* (*IT*) and it's *Channels* are called *Indirect Channels* (*IC*). Both *Entrance* and *Exit* will have a configuration parameter for a mutual tunnel that determines tunnel's mode: *DT* or *IT*. Figure 11 shows an example topology with asymmetric tunnel types.

And the solution is to add one step before packet transformation for *ICs*: to swap Source IP and Destination IP addresses. Further, Source IP becomes S-field instead of Destination IP and the former Destination IP address is carried in the transformed packet as Source IP. This solves the scalability problem, assuming the branch networks are not subject to scalability issues themselves.

Theorem 1: *Every branch computer requires only one Channel in each direction through ZERO tunnel to communicate with all global Internet hosts connected behind the central office network.*

Assumptions: the MAC and IP addresses of computer and gateway are fixed. The tunnel from branch is configured *Indirect* on both ends, the tunnel from central network is configured *Direct* on both ends.

Proof. 1) The *Entrance* in branch network will create *Indirect Channel* for traffic from the computer. A priori, the *IC* will be represented by Source MAC of the computer, Destination MAC of the gateway, and Source IP of the computer. Since all these three parameters are fixed, the ZERO tunnel will use and reuse single *IC* for all traffic from branch computer to global Internet hosts.

2) The *Entrance* in central network will create *Direct Channel* for traffic to the computer. A

priori, the *DC* will be represented by Source MAC of the gateway, Destination MAC of the computer, and Destination IP of the computer. Since all these three parameters are fixed, the ZERO tunnel will use and reuse single *DC* for all traffic from global Internet hosts to the local computer.

From 1) and 2) Theorem 1 is proved.

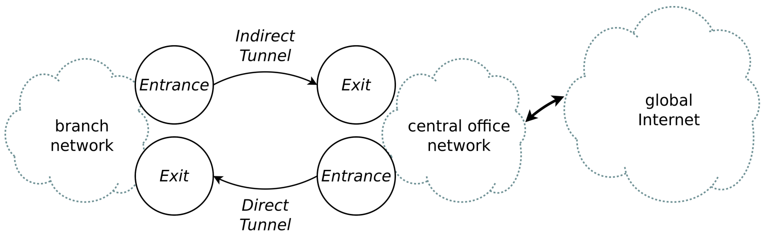

Figure 11: *Direct* and *Indirect Tunnels* topology example

It should be noted that *ICs* are equally well suited for traffic through a computer chosen gateway, as well as through a Proxy ARP gateway [17]. No matter how delusional ARP table would form in local computers, they would generate same frames for Proxy ARP gateway as for a computer chosen gateway: computer's Source MAC and Source IP, Destination MAC of gateway (Proxy ARP) and Destination IP of global host. When a tunnel is configured *Indirect* the *Entrance* would again create an *IC* and reuse it for all traffic from a computer to Internet via Proxy ARP gateway.

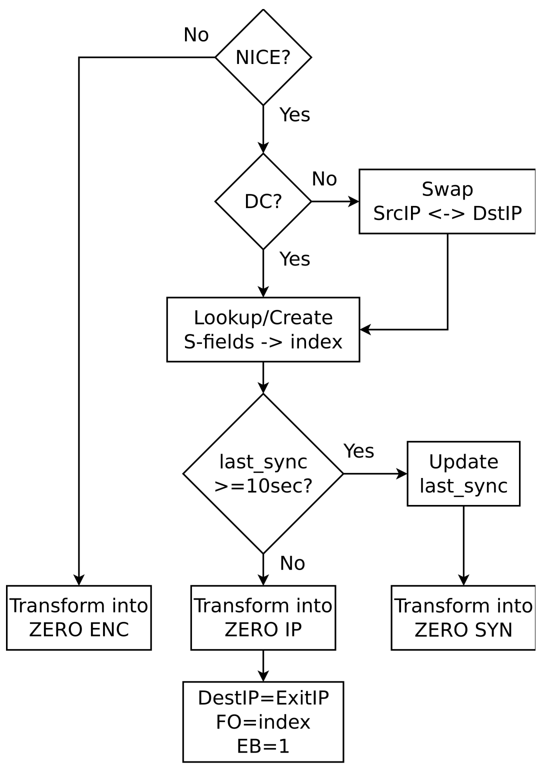

Figure 12: Entrance flow chart

3.2.7 Dealing with convergence

As long as the *Channel Table* is not full (according to Theorem 1 – branch office has less computers than *Channel Table* size of 8192) the *Channel Tables* at *Entrance* and *Exit* will eventually converge in the presence of packet loss and will not change any more.

Once the *Channel Tables* at *Entrance* and *Exit* have converged the tunneling is always correct even in the presence of packet loss in *Transport* network.

To minimize corruption of tunneled packets during convergence the *Exit* discards any packet for unestablished *Channel*.

Theorem 2: *In converged state ZERO protocol will correctly tunnel NICE frames with zero-overhead through Transport infrastructures that: (T1) do not fragment IPv4 packets with size<=1500; (T2) do not filter IPv4 packets by Source IP address; (T3) do not alter IP packet contents besides normal TTL and IP header checksum modification during forwarding.*

Proof. 1) All frame fields that are lost during tunneling are saved in *Channel Table* as S-fields. As long as the *index* value saved in NICE IP FO field is preserved during travel in *Transport* network, all S-fields will be correctly reconstructed on tunnel *Exit*. For this to happen the NICE IP packets may not be fragmented during travel in *Transport* network. That would ruin the *index*. Given (T1) it can be stated that FO field will stay unchanged in *Transport* network and the correct *index* value will be delivered to *Exit*.

2) Besides the unused EB and FO fields of the IPv4 packet embedded in NICE frame (Definition 1) Destination IP field is also modified for travel in *Transport* network. That field is also an S-field and will be reconstructed on *Exit*, if 1) holds. The originally empty EB and FO fields will also be restored to null value on *Exit*.

3) If EB field is not cleared during travel in *Transport* network (T3) the Ext will correctly recognize NICE IP packets.

4) The only remaining critical concern is that Source IP address of original source is carried along into the *Transport* network and there is a possibility an intermediary service provider may filter out packets with unexpected Source IP addresses or set EB. Given (T2) this is not an issue and NICE IP packets will not be filtered.

5) The only other IP field that will change during travel in *Transport* network is TTL. That correctly restored by previously described TTL compensation method.

Using 1), 2), 3), 4) it can be stated that all NICE frame fields that will or may lose original content during tunneling will be restored on *Exit*. With that and 5) Theorem 2 is proved.

3.2.8 Describing ZERO with PTL

The Packet Transformation Language (PTL) is well suited to formally describe the transformation that is at the core of ZERO protocol. The following two paragraphs are PTL transformations for the earlier described ZERO DC and IC algorithms. First, evaluation *functions* are used to find out if a frame is NICE. If so, a transformation continues with NICE frame transformation into a ZERO IP packet. Failure in any step of transformation cancels the whole NICE transformation and the implementation must fall back to full ENC encapsulation.

ZERO DC transformation:

```
equal(([P0-P7],[P50-P63]), (0x45, 0))

flag(P48, 0, 1)

index([P51-P63], ([F0-P0), [P51-P63], [P128-P159]), 1, 4096)
```

ZERO IC transformation:

```
equal(([P0-P7],[P50-P63]), (0x45, 0))
flag(P48, 0, 1)
swap([P96-P127], [P128-P159])
index([P51-P63], ([F0-P0), [P51-P63], [P128-P159]), 1, 4096)
```

In addition to these transformations the ZERO system would need to replicate FTs as *Channel* metadata in SYN packets.

3.3 The prototype implementation

An early ZERO protocol prototype is described in this section that was implemented and tested in GNU/Linux operating system. The implementation is independent of kernel and does not have any obstacles for porting it to other operating systems.

Linux operating system provides several methods for programming Ethernet packet processing:

- Raw sockets
- Linux kernel programming
- TUN/TAP devices

Each method has strengths and weaknesses.

Raw socket data link level interface allows reading inbound frames as well as injecting into network new frames with freely created content [19]. Raw sockets are typically used by network diagnostic programs like *ping*, *traceroute*, *tcpdump*. Raw sockets are available in multiple operating systems but their interfaces to Data link layer may significantly vary [20].

Linux kernel programming is the lowest in the operating system hierarchy in which it is possible to process IP packets and Link layer frames: to receive, modify and send. It is possible to implement widest range of scenarios in this level. Also, this level promises highest performance. However, prototyping in the kernel or a module brings in stability and security concerns that are much lower in the userland implementations. Another drawback is implementation in this level may prove harder to port to other operating systems [21].

Multiple modern operating systems support TUN/TAP devices: Linux, FreeBSD, Solaris, Windows, Mac OS X. TUN/TAP are virtual network devices that provide configuration and semantics very similar to the real ones. Processes may attach to TUN/TAP devices, read and write packets or frames to them. TUN devices operate in OSI Layer 3, TAP devices – in Layer 2. The packets/frames injected into these interfaces are processed by the operating system the same way as in case of real network interfaces.

The he prototype implementation of ZERO protocol is with the TUN/TAP devices because there are less differences between operating systems in that level. It allows easier porting. The choice is also supported by the fact that multiple other tunneling solutions also use these devices. OpenVPN, VTun, OpenSSH are examples of such solutions.

The protocol logic implementation is in Python since it is an easy to program, yet versatile programming language very suitable for prototyping. It does not impose programming paradigms. Object oriented, procedural, imperative, functional paradigms may be used. The implementation is constructed in a way that's easy to port to C language which probably will be the language for production implementation.

3.3.1 ZERO server

The prototype ZERO protocol logic implementation is in a userland process called the ZERO server. It is attached to TUN/TAP devices, reads packets/frames from them, processes and then sends onto another device according to protocol logic. Figure 13 shows data input/output points of the ZERO server. From here on in this section the ZERO server is illustrated functioning in both directions. According to the protocol design terminology ZERO server here is illustrated as both *Entrance* and *Exit*, establishing outbound tunnels and accepting inbound ones, building the appropriate data structures.

There are three different I/O points for the ZERO server:

- The connection to the *Transported* network is established via the TAP interface. It works with Ethernet frames.
- TUN interface is used to send and/or receive the modified ZERO IP packets that go through the *Transport* network. TUN operates with IP packets which is exactly what is needed for

Transport network abstracting away the processing of frames.

- Third point is a standard UDP socket opened on the real *Transport* interface. It is used to send and/or receive the ZERO ENC and ZERO SYN packets.

For ZERO protocol to work in both directions between two sites a ZERO server needs to be configured and functional in both tunnel ends according to Figure 13 . The A and A' clouds represent two parts of a single *Transported* network. B represents a *Transport* network. The *tun* and *udp* points will have different IP addresses in the prototype implementation. That allows the tunnel end systems to separate incoming NICE IP packets from ZERO ENC and ZERO SYN packets by the operating system simply by routing separate Destination IP addresses to separate interfaces. That way the ZERO server takes for granted that the optimal NICE IP packets will always come in to the *tun* interface, but the encapsulated ZERO ENC and ZERO SYN packets will always come in to the UDP socket of the ZERO server program. This is just a convention for the prototype implementation. Single IP address might also be used with help of some IP forwarding/sorting function judging by ZERO flag (EB) as proposed by design. The terms *TunIP* and *UDPIP* will be used for those two IP addresses of ZERO server.

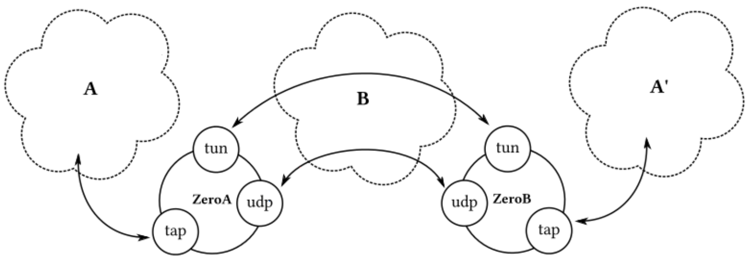

Figure 13: ZERO server bidirectional data paths

3.3.2 Testing environment

For prototype testing an environment was created that resembles a typical Layer 2 tunnel usage between two sites. Conceptually the testing environment is as seen previously in Figure 13. The IP addresses, IP networks, bridging configuration of the testing environment are shown in Figure 14.

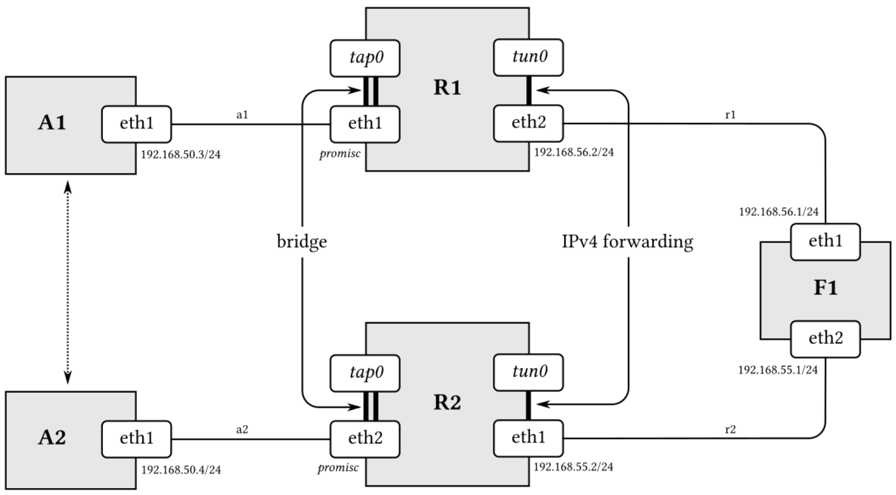

Figure 14: ZERO server prototype testing environment

Five hosts were used in the test:

- A1 and A2 are the end-hosts each residing on it's own side of the *Transported* network (192.168.50.0/24). The end-hosts have no knowledge of the tunneling between them.
- R1 and R2 are the ZERO tunneling end systems, each connected with one interface to the *Transported* network and another interface to the *Transport* network. To test bidirectional communication each of the tunnel end systems acts as *Entrance* for traffic to the other part of the *Transported* network and *Exit* for traffic from the other part.
- F1 is a router simulating internetwork between the R1 and R2 systems.

The R1 interfaces have the following configuration. The *eth1* interface is configured in the promiscuous mode. It allows R1 to accept any frames coming from A1 *Transported* network. The *eth1* interface may be left without IP address which is acceptable for a transparent Layer2 device the R1 system pretends to be. The *eth1* interface is also connected to software bridge created on the R1 system. The bridge further is connected to *tap0* device that is used by the ZERO server process to send and receive Ethernet frames.

On the *Transport* network side ZERO server uses: *tun0* device to send and receive NICE IP packets; UDP socket with a given port to send and receive ZERO ENC and ZERO SYN packets. The R1 host routing table is adjusted so that: packets destined to the other end of tunnel (R2) be properly routed to the next gateway (F1) out via *eth2* interface; packets that are destined to the *TunIP* address (192.168.56.3) are forwarded to *tun0* interface.

Note that *TunIP* address does not need to be configured in any interface, it just needs to be routed to. *UDPIP* address (192.168.56.2) needs to be configured in the *eth2* interface so the R1 system will be able to accept packets for it. It may also serve as the R1 host IP address in the *Transport* network (as in this prototype). UDP datagrams destined to the R1's *UDPIP* address and ZERO server's port will be accepted by the operating system and delivered to the UDP socket of ZERO server.

Same configuration principles as described in previous three paragraphs are applied to the R2 system. The differences are obvious from the X8 picture: the *eth1* and *eth2* interfaces change sides; the *TunIP* address is 192.168.55.3 and the *UDPIP* address is 192.168.55.2.

In addition to the routes of directly connected and configured interfaces the following routing rules are added in the test environment:

R1:

```
# route add -net 192.168.56.3 netmask 255.255.255.255 dev tun0

# route add -net 192.168.55.0 netmask 255.255.255.0 gw 192.168.56.1
```

R2:

```
# route add -net 192.168.55.3 netmask 255.255.255.255 dev tun0

# route add -net 192.168.56.0 netmask 255.255.255.0 gw 192.168.55.1
```

These rules implement previously described routing configuration: local *TunIP* to *tun0*, remote net to F1.

Figure 15 illustrates the data path of packets traveling from A1 computer through a ZERO tunnel to A2 computer in the testing environment. For simplicity this example assumes the *DC* for A1 to A2 is already set up. The numbered arrows are described as follows:

1. Ethernet frame coming from A1 *eth1* device is received in R1 *eth1* device which is in promiscuous mode and accepts any frame.

2. R1 *eth1* device is connected to *tap0* device with a bridge. So, the frame is forwarded into the *tap0*.

3. ZERO server reads the frame coming from *tap0* device.

3a. If the frame is NICE, it is transformed into ZERO IP packet, the Destination IP is set to R2's *TunIP* address (192.168.55.3) and sent out on *tun0* interface.

3a'. The R1 OS uses it's routing table to forward the ZERO IP packet out on *eth2* interface.

3b. If the frame was UGLY (not NICE) it is encapsulated into ZERO ENC packet destined for R2 *UDPIP* address (192.168.55.2) and sent out on *eth2* interface.

4. Packet arrives at next gateway – F1.

5. Packet is routed further to R2 and arrives at R2 *eth1* device.

6a. If it's a ZERO IP packet, it's Destination IP (192.168.55.3) is looked up in the R2 OS routing table and forwarded into *tun0* device.

6a'. ZERO server reads the packet coming in from *tun0* device.

6b. If it's a ZERO ENC packet (Destination IP = 192.168.55.2), it is accepted by the R2 host and delivered to the ZERO server UDP socket.

7. ZERO server restores the original Ethernet frame and sends it out the *tap0* device.

8. R2 *tap0* device is connected to *eth2* device with a bridge. So, the frame is forwarded into the *eth2*.

9. The frame is forwarded out R2's *eth2* device and delivered to A2 host.

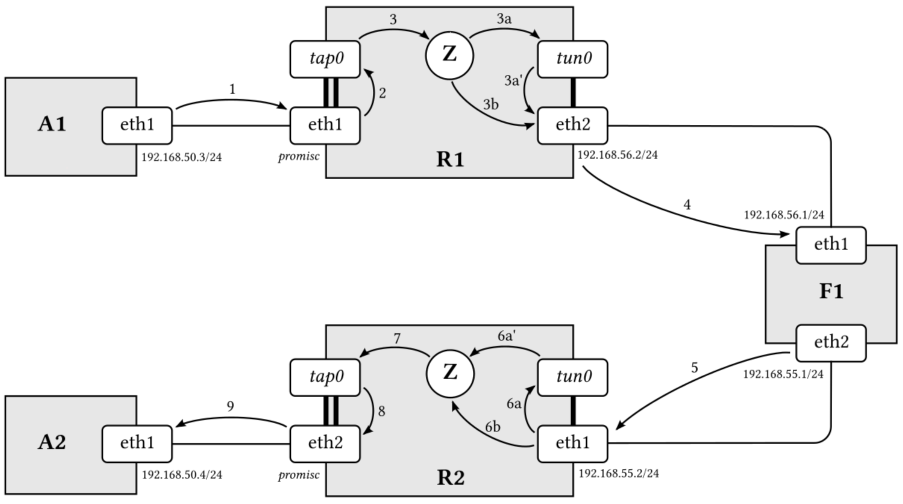

Figure 15: Data path for packets from A1 to A2

3.3.3 Real world test

For the real world test the previously described testing environment was modified the following way. The A2 computer was replaced by a conventional gateway to global Internet. Still, A1 and the gateway were in two parts of same *Transported* network.

During the real world test the A1 computer was used for typical Internet applications: visiting Web sites, watching video, download files. All applications used Client/Server model and only download requests were used, no uploads. Naturally, the traffic volume was expected to be asymmetric – more and larger packets from Servers to Client.

The test lasted for 8 minutes. Both ZERO servers collected protocol statistics: protocol counters, NICE and UGLY counters, frame size distribution.

Table 3 shows the protocol counters for both tunnel ends. The numbers represent packets coming into the tunnel from *Transported* network side.

Table 3: Protocol counters for both tunnel ends

Protocol	R1	R2
ARP	9	5
IPv4	61273	125472

Table 4 shows the counters of NICE and UGLY frames detected by the tunnel ends. From Table 3 and Table 4 it can be deduce:

- All IP packets coming into R2 tunnel (as part of Client request) are NICE and eligible for optimal tunneling. Only 5 incoming frames are UGLY, the ARP frames.
- Almost all IP packets coming into R1 tunnel are NICE. It's 99.97%, still a good result.

Table 4: NICE and UGLY counters

Status	R1	R2
NICE	61253	125472
UGLY	34	5

Table 5 and Table 6 show most popular frame sizes coming into R1 and R2, respectively. These confirm the asymmetric traffic nature of this experiment. 96.58% of the frames coming into R2 were maximum size frames – 1514 bytes (1500 MTU + frame header). And all those frames were NICE and were transported in the optimal NICE IP format. The total percentage of frames that were transported in NICE IP format was 99.94%. The rest were either ZERO ENC (for the UGLY frames) or ZERO SYN (for syncing NICE frame *Channels*).

Table 5: R1 entering frame counters

Frame size	Count
66	55351
78	3502
86	648
54	307
74	207

Table 6: R2 entering frame counters

Frame size	Count
1514	121185
1484	1800
66	431
74	155
316	116

3.3.4 Comparison to OpenVPN

The data from previous real world test was used to model behavior of an unconditional full encapsulation protocol in the same situation. OpenVPN is a popular example of such existing implementations. OpenVPN has many additional functions, like encryption and signing. For this evaluation only the basic tunnel was considered that encapsulates frames into UDP datagrams. It's format is semantically same as ZERO ENC.

Table 7 and Table 8 show Zero server and OpenVPN comparison for both tunnel ends. The numbers show total packet count and size coming out of the ZERO servers into the *Transport* network.

Table 7: R1 transport packets statistics

(R1)	ZERO	OpenVPN
Packet count	61289	61309
Total volume, bytes	4430144	7001420

Table 8: R2 transport packets statistics

(R2)	ZERO	OpenVPN
Packet count	125491	248473
Total volume, bytes	187155054	197586861

ZERO server does not win much in packet count on R1 because that system mostly tunnels small request and acknowledgement packets that don't get fragmented with OpenVPN. The volume, however, differs greatly because ZERO server tunnels NICE packets optimally as opposed to OpenVPN that encapsulates unconditionally. In this aspect R1's ZERO server uses only 63.27% of the volume that OpenVPN uses.

R2 has different gain. That ZERO server mostly tunnels large packets. Because of optimal tunneling ZERO server needs only 50.5% of the packet count required by OpenVPN, which, on the other hand, would encapsulate and fragment all maximum sized frames. The relative reduction in total volume is not significant since the encapsulation overhead is relatively small.

3.4 ZERO discussion

The core ZERO protocol described earlier is intended primarily for the controlled service provider networks adhering to all conditions stated in Theorems 1 and 2 – it cannot be guaranteed to work over public Internet due to various anti-spoofing or connection-tracking filters employed by some ISPs. Therefore in this Section several extensions are discussed that enable ZERO protocol operation over public Internet at the expense of non-essential tunneled frame modification or occasional integrity violation. Tests show that the proposed extensions work well both over national and international public Internet.

3.4.1 Multi-point tunneling topology

So far the ZERO protocol is described as operating from one *Entrance* host to one *Exit* host. The following two data structures are associated with a single tunnel, one – for each end. The *Entrance* data structure:

- configuration parameters: *Entrance* IP, *Exit* IP, *ttl_min*, implementation specific (UDP port, attached interfaces, etc.);
- *Channel Table* – built and updated during operation.

The *Exit* data structure:

- configuration parameters: *Entrance* IP, *Exit* IP, implementation specific (UDP port, attached interfaces, etc.);
- *ttl_delta* – learned during operation from ZERO SYN packets;
- shadow *Channel Table* – a copy of the *Entrance Channel Table* that is learned during *End* operation via ZERO SYN packets;

On both ends the data structures are identified by the <*Entrance* IP, *Exit* IP> pair. To create a tunnel to another tunneling host, a new data structure is created identified by the <*Entrance* IP, *Another Exit* IP> pair.

With these definitions it becomes possible to build multi-point tunneling topologies where any tunneling host can create a tunnel with any other, thus building full or partial mesh tunneling topologies similar to MPLS VPN.

Figure 16 shows an example of 3 tunneling hosts configuration. The data structures are depicted as gray boxes and identified with host letters. For instance, AB identifies an *Entrance* data structure for tunnel from A host to B host. The shadow tables have the same notion, except marked with asterisk, for instance, AB' identifies an *Exit* data structure for tunnel from A host to B host.

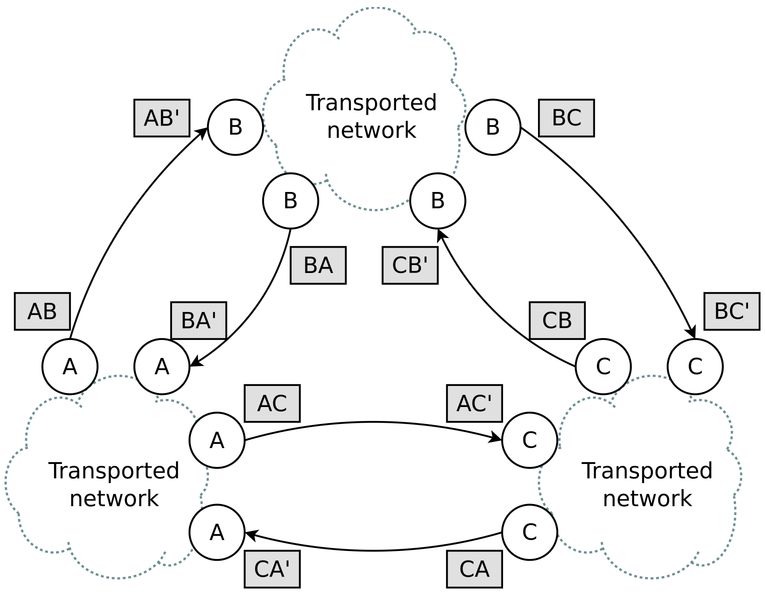

Figure 16: Mesh ZERO tunneling topology example

3.4.2 On security implications

ZERO protocol, like any unencrypted tunneling protocol (such as GRE [9]) is prone to third party injecting spoofed tunnel packets. The described use of Evil bit (EB) is a weak authentication mechanism to minimize spoofing. A stronger mechanism to minimize consequences of spoofing would be to scramble data of tunneled packets. A simple scrambling method would be to XOR the data of tunneled packets with a random shared bit sequence – this would corrupt the spoofed packets during descrambling. An even stronger protection method would be to use DES as the scrambling mechanism – DES encryption does not change data length, yet potentially makes ZERO protocol VPN grade secure.

3.4.3 IPv6 handling

According to Definition 1 NICE can only be frames containing IPv4 packets. Actually, Ethernet frames containing IPv6 packets can also be transformed into IPv4 packets and tunneled with zero-overhead. Since the *Channel Table* stores Destination IP address of IP packets contained in NICE frames, the said address is not included in NICE IP tunnel packets. In case of IPv6 this allows to reduce 16 bytes in NICE IP tunnel packets. Four more bytes can be reduced by mapping three IPv6 header fields Payload length, *Next header*, *Hop limit* into three IPv4 fields *Total length*, *Protocol*, *Time to live*, respectively. This results in 20 bytes reduced for any Ethernet frame containing IPv6 packet sufficient to add whole IPv4 header (*sans*, the three fields mapped from IPv6 header).

3.4.4 Possibility to use IPv4 Ident field

The often enabled TCP segmentation offload functionality in modern Ethernet NIC (Network interface cards) and connection tracking option in Linux kernel Netfilter module tend to defragment IP packets in transit and thus interfere with the FO field use in core ZERO protocol as described earlier. For transport networks where FO field cannot be used as X-field, there is an option to use *identification* field instead. Although use of *identification* field formally violates Theorem 2., it usually does not cause any problems in practice as long as ZERO is the only protocol manipulating the *identification* field.

The intended use of Ident field in IPv4 is described in the RFC 791 [22] that defines fundamentals of IPv4 protocol:

"The identification field is used to distinguish the fragments of one datagram from those of another. The originating protocol module of an internet datagram sets the identification field to a value that must be unique for that source-destination pair and protocol for the time the datagram will be active in the internet system. The originating protocol module of a complete datagram sets the more-fragments flag to zero and the fragment offset to zero."

Therefore it is rather safe to assume that the only requirement for the content of *identification* field is that it needs to be unique for any (Source, Destination, Protocol) triplets for the time packets are active and could be fragmented. It is not mandated for the content be chosen by any specific scheme or algorithm. The *Ident* values may as well be sequence numbers that get incremented for any new packet.

Study of Linux kernel networking subsystem and its source code reveals the algorithm this network operating system uses to generate Ident values.

From [23]:

```
static inline void ip_select_ident(struct iphdr *iph, struct dst_entry *dst, struct sock *sk)
{
        if (iph->frag_off & htons(IP_DF)) {
...
```

```
                iph->id = (sk && inet_sk(sk)->inet_daddr) ?
                                        htons(inet_sk(sk)->inet_id++) : 0;
        } else
                __ip_select_ident(iph, dst, 0);
}
```

From [24]:

```
void __ip_select_ident(struct iphdr *iph, struct dst_entry *dst, int more)
{
...
                if (rt->peer) {
                        iph->id = htons(inet_getid(rt->peer, more));
                        return;
...
        ip_select_fb_ident(iph);
}
```

From the source code and [16] p.540.-541. It can be deduced that Linux generates *Ident* value for a packet with the following algorithm in a pseudo language:

```
if (DF flag is set)
        if (packet is intended for a socket)
                Ident := inet_sk(sk)->inet_id++      //  the next value from socket inet_id
        else
                Ident := 0
else
        lookup inet_peer structure (for Destination IP)
        if (inet_peer exists)
                Ident := inet_getid(peer, more);     //  the next value from inet_peer struct
        else
                // A fallback function generates new "unpredictable" ID
                // combining Destination IP and previous "unpredictable" ip_fallback_id value
                Ident := ip_select_fb_ident(iph);
```

To sum up, in rare cases *Ident* is set to 0 or a new pseudo random value, but otherwise (most of the time) it is generated as an increment of previously used value. In other words, *Ident* really is a "serial number" of packets. The value has no other meaning.

The conclusion is supported by observing real world traffic. The following *tcpdump* output shows beginning of an SSH session:

- the client starts with a random ID 28864 and increments it for the next packet (28865 and on);
- the server starts with 0 Ident and continues with a random ID 52432.

```
    14:34:06.851169 IP (tos 0x0, ttl 64, id 28864, offset 0, flags [DF], proto TCP (6), length 60)
           1.2.3.1.52663 > 1.2.3.254.22: Flags [S], cksum 0x292d (incorrect -> 0xb884), seq
3800618708, win 14600, options [mss 1460,sackOK,TS val 3515924 ecr 0,nop,wscale 7], length 0
    14:34:06.851341 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 60)
          1.2.3.254.22 > 1.2.3.1.52663: Flags [S.], cksum 0xecd9 (correct), seq 3117610857, ack
3800618709, win 5792, options [mss 1460,sackOK,TS val 17578938 ecr 3515924,nop,wscale 6], length 0
    14:34:06.851362 IP (tos 0x0, ttl 64, id 28865, offset 0, flags [DF], proto TCP (6), length 52)
        1.2.3.1.52663 > 1.2.3.254.22: Flags [.], cksum 0x2925 (incorrect -> 0x31d2), seq 1, ack 1,
win 115, options [nop,nop,TS val 3515924 ecr 17578938], length 0
    14:34:06.859320 IP (tos 0x0, ttl 64, id 52432, offset 0, flags [DF], proto TCP (6), length 93)
        1.2.3.254.22 > 1.2.3.1.52663: Flags [P.], cksum 0x5957 (correct), seq 1:42, ack 1, win 91,
options [nop,nop,TS val 17578940 ecr 3515924], length 41
```

Since ZERO IP packets are not expected to be fragmented in the transport network, the content of *Ident* field becomes irrelevant. Thus *Ident* field can serve as alternative to FO field for the role of U-field. In that case the 16 bits of *Ident* field can be used as INDEX, provided that on tunnel *Exit* identification field is filled with a pseudo-random or incremental values.

However, for some degree of compatibility with eventual fragmentation and defragmentation on the transport network the author proposes to set aside few bits of the *Ident* field for unambiguity of packets closely following one another. So, it is proposed to use the *Ident* field in the following format:

- 0.-12. bits: INDEX field, contains *Channel* number (can have 8192 *Channels*)
- 13.-15. bits: ID field, has the lowest 3 bits of the original Ident value (8 unique packets in a row)

3.4.5 Alternative treatment of TTL field

Besides the TTL compensation method described earlier another and rather obvious method to recreate the original TTL value at the tunnel Exit would be to include TTL field in the S-field list. Then, there would be no minimum TTL requirement for NICE classification, any TTL value would be allowed for NICE packets. Also, the earlier TTL compensation method may recreate incorrect TTL value if a load sharing with oscillating hop count appears on the path in the transport network. Adding TTL to S-field would guaranty correct TTL recreation. And since the original TTL value would be saved in the Channel table, tunnel TTL field may be set to value 255 or any large number to make sure the packet does not expire in the transport network.

A drawback of adding TTL field to S-fields is that variations in this field would produce separate Channels for otherwise equal S-field lists.

3.4.6 ZERO NAPT traversal

ZERO protocol as described so far is not designed to traverse Network address translator (NAT) or Network address port translator (NAPT) [64]. Nevertheless such functionality would be of great interest due to massive and increasing use of NAPT devices for Internet access. ZERO protocol can be made to operate via NAPT gateway by forcing the ZERO tunnel endpoints to behave as regular UDP client and server; additionally also two more NICE frame criteria should be introduced:

- *The packet needs to include a TCP or UDP segment;*
- *For TCP segment Urgent pointer field must be 0.*

Under these conditions all data exchange between ZERO tunnel server and client can be carried in UDP packets. ZERO ENC and ZERO SYN are UDP packets by definition and thus do not require any modification apart from UDP port selection in line with client-server model to enable NAPT traversal. Meanwhile ZERO IP packets can be converted into UDP packets (without increasing their length) using the same field substitution principle used earlier on IP header fields, only this time applied to UDP and TCP header fields:

- Original source and destination ports of TCP or UDP header are added to the S-fields list along with source and destination IP addresses and IP protocol number of IP header; afterwards source and destination IP addresses and port numbers are overwritten by new source and destination IP addresses and UDP port numbers in line with client-server model to enable NAPT traversal between ZERO client and server.

- The IP packets containing TCP segment are transformed into IP packets containing UDP segment (without increasing their length, see Fig. 14) by changing IP protocol field value from 6 to 17 to make it appear as UDP segment. Additionally restorable on Exit TCP header fields *Checksum* and *Urgent Pointer* are deleted (4 bytes total) to provide room for UDP header fields *Length* and *Checksum* (also 4 bytes total).
- IP identification field must be used as U-field, because FO field cannot be used with NAPT, which by definition performs packet defragmentation.

The reverse UDP-to-TCP transformation performed on Exit can be easily deduced.

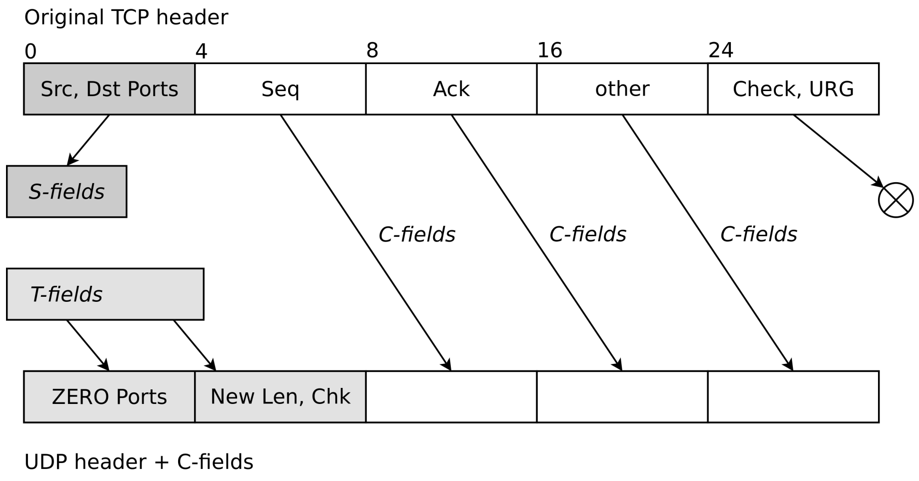

Figure 17: TCP to UDP segment header transformation

The proposed NAPT solution provides ZERO tunneling efficiency gain for small or lightly used networks. Even for large networks where the *Channel Table* might get often overloaded, the ZERO protocol would still maintain communications integrity by frequently replacing old *Channel* entries with new ones. This fully UDP-based ZERO protocol version is compatible with Hole punching techniques [65] popular in P2P networks for one or both ZERO tunnel endpoints behind 3rd party NAT.

3.4.7 Channel synchronization overhead and impact on fragmentation

Since channel synchronization encapsulates a whole NICE frame into a UDP datagram (ZERO SYN) similarly to encapsulation of UGLY frames (ZERO ENC), a question may arise if it leads to fragmentation of ZERO SYN packets, and if yes then how often it happens.

The ZERO SYN structure (Figure 9) clearly shows that any NICE packet is prepended during encapsulation by the fields shown in Table 9 (from right to left).

Table 9: ZERO SYN overhead structure

Appended data structure	**Length, Bytes**
The NICE Layer2 header is inherited, not stripped away	14 (+ optional tags)
Index field	2
UDP header with ZERO port numbers	8
IP header for transport network	20 (+ options)
TOTAL SYN overhead	**44 (+ options)**

It can be seen that minimum ZERO SYN overhead is 44 bytes. This implies the maximum NICE packet size that can be encapsulated in ZERO SYN and still produce a packet no larger than MTU of the transport interface. A new term is proposed here – MEPS (Maximum encapsulated packet size) and it is calculated the following way:

$$MEPS = transport\_MTU - SYN\_overhead$$

In a typical LAN environment that has transport_MTU=1500B and the NICE Layer2 header is a standard, non-tagged Ethernet header, the MEFS would be equal to 1456 bytes:

$$MEPS_{LAN} = 1500 - 44 = 1456$$

If a NICE packet is larger than MEPS, its ZERO SYN encapsulated packet would be larger than transport MTU, hence it would be fragmented upon transmission into transport network. This is not an unrecoverable event by any means, but certainly not welcome for ZERO protocol that aims at minimising fragmentation.

However, if the size of a NICE packet is smaller or equal to MEPS, its ZERO SYN encapsulated packet would fit into the transport MTU, hence it would not be fragmented upon transmission into transport network.

As already described the ZERO SYN encapsulation is done rather rarely during a channel lifetime: at the beginning – with the first packet; and then resynchronized no more often than every 10 seconds. Whatever the traffic intensity between the channel ends, no NICE packets are fragmented between the ZERO SYN updates. Thus, for intensive NICE traffic the percentage of fragmented packets is very low relatively to the whole number of NICE packets. Real world testing described further proves this point.

Elaborating further, one can see that TCP protocol has a very convenient feature – it always starts with a small initiating segment (carries TCP synchronization information in its header, no data), and the first returning packet is also a small TCP segment with similar purpose and size. And so it can be stated:

For any channel (forward, as well as returning) that gets synchronized with a start of a TCP session the first ZERO SYN encapsulated packet will not be fragmented on the transport network.

Also, if a TCP session has started with a ZERO SYN and is less than 10 seconds long, it will not encounter another ZERO SYN encapsulation and hence the session will have no fragmented packets on the transport network at all, even if all the actual TCP session data packets would be of MTU size. This statement is empirically proved in the real world tests described further. And it's manifestations are called here on as "small sync factor". It expresses the degree at which SYN messages encapsulate frames small enough that the resulting ZERO SYN packets do not get fragmented on transport network.

3.5 Real world testing of a kernel module implementation

The author has implemented and tested the ZERO protocol in heavy real life network traffic conditions. Although earlier and simpler versions of ZERO protocol were created by others, the author of this thesis has created an ultimately efficient Linux kernel implementation of ZERO protocol. It is briefly described in this section along with some real world testing results and observations.

3.5.1 The Linux kernel module ZERO implementation

The ZERO protocol support is completely implemented as a Linux kernel module similar to other Linux networking ad-hoc functions like iptables or IPsec. This avoids CPU context switches improving protocol throughput, however requires high quality since a kernel module crash may render the whole system unusable.

To activate this implementation one must load the ZERO kernel module. This can be achieved by configuring the Linux system to load the module on booting, or loaded later on demand. Also, the module can be unloaded at any time and thus leave the system without ZERO protocol support. Upon loading the module one can specify several parameters that influence the protocol behavior. At this point the module name is "nfzero" in which the "nf" letters mean that this implementation uses Linux kernel NetFilter subsystem. Also, the parameters all start with "nfzero_" prefix:

- `nfzero_entr_ip`: tunnel Entrance IP address;
- `nfzero_exit_ip`: tunnel Exit IP address;
- `nfzero_tun_if`: tunneled interface;
- `nfzero_chtab_size`: tunnel channel table size (default = 64);
- `nfzero_tunring_len`: tunneled packets backlog length (default = 50);
- `nfzero_tunmode`: tunnel mode (default = 2):
    - 1: always encapsulating;
    - 2: full ZERO protocol;
- `nfzero_tx_direct`: tunnel sender direction:
    - 1: Direct Channel;
    - 2: Indirect Channel;
- `nfzero_debug_level`: debug level (default = 2):
    - 0: no logging;
    - 1: log start/stop events;
    - 2: log significant events (like channel creation);
    - 3: log everything (multiple stages for any frame).

An example command line of loading the module may look like this:

```
    /sbin/insmod ./nfzero.ko nfzero_entr_ip=1.2.3.4 nfzero_exit_ip=5.6.7.8 \
nfzero_tun_if=eth1 nfzero_chtab_size=128 nfzero_tunring_len=50 \
nfzero_tunmode=2 nfzero_tx_direct=1 nfzero_debug_level=2
```

3.5.2 The performance of this implementation

This implementation has been tested mostly for functionality but also few performance tests have been done on dedicated x86 computers and also virtual machines in the IMCS UL Scientific Cloud. Both setups utilize 1Gbps network infrastructure. The results show that the kernel implementation is functioning as described in this document, is stable and shows near line rate performance (up to 865Mbps throughput on 1Gbps links).

The dedicated computers that performed tunnel end functions during testing were based on 2 core 1,86GHz Intel D2500 Atom processor on D2500CC desktop board with dual Intel 1Gbps Ethernet interfaces. While these computers are not high performance desktop systems, their network performance is adequate for low level 1Gbps networking functions like Ethernet bridging and IP routing. Two such computers were connected back-to-back to simulate a transport network. The other network interface was used for tunneled network connection on both tunnel ends. Both end stations also were rather modest computers: an Intel Atom based net-top and an Intel G860 based desktop. Still, both of these computers are capable of passing Ethernet traffic at 1Gbps line rate.

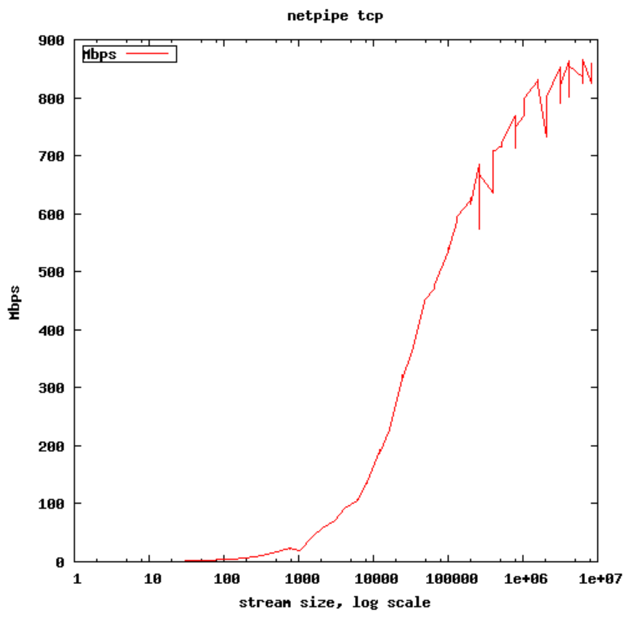

Figure 18: TCP acceleration over ZERO tunnel

During TCP tests the timing of first two packets of the three way handshake give some insight into RTT over the tunnel. Tcpdump data from ten of the tests show that RTT between the two end stations was in range of 500-600 microseconds. A netpipe-tcp test shows that TCP acceleration over the tunnel is very typical for the given stream sizes, as seen in Figure 18.

Further details of the kernel module implementation are left outside this document.

3.5.3 Practical SOHO tunneling

The kernel ZERO implementation has been used as a SOHO (small office/home office) tunneling solution in real home network for over a year. Mostly the usage has been: Web browsing, E-mail, on-demand video (youtube.com and similar), teleconferencing. One tunnel end was at the home premises, the other end – at remote site in different city. The tunneled packets traveled through more than one Internet service provider.

Since the real home network accesses Internet through NAT gateway, the tunnel has been configured to use both IP addresses for S-fields and disregard the channel direction (DC/IC). This generates heavier channel table utilization but has never shown any impact on Internet service quality. Simply put, the users do not notice that any additional tunneling is performed for their Internet access.

A set of statistics for a 31 day period in this setup has been accounted for and given in Table 10.

Table 10: ZERO statistics for one month SOHO usage

Premises	Packets	NICE	NICE, SYN	UGLY
SOHO	Entrance packet counts	3286143, 96,5%	95852, 2,8%	22840, 0,7%
SOHO	Entrance byte counts	241715177, 93,0%	16750197, 6,4%	1417140 , 0,5%
Remote	Entrance packet counts	6807934, 98,4%	87025, 1,3%	23510, 0,3%
Remote	Entrance byte counts	9694042054, 99,8%	16613563, 0,2%	1363580, 0,0%

In this scenario ZERO protocol classified the average of 99,5% (99,3% on one end and 99,7% on the other) of the real world traffic frames as NICE. These results are weaker than the previously shown real world experiment in subsection 3.3.3 (99,94%). The reason for higher percentage of UGLY frames in this particular case may be described by the fact that the client on the SOHO premises acquires IP parameters from DHCP server on the remote tunnel end, and for communication integrity reasons the kernel ZERO implementation classifies all DHCP frames as UGLY.

3.5.4 SOHO channel synchronization overhead

To observe the different ZERO SYN fragmentation rate for long and short TCP sessions (see discussion at 3.4.5) two different test scenarios were performed:

1. Long TCP sessions with MTU packets – pulling a 700MB file from a server only few hops away from the tunnel remote end. This scenario was run for 3 times – one pull at a time.
2. Short TCP sessions – pulling a 1MB file with 11 seconds intervals from the same server. The intervals are there to give time for the channel to outdate and have a need to resynchronise on start of next TCP session. This scenario was run for 10 minutes.

The kernel implementation was improved to collect and dump out per-channel statistics: total packets, total bytes, total ZERO SYNs, total fragmented ZERO SYNs. The test results are in Table 11.

Table 11: SOHO synchronization overhead statistics

Scenario	**Packets**	**Bytes**	**SYNs**	**Fragd SYNs**
Long TCP sessions, tunn. to client	1412722	2118545373	175	169
Long TCP sessions, tunn. to server	690894	35976562	175	0
Short TCP sessions, tunn. to client	37889	56403591	54	1
Short TCP sessions, tunn. to server	12161	643252	54	0

The results show what was expected after the mentioned discussion:

1. Long TCP sessions with mostly MTU sized packets generate fragmented ZERO SYN packets because the need to resynchronize channel often (169 of 175) arises during encounter of another MTU sized packet.
2. It must be noted that the fragmentation arises only on the channel towards the client – where the large packets go. The other direction tunnel sends only small request segments and TCP ACK segments with no data of significant volume for the whole long TCP session. That direction has no fragmentation during the whole experiment.
3. However, short TCP sessions benefit from "small sync factor" and produce no fragmentation

in any of the two directions. In this particular test one fragmentation had occurred and is believed to be caused by TCP FIN packets interfering with the pull intervals. More research of the “small sync factor” is anticipated in the future work.

3.5.5 High load Web server channel synchronization overhead

The same kernel implementation was also put to the test in a different setup: a short tunnel that extends a public VLAN through a dedicated LAN segment. A high load Web server was disconnected from the direct connection to the public VLAN and connected to the other end of the tunnel. That way the server was still accessible with the same MAC and IP addresses – only through the test tunnel.

First, the functionality and stability of the ZERO implementation was good. No changes in the traffic patterns and network load were noticed. Still, some performance penalty was expected in form of increased latency and lower maximum throughput. ICMP echo replies below 500 microseconds from another machine in the VLAN through the tunnel were observed. Compared to the sub 200 microseconds in directly connected setup the sub 300 microseconds added latency was considered acceptable. At least for the sake of this experiment. The latency aspects were not investigated further. The experiment was run for 1 hour and then the server was reconnected back to the VLAN directly.

The “small sync factor” was main topic of interest in this experiment. Since the server is used for software updates, it predominantly has short-time visitors that quickly fetch small files of software indexes and leave to return only hours or even days later. The S-fields again included Source and Destination IP addresses, so the tunnel was expected to generate new channel for any new visitor. The experiment statistics are given in Table 12.

Table 12: High Web load synchronization overhead statistics

Scenario	**Sessions**	**Total sessn. time,s**	**Packets**	**Bytes**	**SYNs**	**Fragd SYNs**	**Frags/ packets**	**Frags/ SYNs**
Tunn. to client	3164	53331	9852 k	14317 MB	6666	390	0,0040%	**5,85%**
Tunn. to server	3364	57416	4599 k	284 MB	7107	166	0,0036%	**2,33%**

This experiment gave result of only 5,85% of ZERO SYN packets being fragmented in the direction towards client. And only 2,33% of ZERO SYN packets were being fragmented in the direction towards server. This last result is somewhat spoiled by the fact that the server also did some large volume information pulling during the test, receiving large packets and generating fragmented ZERO SYN packets. Still, the results are worth noting and expose an argument that ZERO protocol requirement to resynchronize channels for redundancy gives very low (few percent) contribution to fragmentation in real life scenarios.

Further analysis of the experiment data shows that no fragmentation was detected for all 2796 sessions that were shorter than 11 seconds (88,37% of all 3164 sessions). The same observation is for the opposite tunnel. This proves that the "small sync factor" of ZERO protocol is very beneficial in real life.

3.6 Conclusions on ZERO protocol

The core ZERO protocol for efficient Ethernet-over-IP tunneling has been presented in this chapter along with formal proof of its transparency, efficiency, and convergence. The core ZERO protocol is suitable for controlled service-provider networks where guaranteed transparency and efficiency is required. The core protocol has been developed with satellite service-provider networks in mind, but it could equally benefit also other infrastructures where true L2 transparency is required for the Internet of things or other purposes.

The overhead-less nature of ZERO tunneling enables new IP network design patterns, where user IP addressing and routing is fully isolated from the service provider IP addressing and routing through the L2 abstraction. This design principle extends also to the ZERO protocol capability of tunneling IPv6 without any overhead over IPv4 legacy infrastructure thus providing an easy migration path.

Two definitions (1 and 2) have been proposed that define criteria for Ethernet frames to be NICE or UGLY.

The following theorems have been proved.

Theorem 1: *Every branch computer requires only one Channel in each direction through ZERO tunnel to communicate with all global Internet hosts connected behind the central office* *network.*

Theorem 2*: In converged state ZERO protocol will correctly tunnel NICE frames with zero-overhead through Transport infrastructures that: (T1) do not fragment IPv4 packets with size<=1500; (T2) do not filter IPv4 packets by Source IP address; (T3) do not alter IP packet contents besides normal TTL and IP header checksum modification during forwarding.*

The core ZERO protocol *DC* and *IC* transformations have been defined in the PTL language.

The ZERO protocol extensions discussed in Section 3.4 disrupt full transparency and efficiency guarantee, but enable ZERO protocol use over un-controlled public Internet, including support for NAPT gateway traversal. The extended ZERO protocol is aimed at end-users ready to tolerate non-essential frame modification to achieve overhead-less L2 connectivity through public Internet.

The extended ZERO protocol can operate on top of service-provider core ZERO protocol – the overhead-less operation is preserved for both thanks to their reliance on modifying different (*identification/port* or FO/EB respectively) header fields.

Two ZERO protocol prototype implementations (user-land and kernel) have been demonstrated and tested both in lab and across public Internet. The test results confirm nearly zero overhead efficiency of the ZERO protocol. The kernel implementation also demonstrates 850Mbps throughput over a tunnel of 1Gbps links and low power x86 architecture tunnel gateways.

Tcpdump data from ten of the tests show that RTT between the two end stations (forth and back through the test tunnel) was in range of 500-600 microseconds. A *netpipe-tcp* test shows that TCP acceleration over the tunnel is typical for the given stream sizes.

The kernel ZERO implementation has been used as a SOHO tunneling solution in real home network for over a year. Mostly the usage has been: Web browsing, E-mail, on-demand video (youtube.com and similar), teleconferencing. One tunnel end was at the home premises, the other end – at remote site in different city. The tunneled packets traveled through more than one Internet service provider. The subjective user experience suggested no tunneling impact on Internet service stability and performance.

A new factor in real life behavior of the ZERO protocol has been proposed – the “small sync factor”. It expresses the degree at which SYN messages encapsulate frames small enough that the resulting ZERO SYN packets do not get fragmented on transport network. The factor has been observed in simulated and real life scenarios and gave interesting results: only 5,85% of ZERO SYN packets being fragmented for mixed length sessions; and no fragmentation was detected for all sessions that were shorter than 11 seconds.

The basic idea behind ZERO protocol and a simple implementation concept has been proposed by professor Guntis Barzdins. The rest of the work is presented in this chapter and is done by the author, including: redesign of the idea into a protocol that works over public Internet; the terminology of the protocol; solutions to challenges; the formal proofs of it's behavior and properties; the extensions and discussions; and so on.

4 Real time batch processing of streamed data using Lustre

Data processing in the field of radio astronomy is perhaps among the most data and throughput hungry applications in modern ICT [72][73]. While provision of dedicated hardware infrastructures for this field of data processing is popular, the author proposes to use four concepts described in this thesis for more effective radio astronomy data processing in the virtualized infrastructures: ZERO tunneling protocol; the Unified computing facility (proposed later); the conclusions from file transfer protocol performance study (described later); and the batch stream processing system proposed in this chapter.

Synchronous stream processing [25] in real time requires dedicated resources sufficient for worst case samples and their rates. [26][27] Insufficient resources even for a brief period may leave some raw data unprocessed and/or unsaved leading to corruption or loss of data. However, dedicating significant fixed resources often raises a question of their utilization effectiveness.

Asynchronous stream processing [25] allows processing of a new sample or set of samples before the previous ones have been completed and even create a backlog of samples. This makes the processing more complex but also more flexible.

The following hybrid system is proposed in this chapter: raw data is stored synchronously, while processing is done asynchronously. One crucial implementation detail has been researched and also proposed in this chapter – Lustre distributed file system as the synchronous raw storage.

4.1 System requirements

While exploring the methods and tools to store and process high volume astronomical data streams the author came up with a model and some implementation ideas for such a function. An assumption was present that astronomical data streams may be hard, expensive or impossible to recapture in case of original data loss or corruption. The raw data integrity was recognized as a priority. However, the processing model was a matter of implementation. The following is the authors' vision of how to implement an effective processing of high volume data streams while maintaining the raw data integrity.

4.2 System architecture

The author proposes a model of asynchronous real time batch processing for streamed data that would provide synchronous, lossless raw data storage and asynchronously dedicate only necessary computational resources to process the so far unprocessed data.

The solution is to save incoming raw data in a distributed high performance file system and create a backlog of unprocessed data. The backlog gets processed by a number of processes that increases if the backlog is extending and shrinks when the backlog is reducing.

The raw data can be kept in storage if it may be useful for additional analyses. Or it can be removed or rotated as soon as those get processed and the results get committed.

Another justification for an asynchronous approach comes from reasoning about the possibility to precisely predict the timing and utilization of the processing even if the parameters of a new stream are known.

Some aspects of processing may be known before it starts:

- stream volume;
- rate and size of the stream samples.

The number and capacity of the entities may be provisioned according to the known stream parameters.

Still some aspects of processing may change as it starts and even during it:

- How much time does it take to process a sample. It may vary depending on complexity of the sample.
- What technical factors will influence processing time of each sample. Some factors may be: cache effects, storage devices seek time, congestion of shared resources.

The more complex the whole system the more elements of it may influence processing predictability. The proposed model does not suffer from such uncertainty.

The three conceptual entities of the architecture are:

- File system nodes – store raw data in a clustered architecture utilizing an adjustable number of nodes. Priority – performance available to all Receivers and Processors.
- Receivers – tasks that receive the stream data and store into a common File system. Priority – lossless storing.

- Processors – tasks that retrieve stream samples from the File system, process them and commit the results ether by replacing or adding to the same File system or storing in a new one.

The following two pictures illustrate stream data paths in two scenarios (the gray boxes represent separate physical or virtual machines):

- A simple scenario when one File system node and one Processor will be enough to process the incoming stream is given in the following picture. Still, processing the data separately from the receiving and storing part is recommended for data integrity reasons.

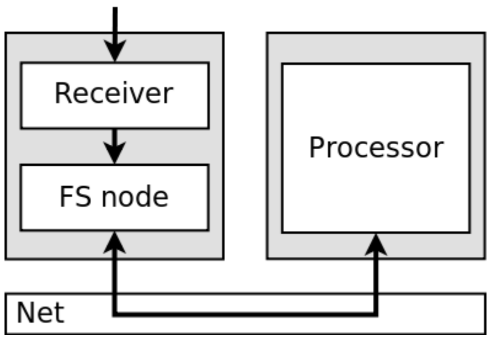

Figure 19: Simple streaming scenario

- A scalable scenario with up to N storage nodes and up to M Processors. Dynamically changing the N and M numbers is suggested. Although, dynamically changing the number of File system nodes may have some issues depending on the chosen file system technology.

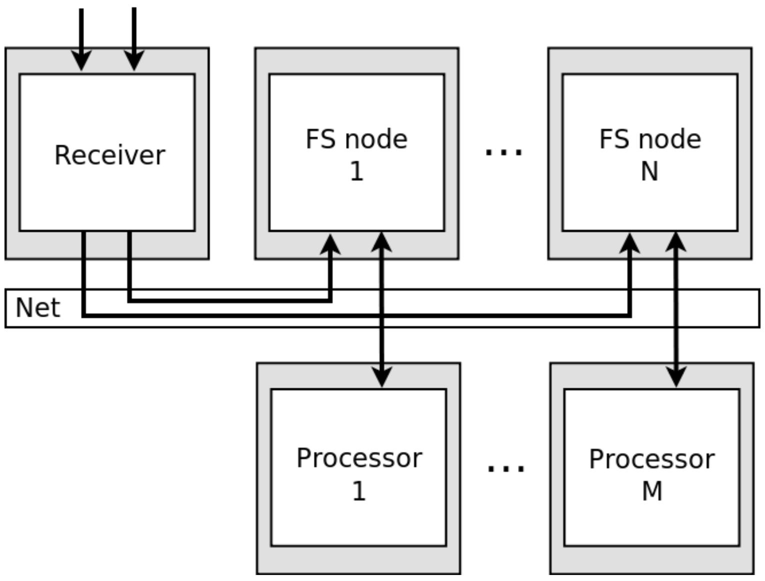

Figure 20: Scalable streaming scenario

Only one Receiver is proposed in the second scenario since the data forwarding capabilities of modern computers are quite high. Forwarding several Gbps with a 3 year old desktop computer in a software firewall is proved by the authors. Still, scaling the Receivers is also an option if the incoming stream can be split and processed in a form of multiple sub-streams.

4.3 File systems

Recognizing the data integrity priority and also the wide range of existing file systems the most popular ones were researched paying attention to the following properties: limitations, performance, file locking granularity.

Six scalable file systems available in Linux were compared. Two main categories of them are:

- Shared storage file systems – utilize concurrent access to the same underlying storage via storage controllers, often via FibreChannel to a SAN. GFS2 [28], GPFS [29] and OCFS2 [30] are considered.
- Aggregated storage file systems – utilise local storage of all the nodes providing distributed access via various networking types, often Ethernet. Ceph [31], GlusterFS [32] and Lustre [33] are considered.

4.3.1 GFS2

GFS2 is available in the RedHat Linux distribution. It uses Clustered Logical Volume Manager (CLVM) for managing distributed access and locking, which in turn depends on Red Hat Cluster Suite. High availability is ensured by means of SAN and node fail-over. While a GFS2 file system may be used outside of LVM, Red Hat supports only GFS2 file systems that are created on a CLVM logical volume .

Fail-over feature requires physical fencing, which is a drawback since all other file systems in this study can perform fail-over in software.

GFS2 provides a wide choice of lock types :

• Non-disk	mount/umount/recovery
• Meta	The superblock
• Inode	Inode metadata & data
• Iopen	Inode last closer detection
• Rgrp	Resource group metadata
• Trans	Transaction lock
• Flock	flock(2) syscall
• Quota	Quota operations
• Journal	Journal mutex

A recent study “Adventures with clustered filesystems” by Bank of Italy [34] shows that even a 16 nodes symmetric clustered file systems like GFS2 and OCFS2 show operation times much higher than those provided by the most resource consuming Lustre file system.

4.3.2 OCFS2

OCFS2 uses a distributed lock manager (DLM) which resembles the OpenVMS DLM but is much simpler.

It provides file level locking (inode), uses flock(2) . File locks taken on one node from userspace will interact with those taken on other nodes. All flock(2) options are supported, including the kernels ability to cancel a lock request when an appropriate kill signal is recieved by the user. Unfortunately, POSIX file locks, also known as lockf(3) or fcntl(2) locks are not yet supported in a cluster manner.

Fencing is implemented as the act of forcefully removing a node from a cluster. A node with OCFS2 mounted will fence itself when it realizes that it does not have quorum in a degraded cluster. It does this so that other nodes won’t be stuck trying to access its resources.

OCFS2 allows each node to read and write both meta-data and data directly to the SAN.

OCFS2 has a feature called Inline Data which makes use of OCFS2’s large inodes by storing the data of small files and directories in the inode block itself. This saves space and can have a positive impact on cold-cache directory and file operations. Data is transparently moved out to an extent when it no longer fits inside the inode block. This feature entails an on-disk change.

4.3.3 Ceph

Ceph architecture is based on the assumption that systems at the petabyte scale are inherently dynamic: large systems are inevitably built incrementally, node failures are the norm rather than the exception, and the quality and character of workloads are constantly shifting over time.

Ceph decouples data and metadata operations by eliminating file allocation tables and replacing them with generating functions. This allows Ceph to leverage the intelligence present in OSDs to distribute the complexity surrounding data access, update serialization, replication and reliability, failure detection, and recovery. Ceph utilizes a highly adaptive distributed metadata cluster architecture that dramatically improves the scalability of metadata access, and with it, the scalability of the entire system.

The Ceph file system has three main components: the client, each instance of which exposes a near-POSIX file system interface to a host or process; a cluster of OSDs, which collectively store all data and metadata; and a metadata server cluster, which manages the namespace (file names and directories) while coordinating security, consistency and coherence .

Formely the Ceph client was implemented as FUSE, but since 2010 it is included in the Linux kernel.

The smallest locking object is inode. Concurrent file read ir possible, concurrent writes – not.

4.3.4 GlusterFS

GlusterFS is a software-only, highly available, scalable, centrally managed storage pool for public and private cloud environments. [32]

GlusterFS is implemented as filesystem in userspace (FUSE). It's maintained by RedHat (like GFS) since 2011.

GlusterFS utilizes existing filesystems as the underlying data structures, e.g., XFS, EXT3/4. It aggregates them in namespaces. The communication uses client/server model and TCP/IP protocol suite.

File level locking is handled distributedly across the storage nodes using posix-locks translator.

GlusterFS supports both fcntl() and flock() calls.

4.3.5 Lustre

Fifteen of the top 30 supercomputers in the world have Lustre file systems in them, including the world's fastest supercomputer – K computer.

It has three level architecture:

- Metadata servers (MDS) – store file system metadata in metadata targets (MDT).
- Object storage servers (OSS) – store file data in object storage targets (OSTs).
- Clients – access Lustre with standard POSIX semantics.

A Fujitsu study [11] shows high Lustre performance: ~150GB/s reads, ~100GB/s writes.

Lustre has a locking feature rare in this category – file range locking. File data locks are managed by the OST on which each object of the file is striped, using byte-range extent locks. Clients can be granted both overlapping read extent locks for part or all of the file, allowing multiple concurrent readers of the same file, and/or non-overlapping write extent locks for regions of the file. This allows many Lustre clients to access a single file concurrently for both read and write, avoiding bottlenecks during file I/O. In practice, because Linux clients manage their data cache in units of pages, the clients will request locks that are always an integer multiple of the page size (4096 bytes on most clients). When a client is requesting an extent lock the OST may grant a lock for a larger extent than requested, in order to reduce the number of lock requests that the client makes. The actual size of the granted lock depends on several factors, including the number of currently-granted locks, whether there are conflicting write locks, and the number of outstanding lock requests. The granted lock is never smaller than the originally-requested extent. OST extent locks use the Lustre FID as the resource name for the lock. Since the number of extent lock servers scales with the number of OSTs in the filesystem, this also scales the aggregate locking performance of the filesystem, and of a single file if it is striped over multiple OSTs.

Overall results of the distributed filsystems study are in favor of Lustre file system:

- It does not require fencing;
- It does not require SAN infrastructure;

- It provides high availability, scalability and performance; [34][35]
- It's used in half of the TOP 30 clusters, which implies Lustre maturity and stability;
- It allows file range locking, thus multiple clients can write to different parts of the same file.

4.4 Receivers and Processors

The architecture of these entities is quite simple. Any platform that can activate and stop a receiving or processing software can be used. The author proposes a physical or virtual machine with appropriate software to be used for any Receiver or Processor. Those machines can be provisioned to scale the processing system resources.

The choice of software for Receivers and Processors is very wide. The basic functionality may be covered by a simple *netcat* utility for the Receivers and simple Processor scripts for automated task execution.

Application of MapReduce principle [36] to process large volumes of data is becoming popular in recent years. Hadoop [37] – the Apache open source implementation of this principle is widely used in production by many large IT corporations. However, this technology is well suited for processing large fixed data sets when only the total processing time is relevant. Processing unbounded streams can be better organized in a backlog queue that gets processed in a FIFO manner.

One sophisticated stream processing distribution solution is Apache S4 [38], formerly Yahoo! technology for high volume stream processing, now an open source project. The adapting Processor count idea can be implemented as a separate module or as an improvement to the S4 project. An important drawback of Apache S4 is it's partial fault-tolerance.

Although the proposed asynchronous stream processing method itself is fault-tolerant a distributed processing technology with same property may alleviate a need for additional iterations of stream walk-through in case of Processor failure. An example of such a technology is Gearman [39]. It uses a queue of jobs submitted from clients, distributes the jobs to a number of workers and communicates the results back to clients.

4.5 Conclusions on the system

A hybrid processing model for astronomical streamed data is proposed in this chapter: storing raw data is synchronous, processing is asynchronous. It honors the integrity of raw data and dedicates only the necessary volume of processing resources as the requirements change.

The choice of a critical implementation detail – file system – has been made. Lustre provides the necessary functionality and performance, yet avoids limitations of some other candidates. Choice of processing elements organization method is bound to the properties of specific processing software. Gearman is an example of lightweight but feature rich job management technology.

The proposed system can be implemented in the existing IMCS UL Scientific cloud platform as well as other cloud or clustered platforms. The research in this chapter is supported by European Union via European Regional Development Fund Project No.2010/0206/2DP/2.1.1.2.0/10/APIA/VIAA/011.

All the work in this chapter is done by the author, including research and proposed system concept, except one section – the research of the filesystems is mostly done by a fellow researcher Kaspars Krampis at the Institute of Computer Science and Mathematics, University of Latvia.

5 Unified computing facility design based on open source software

The unified computing facility design proposed here describes a modern federated IaaS cloud infrastructure in which the earlier described technologies and concepts may be utilized:

- PTL and ZERO protocol – to improve network traffic efficiency of tunneling multiple virtual networks between federated IaaS cloud installations;
- The stream processing system – to provide dynamically provisioned high throughput computing (HTC) services.

Traditionally ICT has been one of the priorities in Latvia. There are several universities with ICT programs and also research institutions that cover different research directions. For ICT research and higher education there is necessity to introduce different new ICT concept implementations. As country is small and developing it is critical to do it with very limited budget. As computing resources are planned for different types of tasks the aim is combine all resources both computing and data storage resources in unified computing facility so that it could be used for different needs.

The design of unified computing facility combines modern computing concepts. The open source software coherent operation is chosen in one integrated computing resource. The unified computing facility design and author's experience can be taken as an example of how to design e-infrastructure.

Keeping conformity with the status of IMCS UL in order to provide comprehensive understanding of technologies in study process and in order to perform research in the relevant field the proposition does not use expensive commercial solutions. [44] In lieu free packaged software bundles are used as more effective approach. The choice of open source will be substantiate more extensively.

User needs have been analyzed and the necessity of Computing concepts are being substantiated. Networking infrastructure available in IMCS UL and the applications where the chosen Open source software partly or fully ensure those concepts are being described.

Table 14. Components of the converged infrastructure

Software components of unified computing facility, arguments to be explained	**Hardware components of unified computing facility, arguments to be explained**
• OpenStack federated Cloud computing software • Identity, Authentication and Authorization Infrastructure system (with standards of Open Virtualization Format (OVF), Cloud Data Management Interface (CDMI) and Open Cloud Computing Interface (OCCI) • European Middleware Initiative (EMI) recommendations • Citrix XenServer 6.5 Enterprise hypervisor • RedHat 6 Enterprise Linux • Windows HPC Server 2008 R2 SP2 • OpenMPI (MPI-2) • CUDA GPGPU a parallel computing platform and programming model offered by NVIDIA. • GPU-accelerated MATLAB operations via MATLAB Distributed Computing Server • EGEE GRIG cluster interface to operation of GRID clusters certified according to EGEE requirements	• Blade server 4xIntel Xeon L7555, 10GE/FCoE/RoCE converged network adapters • Storage Area Network (SAN) Storage System with FATA un SATA drives • Fabric-based encryption • SAN Volume Controller with block storage virtualization appliance and SAN Disk Virtualization System • Tesla GPU expansion M2050 "Fermi" • MATLAB GPU Computing with NVIDIA CUDA-Enabled GPUs via MATLAB Distributed Computing Server • GRID clusters certified according to EGEE requirements

• Lustre 2.0 • Drupal 7.0 as social network development tool	

Aggregation of CLOUD resources for separate computing tasks will be described. Unified computing facility aggregates computer resources from farms of physical servers, storage, and network into logical resource pools. A resource pool model allows to allocate and delegate responsibility for logical resources to different separate tasks according to their resource needs. Resource shared logical pools of CPU and memory guarantee a level of resources for specific groups of users or specific computing tasks which must be executed in single-handed environment for example, GRID computation or radio astronomy data streaming. They can be flexibly added, removed, or reorganized according to business needs.

5.1 Experience with Existing IMCS UL's E-infrastructure For E-science

IMCS UL was established in 1959 (http://www.lumii.lv). Currently IMCS UL is the largest and the most relevant research institution in the field of information technology, mathematics, computer science and computer linguistics in Latvia. IMCS UL has longstanding traditions in developing and maintaining progressive e-infrastructure and providing public services in related areas. Different layers of e-infrastructure are available today to support scientific research: GÉANT network, GRID technologies and scientific field-specific e-infrastructures. Today for researchers in Latvia IMCS UL provides:

- Networking and international connectivity to GÉANT (IMCS is partner of GN3 [40] and have responsibility in Latvia for National Research and Education Network (NREN), CERT.LV, domain names .LV.
- Computing environment (GRID computing environment and National Grid Initiative (NGI), project EGI-InSPIRE [41] national scientific Cloud prototype with servers and Storage Area Network of ½ Petabyte capacity - in operation);
- e-Science functional application of e-infrastructure (CLARIN, ELIXIR [CLAR,ELIX], radio

astronomy data streaming facility from Irbene radio telescope, social network environment Barikadopedija etc.).

There are many different possibilities and platforms for cloud construction (design and installation). Open source software is chosen here. Open source software usage is evaluated by pro and contra, compared to proprietary software and competing solutions should be evaluated. Open source pro are:

1. Ready made community, software is community driven and community serving;
2. Software does not have license costs and there's no so called vendor lock-in;
3. Software is widely customizable;
4. Mostly investment is in training local staff rather than outsourcing third party.

Keeping the conformity with the education process in order to provide comprehensive understanding of technologies in study process and in order to perform research in the relevant field the author tends not to use expensive commercial solutions which are composed of proprietary software. The proposed solution for cloud computing core development is based on OpenStack software [45].

The design of unified computing facility combines modern computing concepts. It uses open source software coherent operation in one integrated computing resource. The unified computing facility design and IMCS UL experience can be taken as an example of how to design e-infrastructure as small national research computing node.

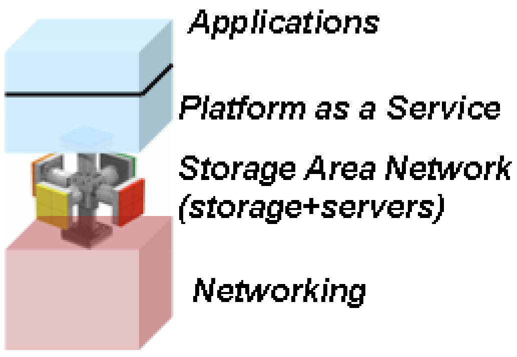

Figure 21: Three Rubik's cubes

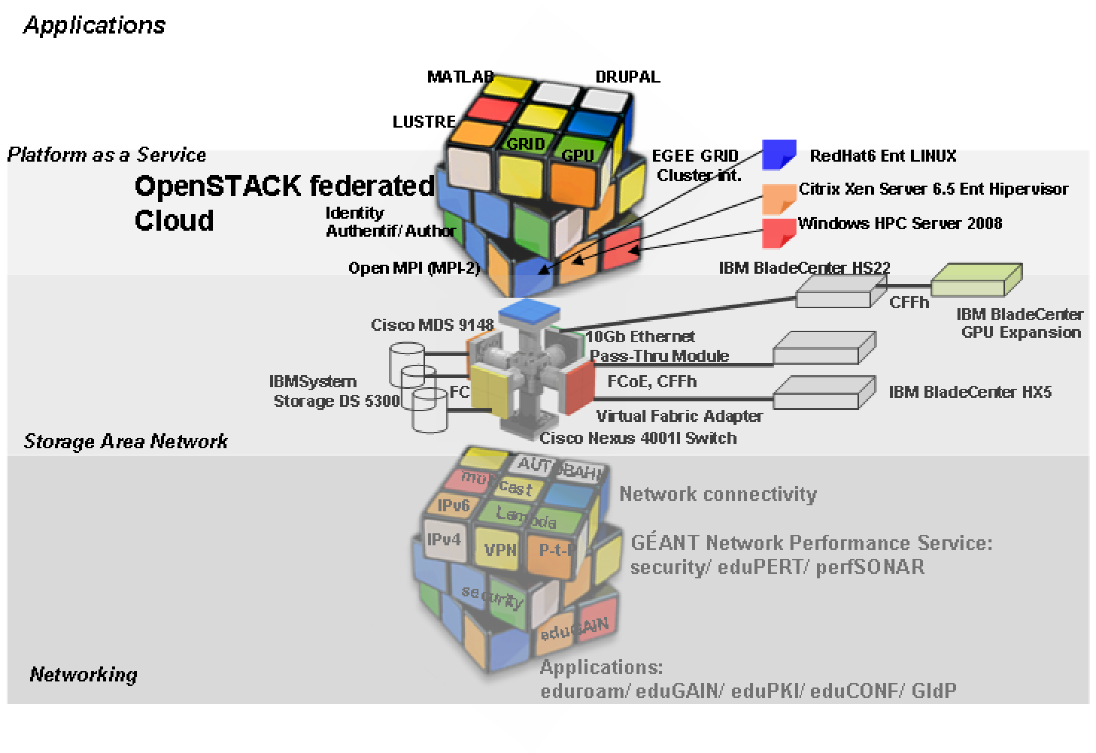

Figure 22: Unified computing facility represented as three Rubik's cubes

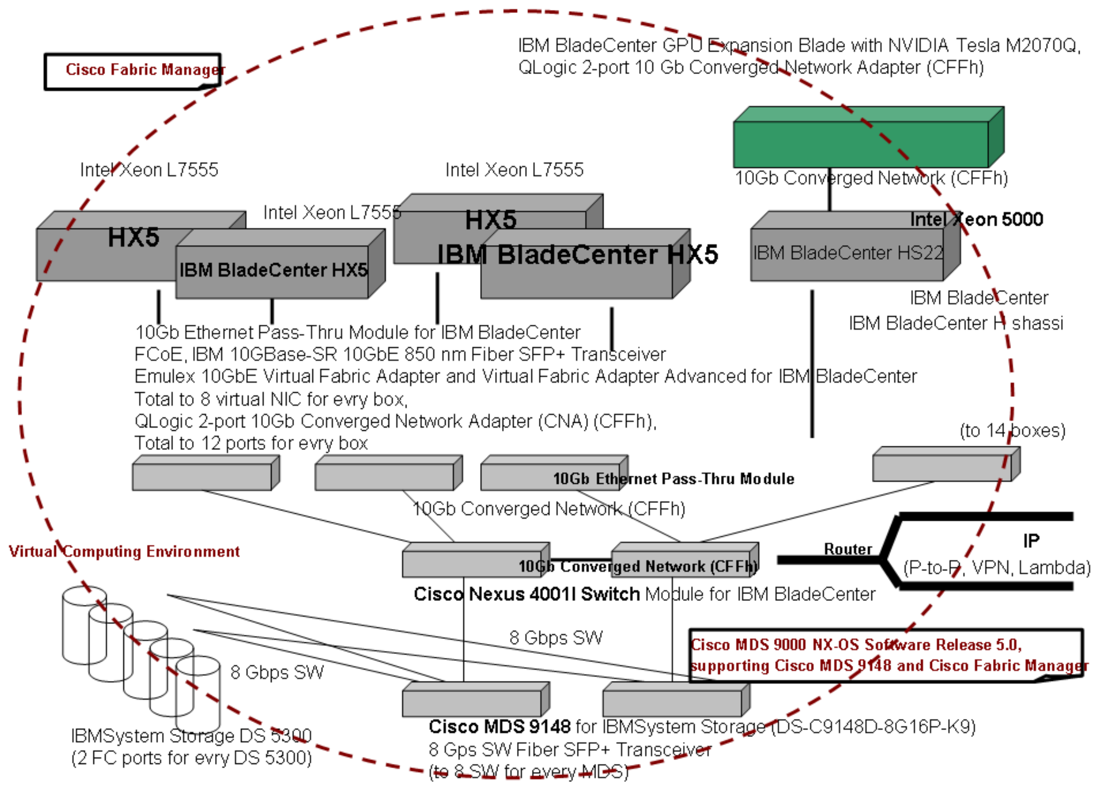
Cisco Fabric Manager
IBM BladeCenter GPU Expansion Blade with NVIDIA Tesla M2070Q,
QLogic 2-port 10 Gb Converged Network Adapter (CFFh)
Intel Xeon L7555
Intel Xeon L7555
Intel Xeon L7555
HX5
IBM BladeCenter HX5
HX5
IBM BladeCenter HX5
10Gb Converged Network (CFFh)
Intel Xeon 5000
IBM BladeCenter HS22
IBM BladeCenter
IBM BladeCenter H shassi
10Gb Ethernet Pass-Thru Module for IBM BladeCenter
FCoE, IBM 10GBase-SR 10GbE 850 nm Fiber SFP+ Transceiver
Emulex 10GbE Virtual Fabric Adapter and Virtual Fabric Adapter Advanced for IBM BladeCenter
Total to 8 virtual NIC for evry box,
QLogic 2-port 10Gb Converged Network Adapter (CNA) (CFFh),
Total to 12 ports for evry box
(to 14 boxes)
10Gb Ethernet Pass-Thru Module
10Gb Converged Network (CFFh)
Virtual Computing Environment
Router
IP
(P-to-P, VPN, Lambda)
10Gb Converged Network (CFFh)
Cisco Nexus 4001I Switch Module for IBM BladeCenter
8 Gbps SW
8 Gbps SW
Cisco MDS 9000 NX-OS Software Release 5.0, supporting Cisco MDS 9148 and Cisco Fabric Manager
IBMSystem Storage DS 5300
(2 FC ports for evry DS 5300)
Cisco MDS 9148 for IBMSystem Storage (DS-C9148D-8G16P-K9)
8 Gps SW Fiber SFP+ Transceiver
(to 8 SW for every MDS)

Figure 23: Specification of Unified computing facility

Also confining with Open source software development tools, there are many solutions for Cloud platform's development. The Cloud platform's design is compared with Rubik's cube "lego" process (Figure 21) and represent it by three cubes – two classic cubes and one "view mode" how it looks on the inside and how can one start to build a cube [46].

Unified computing facility is incorporated in European Research and Education Network GÉANT as national node - National Research and Education Network (NREN) (Figure 22). Specification of Unified computing facility is shown in Figure 23.

In Latvia science infrastructure has been developed using national public financing. The funding devoted to science is critically low both, in relative terms as a percentage of GDP and in absolute terms due to the small size of Latvian economy. Therefore, when executing future projects it is crucial to find the most favorable solution financially that would also satisfy the needs of many radically different users by ensuring wide range of services and make the most effective use of the existing infrastructure. From the beginning of 2012 a project has been started to create state research center (SRC) in ICT and signal processing that involves update of the infrastructure in five scientific institutions.

As a part of this project IMCS is going to modernize computing resources. For this reason the research of user needs, existing solutions, latest developments in technology world, usage experience of computing resources by science institutions in other countries and opportunities to integrate national resources into common European science e-infrastructure network has been done.

5.2 Existing e-infrastructure for e-science

During the last years, IMCS UL and others research institutions has developed their own e-infrastructure platforms in a computationally isolated fashion, which are not necessarily inter-operable and inter-cooperative with others for effective data portability, service and resource sharing, discovery, scheduling and integration. The inter-operable and inter-cooperative initiatives were always of a particular interest and although significant progress has been made there is still evidence of current trends keep pushing towards this direction. Specifically, the rapid developments in networking and resource integration domains have resulted in the emergence and in some instances to the maturation of various distributed and collaborative computational technologies including Web 2.0, SOA, P2P, GRID and Cloud computing. A number of relevant e-infrastructure implementations demonstrate the applicability of these technologies in a manner that enables improved intelligence in decision-making.

However, as the number of resource consumers is increasing, it has become apparent that the capacity-oriented e-infrastructures require coming together and agreeing on common behaviors for improving their quality of service (QoS), thus providing an optimization of aggregated workloads. The underlined inter-operable and inter-cooperative requirements highlight the current need for supporting a coordinated distribution of the workload between different e-infrastructures for the benefit of their Internet users. The computational vision is to continue developing inter-functionality between e-infrastructures, that is to say, forming a pool of inter-operable and inter-cooperative sub e-infrastructures that enables the dynamic collaboration of networked inter-connected organizations.

The design of unified computing facility combines modern concepts of Cloud computing, CloudVerse, OpenStack, OpenNebula, Open Fabrics, converged infrastructure and modern networking possibilities based on open source software coherent operation, as well as novel technologies like PTL and ZERO protocol – in one integrated computing resource. The unified computing facility design and experience can be used as an example in designing e-infrastructure for other small countries.

5.3 Architectural Requirements for Unified Computing Facilities

In 2014 IMCS has started project of upgrading existing cloud computing facilities to increase data storage capacity and computing performance. Based on requirements from current and potential users – researchers from different institutions, the author proposes following architectural requirements for unified computing facilities design.

5.3.1 Commodity computing and services, HPC, HTC

Commodity computing traditionally is used for large numbers of ready available computing components for parallel computing to achieve the greatest amount of useful computation at low cost. The same idea will be looked at in a more general way – not only parallel computing but any of whole range of IT functions – starting from email to device and test to supplier relationship management – can and should be thought of as commodity service. Existing IMCS computing platforms composed of commodity computing resources – GRID clusters and storage area network at this time is looked upon as a commodity service. IMCS developed traditional High Performance computing (HPC) in three directions: calculations (FLOPS), data intensive computing, and general purpose graphics processing (GPGPU). Traditional HPC my be commodity HPC, as well.

Instead of high performance computing in the project's main focus is on High-Throughput Computing (HTC) architectural concept, the use of many computing resources over long periods of time to accomplish a computational task. The main challenge a typical HTC environment faces is how to maximize the amount of resources accessible to its customers.

The key to HTC is to efficiently harness the fast access to all available resources, for example, direct access to large volume of data allocated in storage area network (FC or RDMA functions) for calculations.

Matching of scalability and performance is proposed as a general architectural requirement.

5.3.2 GPU computing

One of IMCS UL research directions is graph theory and visual information processing. Therefore it is proposed to compose unified computing facility using graphics processing unit (GPU). Many mathematical modeling tasks can be effectively solved using GPU. Based on proposition of GPU as an architectural requirement for unified computing facilities for different tasks including image analysis and general purpose scientific and engineering computing as well.

5.3.3 Data intensive computing

As scientific applications become more data intensive, the management of data resources and data-flow between the storage and compute resources is becoming the main bottleneck. Analyzing, visualizing, and disseminating these large data sets has become a major challenge and data intensive computing is now considered as the "fourth paradigm" in scientific discovery after empirical, theoretical, and computational scientific approaches.

Data-intensive computing is a class of parallel computing applications which uses data parallel approach to processing large volumes of data (typically terabytes or petabytes in size) allocated in storage area network.

5.3.4 Shared disk file systems

Storage area network with possibility of allocating large data volumes and the transfer of data at a high-speed rate in multi-usage environment is needed to support such applications:

1. Real-time analytic processing and a stream-computing approach with the emphasis on the use of Irbene radio telescope accelerators. Its viability for managing and ensuring interoperability and integrity of signal processing data pipelines is necessary for this radio astronomy tasks.
2. High-definition television and multimedia.
3. Continuous backup to a storage medium of the data flow within computer systems

5.3.5 Urgent computing

Urgent computing is a new and evolving field made possible by the improved fidelity and utility of high-performance computing to decision making. It refers to the concept of providing prioritized and immediate access to supercomputers and GRID for emergency computations, for example, IMCS Computer Emergency Response Team (CERT) may need processing of extremely large volume of data (e.g. log files) during network attacks or during other matters of immediate concern. Applications that provide decision makers with information during critical emergencies cannot waste time waiting in the job queues and need access to computational resources immediately.

5.3.6 Social networks

Currently in institute there are two specialized social networks: for gathering and saving IMCS IT history from current and preceding employees. The other network is for documenting the history of renewal of independence of Latvia based on partaker memories. A social network hosting service is based on unified computing facility.

Unified computing facility must consist of free software kit for specialized social network engines and social network analysis software establishing and maintenance. The choice is Drupal. [47]

5.3.7 Virtualization

It is one of the key technologies in the unified computing facility architecture since it allows functionally separated objects to share physical resources of their class. Some key elements of the infrastructure to extensively use virtualization are: computer resources, networking, software platforms and software instances.

5.3.8 Cloud

OpenStack project emerged as a cloud management software with Amazon Web Services compatible API. Since 2012 it has grown into a complex of projects that provide management for multiple clouds and different service types. Some of these projects are core components of the Unified computing facility:

- OpenStack Compute and Image Service are at the core of the proposed Infrastructure as a Service component;
- OpenStack Keystone identity service provides unified and federated authentication;
- OpenStack Metal as a Service (MaaS) service allows to automatically provision physical server resources for dedicated computing applications like HTC or GPGPU.

Other OpenStack sub-projects may be used or extended for additional functionality.

5.3.9 Open source software usage

According to IMCS UL values to deeply understand the core idea of a technology, during studies and research, together with students the staff tend to use open source software as much as possible instead of proprietary ready to use commercial solutions. [44]

Since practically all industry standards for data processing an transfer are well implemented as Open Source software IMCS heavily uses these throughout the Unified computing facility.

5.4 Requirements for networking infrastructure

IMCS UL Unified computing facility is to be incorporated in European Research and Education Network GÉANT as national node - National Research and Education Network (NREN). For operation IMCS is using following GÉANT network services.

5.4.1 GÉANT IPv4 and IPv6 connectivity

The GÉANT network provides transit to all IPv4 and IPv6 traffic to and from connected international academic partners and to the Internet. Currently Latvian access to this network is limited to 2.5 Gbps bandwidth. To improve BalticGrid and HTC cooperation in Europa the Latvian connectivity needs to be updated to 10 Gbps.

5.4.2 Dedicated point to point connectivity to *GÉANT* partners

Planned usage of this service is in specialized radio astronomy network for data streaming from Irbene radio telescope.

5.4.3 GÉANT Lambda

A GÉANT Lambda is presented to the NREN as a transparent wavelength on which they can then develop their own higher-level network layers or point to point connectivity. It is planned to carry out this service within Baltic Ring project in future.

5.4.4 Bandwidth on demand

The Automated Bandwidth Allocation across Heterogeneous Networks (AutoBAHN) GÉANT system has been designed to allocate network bandwidth to users/applications both immediately and in advance. Networking resources in the form of dynamic circuits are allocated, end to end, across multiple domains. The granularity of resource reservations in terms of bandwidth and duration is important, together with the required Quality of Service (QoS) parameters. Planned usage of this service is in specialized radio astronomy network for data streaming from Irbene radio telescope.

5.4.5 A virtual private network (VPN)

A virtual tunnel service securely linking two academic institutions sites that use third party IP infrastructure is a part of the Unified computing infrastructure. Such a service is already used in the the existing IMCS cloud system. The non-fragmenting ZERO tunneling protocol developed at IMCS will be tested for efficiency in VPN services on the Unified computing infrastructure.

5.4.6 Multicast

Multicast provides efficient delivery of data traffic in one–to-many and many-to-many scenarios. Currently the use of this service in Latvia is infrequent.

5.4.7 Networking security

IMCS UL have national responsibility for data and networking security monitoring in the country. CERT team has been collaborating in the area of security with Trans-European Research and Education Networking Association (TERENA) task force TF-CSIRT. [48] Internet Service Providers are using structured format for the exchange of computer incident information. Such format is useful in speeding up the exchange of information internationally and helps to avoid misunderstandings. Latvia supports the application for CERT system the Incident Object Description Exchange Format (IODEF), traffic filters on routers, stateless/stateful packet filtering, IDS/IPS (intrusion detection and prevention systems), e-mail greylisting, DNSBLs (Domain Name System Blacklists) to protect certain domains.

5.4.8 Videoconferencing based on GÉANT Web Conference Service

At this time different organizations and NRENs have different standards of service level, making it technically and administratively difficult for users to locate and make use of existing videoconferencing facilities to collaborate with their colleagues. Such complexity of creating and integrating videoconferencing services into new or existing pan-European systems requires a detailed investigation of the service requirements, which should map to real-life user and service demands.

5.4.9 Roaming, authorization/ authentication

There are many different user Identification and Authentication and Authorization Infrastructure systems in use across Europe, all of which are designed to control access to networks and applications and computing resources, and ensure the secure movement of information within a network. It is currently necessary for organizations to join one another's federation in order to establish the relationship necessary to exchange identity information across networking, GRID and Cloud federated systems. IMCS proposes to deploy such interface combining OpenStack Keystone service and existing LDAP directories of scientific institutions. For standardization, interoperability of Cloud, GRID, GÉANT systems and Identity (authorization/ authentication), accounting and resources description data exchange IMCS is using European Middle-ware Initiative (EMI) recommendations together with standards of Open Virtualization Format (OVF), Cloud Data Management Interface (CDMI) and Open Cloud Computing Interface (OCCI). [49]

5.5 Software as a Service Level

Software as a service level will provide users with the following software bundles:

- OpenMPI (MPI-2) libraries;
- CUDA GPGPU a parallel computing platform and programming model;
- GPU-accelerated MATLAB operations via MATLAB Distributed Computing Server;

- EGEE GRIG cluster interface to operation of GRID clusters certified according to EGEE requirements;
- ANSYS engineering simulation software package;
- Drupal as social network development tool.

5.6 Platform as a Service Architecture Level

The following platforms will be available for users' software and solutions:

- Hadoop distributed computing platform employing Map-Reduce programming principle;
- Apache Web server with several application platforms: Django, etc.;
- PostgreSQL and MySQL database instances;
- Highly scalable file storage platform with flexible user quotas;

5.7 Hardware as a Service Architecture Level

Unified computing facility unified fabric interconnect architecture with Virtual link, VM direct path to network interface card as converged infrastructure, converged storage is a storage architecture that combines storage and compute into a single entity. This can result in the development of platforms for server centric, storage centric or hybrid workloads where applications and data come together to improve application performance and delivery dramatically simplifying the data center network, enabling any-to-any connectivity.

The combination of storage and compute differs to the traditional IT model in which computation and storage take place in separate computer equipment. The traditional centralized SAN model can become a bottleneck as data sets gets bigger and the time to access that data gets shorter. This refers to a consolidated high-performance computing system consisting of loosely coupled storage, networking and parallel processing functions linked by high bandwidth interconnects (such as 10 Gigabit Ethernet, Fibre Channel over Ethernet (FCoE) and InfiniBand).

Using unified orchestration tools hardware will also be available as timed service to those demanding highest dedicated performance.

5.8 Cloud Aggregation

Unified computing facility aggregates computational resources from so called farms of physical servers, storage, and network into logical resource pools. A resource pool model allows to allocate and delegate responsibility for logical resources to different tasks according to their needs. Resource shared logical pools of CPU and memory guarantee a level of resources for specific groups of users or specific computing tasks which must be executed in a single - handed environment for example GRID or radio-astronomy data streaming. They can be flexibly added, removed, or reorganized as needed.

In addition, by supporting unified fabric provides both the LAN and SAN connectivity for all nodes within its domain. Typically deployed in redundant pairs, fabric interconnects provide uniform access to both networks and storage, eliminating the barriers to deploying a fully virtualized environment.

5.9 National Cloud Federation, Regional Cloud Exchange

IMCS proposes federation of multi-cloud environment by Regional Cloud Exchange for integrated use of regional cloud computing facilities (each separate computer resource is defined as cloud cluster). Similar tasks perform regional Internet Exchange Point (IXP) for data flows between different Internet Service Providers and distribution of tasks within GRID network. Cloud interoperability is carried out using OpenStack toolkit.

The task for Cloud Exchange is to integrate several regional computer resources into one regional national Cloud (Cloud as RPF: Regional Cloud Exchange). Regional Cloud Exchange is a hybrid cloud consisting of Cloud clusters with possible deployment models (public cloud, private cloud, community cloud).

Scope of technology activities for Regional Cloud Exchange and organizational and institutional activities for Regional cloud cluster (see below) are: security, customer support, Peering Agreements, Service Level Agreements (SLA's), innovative technologies, interoperability, authentication and authorization policy, user (client and cloud community member) authentication/authorization (e.g. TERENA), quality and risks management, load balancing between clusters, accounting of use of Cloud cluster resources.

Important functions furnished by Regional Cloud Exchange technology are following:

- Ensure integrated use of cloud community cluster, rendering service required by clients or Cloud cluster (establish a task package for one or more Cloud clusters observing defined access policy);
- Since community Cloud clusters are connected with optical cable communication between cloud clusters Regional Cloud Exchange dynamically ensures establishing either virtual communication channel between necessary Cloud clusters or engaging Cloud cluster agents work for use of communication channel.

5.10 Regional Cloud Cluster Community as Non Profit Organization

A Cloud cluster is institutionally represented by academic institution or commercial entity. IMCS has been searching for an appropriate organizational form for their integrated operation. In order to formulate common opinion and coordinate steps to be taken usually conferences are being convened, working groups and steering committees are formed which in later stages often transform into associations. In the field of Cloud Computing Association [50] and Asia Cloud Computing Association [51] are known. For understanding of regional cloud cluster operation IMCS has analyzed IXP's institutional model [52]. To ensure regional cloud exchange and to frame regional cloud cluster community policy an institutional model is chosen in the form of non profit geographical horizontal business cloud cluster [53]. Cloud cluster members are research institutions, high schools and commercial entities who provide Cloud services. Cloud community have ownership and management rights; staff initially comes from academic institutions. The important goal of Cloud cluster is to attract new users and achieve international usage in the amount of 30%.

The neutrality of a Cloud cluster is important factor to its success. At first step organizational (Cloud cluster members') neutrality is considered, but later Cloud cluster members can decide on the issue of neutrality in more details: to be carrier neutral or services collocation neutral or possibly both.

5.11 Latvian Cloud as a Regional Partner Facility in European Union

The ESFRI strategy 2010 defines role of regional partner in European Research Infrastructure Consortium (ERIC) [54,55]. Strategic choice between development of national Cloud Computing and High Performance Computing centers or usage of pan-European community cloud infrastructure (e.g., Luxemburg or Amsterdam) raises the question of global competition.

Latvia will develop research infrastructures in respect to ESFRI road-map to have a node of distributed infrastructures or to build a regional partner facility in the case of single sited research infrastructures. Many EU states maintain policy similar to ERI development – Czech Republic do it in the level of general policy [56] or there is also example of Eastern Mediterranean country project LinkSCEEM [57] and HP-SEE [58] for HPC development. In the same way policy is implemented in order to develop Latvian Cloud as RPF in EU. RPF sets requirements of EU level excellence e.g., claims that at least 30% of RPF’s usage is international. In order to maintain appropriate status and be attractive and competitive within EU resources should be sufficiently large; services should be of good quality and innovative. Actions were taken to accredit Latvian Cloud in the status of RPF in 2012-2013.

5.12 Conclusions on the proposed Unified computing facility design

This chapter described scientific e-infrastructure development in Latvia and migration to national Cloud as regional partner facility (RPF) in European Union (EU). Multiple public and private Computing Clouds in Latvia are in operation and it's not clear yet how to integrate these resources as one RPF and how to design one unified computing facility that is used for many different applications. The author offers their solution at Cloud software as a Service (CaaS) and Hardware as a Service (HaaS) level which is based on open source software packaged bundles.

Conceptual ideas and research of governmental regulations described in this chapter come from leading researcher, Dr.sc.comp., Rihards Balodis and researcher, Mg.mat., Inara Opmane of the IMCS UL. The rest of this chapter is done by the author, including: the cloud and networking design and services, OpenStack applications, integration of other technologies proposed by author, etc.

6 File Transfer Protocol Performance Study

The described study provides the experimental results and the analysis for selection of the file transfer protocol to be used in the upcoming Meteosat Third Generation Programme requiring sustained dissemination data rate in the range of 300-400Mbps over heterogeneous networks. This dissemination speed cannot be easily achieved with default TCP protocol settings and file transfer applications with significant round trip time (RTT) and packet loss typical to large heterogeneous networks. The designed test lab allowed finding the optimal TCP protocol and file transfer application settings reaching the target data rate at 70ms RTT and 10^{-6} packet loss, typical to terrestrial networks. Meanwhile, none of the surveyed applications were able to reach the target data rate at 700ms RTT, typical to satellite distribution networks.

In January 2010, the European meteorological union (EUMETSAT) commissioned IMCS to perform a detailed study on currently available open standards file transfer protocols for TCP/IP networks. The purpose of the study was to provide the background experimental material for the selection of a file transfer architecture in the upcoming next generation Meteorological weather satellite system to be launched in 2014 – EUMETSAT.

The results obtained in this study could be of interest to much wider audience, as there are much ungrounded myths about the performance of underlying TCP protocol and data transfer applications built on top of it. In this chapter the author provides a condensed version of the original technical report. The author has done most of the research, tasks and analyses described here.

The purpose of the study was to perform a multi-dimensional survey of four file transfer protocols (FTP, UFTP, bbFTP, GridFTP, RSYNC) under widely varying conditions characteristic to various network conditions. Namely, performance 70ms and 700ms RTT characteristic to intercontinental terrestrial Internet and geostationary satellite communications were studied. Additionally, various packet loss patterns were examined.

The measurements were conducted in controlled laboratory environment, which was meticulously fine-tuned and validated to ensure that the lab setup itself could not be the cause of negative artifacts during measurements. The lab itself was built from open-source components rather than from closed commercial network emulators. This enabled full tunability of the network simulator performance characteristics and parameters (e.g. insertion of various packet loss patterns: random packet loss, packet loss in random bursts, etc.) – the research was not limited by the constraints of the given test platform.

6.1. Test Lab Description

As depicted in Figure 24, a single test bed (two identical test bed sets were used during this study) consisted of a file transfer server connected via LAN switch with one of the clients and a network simulator. Second client was placed behind a network simulator. Switch port connected to the server was mirrored and all traffic originated from or sent to the file transfer server was copied to the traffic monitoring server. In case of unicast file transfer scenarios, data was sent between the server and client behind the network simulator. For multicast scenarios, data was sent from server to both clients. All machines had at least dual 1GbE NICs, and each machine had a separate interface used for management purposes only.

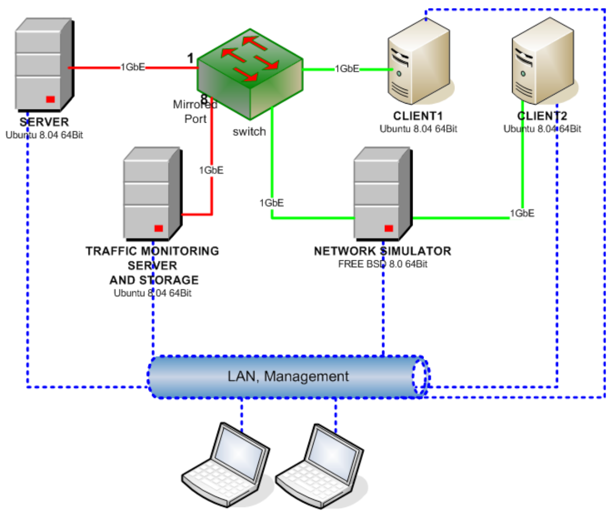

Figure 24. Test lab topology

6.1.1 Hardware

All servers used within the test bed achieved or surpassed the necessary performance levels to ensure that results obtained in the study were not biased due to performance bottlenecks in equipment used.

Network simulator had AMD Opteron 148, 2.2 GHz single core CPU, 2GB RAM, dual

Broadcom NetXtreme 1GbE network interfaces (BCM5704), 160GB SAMSUNG HD160JJ WU100-33 HDD.

File transfer server, clients and traffic monitoring server had two AMD Opteron 275, 2.2 GHz, dual core CPUs, 8GB RAM, dual Broadcom Tigon3 1GbE network interfaces (BCM95704A7), 80GB WDC WD800JD-00LSA0 HDD.

A small but capable HP ProCurve Switch 1800-8G was used for LAN connectivity.

6.1.2 Software

Network simulator operating system was FreeBSD 8.0-RELEASE. Network simulator software: 'ipfw' and 'dummynet' subsystems in the default system kernel.

File transfer server, clients and traffic monitoring server had Ubuntu Linux 8.04.4 LTS, 64-bit operating system. Usage of 64-bit kernel was essential for optimal memory addressing and necessary for large TCP buffers. File transfer applications: ProFTPd 1.3.1, vsftpd 2.0.6, bbFTP 3.2, GridFTP 4.2.1, UFTP 2.10.3, RSYNC 2.6.9. Traffic logging software: 'tcpdump', default version provided with distribution.

6.1.3 Network tuning

After the default server installation, TCP parameters for all machines were tuned for better TCP throughput. The following system configuration variables were tuned in all Ubuntu servers in accordance with best current practice [66]:

```
# Enable advanced Linux TCP features

sysctl net.ipv4.tcp_window_scaling=1

sysctl net.ipv4.tcp_timestamps=1

sysctl net.ipv4.tcp_sack=1

sysctl net.ipv4.tcp_moderate_rcvbuf=1

sysctl net.ipv4.tcp_syncookies=0

sysctl net.ipv4.tcp_no_metrics_save=1

sysctl net.ipv4.tcp_ecn=1

sysctl net.ipv4.tcp_adv_win_scale=7
```

```
# Increase Linux TCP buffers
sysctl net.core.rmem_max=16777216
sysctl net.core.wmem_max=16777216
sysctl net.ipv4.tcp_rmem="4096 16000000 180000000"
sysctl net.ipv4.tcp_wmem="4096 16000000 180000000"

# Increase network interface egress queue length
ifconfig eth1 txqueuelen 10000
```

After initial server distribution installation on the network simulator, FreeBSD 8.0 kernel was recompiled with the following kernel configuration modifications to enable Dummynet network simulator functionality, as well as to increase kernel time resolution to 40000Hz for more precise and consistent RTT simulation:

```
options IPFIREWALL
options IPFIREWALL_VERBOSE
options IPFIREWALL_VERBOSE_LIMIT
options IPFIREWALL_DEFAULT_TO_ACCEPT
options DUMMYNET
options HZ=40000
```

After kernel recompilation, both network interfaces were configured in single virtual bridge, so that traffic was transparently passed through FreeBSD network simulator between Ubuntu server and client machines on both interfaces.

6.1.4 Test bed validation

In order to validate test bed host capability to execute all test cases and produce correct measurements for scenarios specified in the study, several baseline performance measurements were performed. Initially, raw TCP and UDP throughput was measured for test bed hosts connected in a back-to-back configuration. After initial measurements, another set of test runs was performed with addition of switch between test bed hosts. Also, baseline RTT measurements for back-to-back and switched cases were performed for later comparison with test bed configuration accommodating network simulator.

Rapid sending of 100,000 ICMP ECHO requests (further – 'ping flood') was used for measuring consistency of introduced RTT at network simulator. Iperf TCP and UDP tests were carried out to measure raw TCP and UDP throughput. These tests were run for one hour. RTT measurements were performed for first 100,000 ICMP ECHO (ping) packets.

Back-to-back, test bed hosts achieved raw TCP throughput of 941Mbps and raw UDP throughput of 957Mbps as shown in the following iperf outputs:

```
$ iperf -c 192.168.1.4 -t 3600
------------------------------------------------------------
Client connecting to 192.168.1.4, TCP port 5001
TCP window size: 15.3 MByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 46584 connected with 192.168.1.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec    395 GBytes    941 Mbits/sec

$ iperf -c 192.168.1.4 -b 2000000000 -t 3600
------------------------------------------------------------
Client connecting to 192.168.1.4, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   122 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 37178 connected with 192.168.1.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec  401.1 GBytes    957 Mbits/sec
[  3] Sent 292948732 datagrams
[  3] Server Report:
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-3600.0 sec  401.1 GBytes    957 Mbits/sec  0.032 ms    0/292948732 (0%)
```

RTT of 73μs – 7.8ms (avg. 76μs) in a back-to-back configuration was measured as shown in the following ping output:

```
$ sudo ping -f -i 0.01 -c 100000 192.168.1.4
PING 192.168.1.4 (192.168.1.4) 56(84) bytes of data.

--- 192.168.1.4 ping statistics ---
100000 packets transmitted, 100000 received, 0% packet loss, time 999990ms
rtt min/avg/max/mdev = 0.073/0.076/7.840/0.026 ms, ipg/ewma 10.000/0.076 ms
```

The highest RTT was observed for the first packet, and was caused by necessity to perform the ARP request. The standard deviation of 26μs shows reasonably timed and predictable network interface operation.

Addition of a switch between test bed hosts did not have any significant effect on achievable TCP/UDP throughput or RTT. Switched performance was exactly the same as in case of back-to-back configuration – 941Mbps TCP and 957Mbps UDP as shown in the following iperf outputs:

```
$ iperf -c 192.168.1.4 -t 3600
------------------------------------------------------------
Client connecting to 192.168.1.4, TCP port 5001
TCP window size: 15.3 MByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 46584 connected with 192.168.1.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec    395 GBytes    941 Mbits/sec

$ iperf -c 192.168.1.4 -b 2000000000 -t 3600
------------------------------------------------------------
Client connecting to 192.168.1.4, UDP port 5001
```

```
Sending 1470 byte datagrams
UDP buffer size:   122 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 37178 connected with 192.168.1.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec  401.1 GBytes    957 Mbits/sec
[  3] Sent 292952698 datagrams
[  3] Server Report:
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-3600.0 sec  401.1 GBytes    957 Mbits/sec  0.032 ms    0/292952698 (0%)
```

RTT was a little lower – 61µs-103µs (avg. 66µs) with a low standard deviation of 11µs as shown in the following ping output:

```
$ sudo ping -f -c 100000 192.168.0.3
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
--- 192.168.0.3 ping statistics ---
100000 packets transmitted, 100000 received, 0% packet loss, time 8440ms
rtt min/avg/max/mdev = 0.061/0.066/0.103/0.011 ms, ipg/ewma 0.084/0.063 ms
```

All measurements show that in both back-to-back and switched modes hosts are capable of data transfer at close to 1Gbps speed. Addition of a switch between test bed hosts does not have any significant effect on achievable TCP/UDP throughput or RTT. Thus, the test bed host validation can be considered as successful and completed.

In order to validate test bed network simulator capability to execute all test cases and produce correct measurements for scenarios specified in the study, several baseline performance measurements have to be performed. Initially, raw TCP and UDP throughput must be measured for test bed hosts connected through a network simulator with pre-configured zero packet loss rate and no additionally introduced RTT. Also, RTT measurements for this setup must be made for comparison with back-to-back and switched configurations. Afterwards, consistency and reliability of introduced RTT and packet loss rate has to be validated. RTT consistency is validated by running

RTT measurements at various pre-configured RTT values on the network simulator, and observing RTT deviation from the pre-configured value. Packet loss rate is validated by running UDP throughput tests at pre-configured RTT and packet loss rate settings, and observing statistics about how many packets were lost during UDP data transfer.

Eventually, this ensures that all test bed network simulator is 1Gbps transparent and RTT introduction and packet loss introduction functionality is working correctly. Any performance bottlenecks identified during the study would thus be linked to file transfer applications used, not the testbed hosts or network simulator.

At no packet loss and no introduced delay the raw TCP and UDP performance was again 941Mbps and 957Mbps, respectively, as shown in the following iperf outputs:

```
$ iperf -c 192.168.1.4 -t 3600
------------------------------------------------------------
Client connecting to 192.168.1.4, TCP port 5001
TCP window size: 15.3 MByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 51112 connected with 192.168.1.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.1 sec    395 GBytes    941 Mbits/sec

$ iperf -c 192.168.1.4 -b 2000000000 -t 3600
------------------------------------------------------------
Client connecting to 192.168.1.4, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   122 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 37178 connected with 192.168.1.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec  401.1 GBytes    957 Mbits/sec
[  3] Sent 292949070 datagrams
[  3] Server Report:
```

```
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-3600.0 sec  401.1 GBytes    957 Mbits/sec  0.032 ms    0/292949070 (0%)
```

The RTT slightly increased to the range of 175-411µs (avg. 330µs) with a standard deviation of 44µs as shown in the following ping output:

```
$ sudo ping -f -i 0.01 -c 100000 192.168.1.4
PING 192.168.1.4 (192.168.1.4) 56(84) bytes of data.
--- 192.168.1.4 ping statistics ---
100000 packets transmitted, 100000 received, 0% packet loss, time 999993ms
rtt min/avg/max/mdev = 0.175/0.330/0.411/0.044 ms, ipg/ewma 10.000/0.326 ms
```

These numbers demonstrated network simulator performance as stable and low impact on packet flow [67].

Results from measurements of RTT consistency at pre-configured RTT of 70ms and 700ms showed that maximum observed deviation from specified RTT was 1.9ms, average deviation from specified RTT was 0.58ms at 70ms RTT and 1.2ms at 700ms RTT as shown in the following ping outputs (for 70ms RTT and 700ms RTT, respectively):

```
$ sudo ping -f -i 0.01 -c 100000 192.168.1.4
PING 192.168.1.4 (192.168.1.4) 56(84) bytes of data.
--- 192.168.1.4 ping statistics ---
100000 packets transmitted, 100000 received, 0% packet loss, time 1722123ms
rtt min/avg/max/mdev = 68.093/69.614/70.751/0.582 ms, pipe 7, ipg/ewma 17.221/69.867
ms
```

```
$ sudo ping -f -i 0.01 -c 100000 192.168.1.4
PING 192.168.1.4 (192.168.1.4) 56(84) bytes of data.
--- 192.168.1.4 ping statistics ---
100000 packets transmitted, 100000 received, 0% packet loss, time 1198654ms
rtt   min/avg/max/mdev   =   698.510/700.310/701.041/1.204   ms,   pipe   68,   ipg/ewma
```

```
11.986/700.340 ms
```

Since the RTT deviations fall well below 1% of the measured values, network simulator RTT simulation could be considered as consistent.

Iperf UDP data transfer was used to measure actual packet loss introduced by the network simulator. Note that, since the Iperf is not able to precisely throttle UDP data transfer at speeds close to maximum 1Gbps, the last stably throttable speed of 900Mbps was determined and specified to Iperf in order not to cause occasionally added packet loss due to packets being sent out of the interface at higher speed than supported by NICs. Results from measurements of packet loss rate consistency at pre-configured RTT of 0ms and 70ms show that observer packet loss rate from the UDP data transfers correspond to the pre-configured packet loss rate as shown in the following iperf outputs.

UDP data transfer, RTT 0ms, packet loss rate 10^{-3}:

```
$ iperf -c 192.168.1.4 -b 900000000 -t 3600
WARNING: option -b implies udp testing
------------------------------------------------------------
Client connecting to 192.168.1.4, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   122 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 54214 connected with 192.168.1.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec    379 GBytes    905 Mbits/sec
[  3] Sent 276917981 datagrams
[  3] Server Report:
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-3603.4 sec    379 GBytes   903 Mbits/sec  0.015 ms 278346/276917981 (0.1%)
```

UDP data transfer, RTT 0ms, packet loss rate 10^{-6}:

```
$ iperf -c 192.168.0.3 -b 900000000 -t 3600
WARNING: option -b implies udp testing
```

```
------------------------------------------------------------
Client connecting to 192.168.0.3, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   122 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.1 port 49390 connected with 192.168.0.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec    379 GBytes    905 Mbits/sec
[  3] Sent 276922048 datagrams
[  3] Server Report:
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-3602.5 sec    379 GBytes   904 Mbits/sec 0.012 ms 271/276922048 (9.8e-05%)
```

UDP data transfer, RTT 70ms, packet loss rate 10^{-3}:

```
$ iperf -c 192.168.1.4 -b 900000000 -t 3600
WARNING: option -b implies udp testing
------------------------------------------------------------
Client connecting to 192.168.1.4, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   122 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 43191 connected with 192.168.1.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec    379 GBytes    905 Mbits/sec
[  3] Sent 276918941 datagrams
[  3] Server Report:
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-3608.0 sec    379 GBytes    902 Mbits/sec 0.014 ms 278421/276918941 (0.1%)
```

UDP data transfer, RTT 70ms, packet loss rate 10^{-6}:

```
$ iperf -c 192.168.0.3 -b 900000000 -t 3600
```

```
WARNING: option -b implies udp testing
------------------------------------------------------------
Client connecting to 192.168.0.3, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   122 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.1 port 46685 connected with 192.168.0.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec    379 GBytes    905 Mbits/sec
[  3] Sent 276922439 datagrams
[  3] Server Report:
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-3601.8 sec    379 GBytes    904 Mbits/sec 0.012 ms 288/276922439 (0.0001%)
```

As it can be seen from the obtained results, network simulator produces consistent RTT and packet loss according to pre-configured value. Thus, network simulator packet loss simulation can be considered as validated and consistent.

6.2. Testing Methodology

To understand and demonstrate the practical limitations of selected applications and protocols, a plan of test scenarios was created. The scenarios fall into six categories:

1. Small unicast;
2. Medium unicast;
3. Large unicast;
4. Mixed unicast;
5. Mixed multicast;
6. Large multicast.

All unicast tests will be performed on the following unicast application pairs.

Table 15. Unicast application pairs

Application ID	Sender	Receiver
ftp	ProFTPd	ARPANET FTP
uftp	UFTP	UFTPD
bbftp	bbFTP	bbFTP
gridftp	GridFTP	GridFTP
rsync	rsync	rsync

Unicast scenarios are comprised of four categories. First three categories use a common file size each (10kB, 5MB, 2GB) and all combinations of other variables. The fourth category uses a mix of all file sizes with all RTT variants but without packet loss. The mixed file set was created to include more smaller files and less larger ones to lessen the overwhelming percentage of time spent on the larger files. The mixed file set structure is given in the following table.

Table 16. Mixed file set structure

File size, kB	Number of files	Total size, kB
10 (10kB)	320	3200 (3,2MB)
512 (0,5MB)	160	81920 (80MB)
5120 (5MB)	16	81920 (80MB)
51200 (50MB)	8	409600 (400MB)
2097152 (2GB)	1	2097152 (2GB)

The 5th category used the mix of all file sizes with all RTT variants but with only two packet loss rates (10^{-6}, 10^{-3}). The 6th category used just 2GB large files and the worst packet loss (10^{-3}). This last category was added to demonstrate what performance a multicast application like UFTP can achieve on high latency links.

The performance of applications was measured on the line between the sender and switch. All the tests were run for one hour, except test 19 which was run for 5 hours and scenario 24 was run for two hours.

For multicast applications there were two receivers. One of them was connected to the server through a switch and the other was connected to the switch through the network simulator. That way one of the receivers was using high performance switched Ethernet path to the sender while the other was set to use a path with delayed and dropped packets through the network simulator. Again, the performance of applications was measured on the line between the sender and the switch.

Table 17. All test scenarios

Scenarios	*Category: file sizes*	*RTT*	*Packet loss rate*
1,2,3,4,5,6	Cat 1: 10kB	1,2,3: 70ms 4,5,6: 700ms	1,4: 0 2,5: 10^{-6} 3,6: 10^{-3}
7,8,9,10,11,12	Cat 2: 5MB	7,8,9: 70ms 10,11,12: 700ms	7,10: 0 8,11: 10^{-6} 9,12: 10^{-3}
13,14,15,16,17,18	Cat 3: 2GB	13,14,15: 70ms 16,17,18: 700ms	13,16: 0 14,17: 10^{-6} 15,18: 10^{-3}
19,20	Cat 4: mix of 10kB, 512kB, 5MB, 50MB, 2GB	19: 70ms 20: 700ms	0
21,22,23,24	Cat 5: mix of 10kB, 512kB, 5MB, 50MB, 2GB	21,22: 70ms 23,24: 700ms	21,23: 10^{-6} 22,24: 10^{-3}
25,26	Cat 6: 2GB	25: 70ms 26: 700ms	10^{-3}

During each test, the tcpdump utility on traffic monitoring server was used to capture first 68 bytes of each packet. After each test, raw captured data was uploaded to a separate system with large storage for off-line analysis. The analyses allowed to account data packets separately from protocol control messages and was protocol specific. All performance indicators in this study are for goodput – actual data without protocol overhead.

Common network monitoring systems like MRTG, Munin, Zabbix poll network traffic counters rather rarely – every minute or half minute. Online monitoring functions like ones built into modern networking devices from Cisco, HP, Mikrotik and other manufacturers may poll counters and dynamically refresh their display every second or so. All these systems would calculate average number of bytes or bits for every step and express these rates in bytes or bits per second. For instance, if 120MB of traffic has passed during a 60 second step, the average throughput for the whole step would be 2 megabytes per second – 2MBps:

$$\frac{120MB}{60s}=\frac{2MB}{1s}=2MBps$$

But when analysing headers that are captured with microsecond precision timestamps like in this study one can use much shorter steps and still express the throughput during any of them in bytes per second. For instance, if 150kB of traffic has passed during a 10ms (millisecond) step, the average throughput for that step would be 15000 kilobytes per second – 15MBps:

$$\frac{150kB}{0{,}01s}=\frac{15000kB}{1s}=15MBps$$

Also, step length can be tailored for analyses of different scenarios to better demonstrate the influence of scenario parameters to protocol traffic pattern. This approach gives flexible analyses granularity but the same universal traffic notation – bytes per second.

To better observe behavior of the protocols and applications, two histograms were generated for each scenario:

- 1st histogram: 5 second period for the 70ms RTT scenarios; 50 second period for the 700ms RTT scenarios;
- 2nd histogram: 15 minute period for all scenarios.

The reason why the first histogram represents the longer period of 700ms RTT scenarios is that in most such scenarios little data throughput was observed during the first 5 seconds. These seconds were mostly used for session initiation and negotiation. It can be seen in the actual histograms that traffic pattern of 5 second period at 70ms RTT was very similar to that of 50 second period at 700ms RTT, given other parameters equal. This confirms earlier observations that TCP throughput is inversely proportional to RTT [68].

Although all the tests were run for at least 1 hour, the 15 minute interval was determined to demonstrate at least one full session, yet be short enough to distinguish session cycles.

Graphs are drawn in logarithmic throughput scale to better illustrate performance patterns in presence of highly disproportionate absolute values.

6.3. File Transfer Applications and Test Results

The following applications performance results were obtained based on data gathered during this study. Figure 25 shows the average throughput of all tested applications for all scenarios for one hour period.

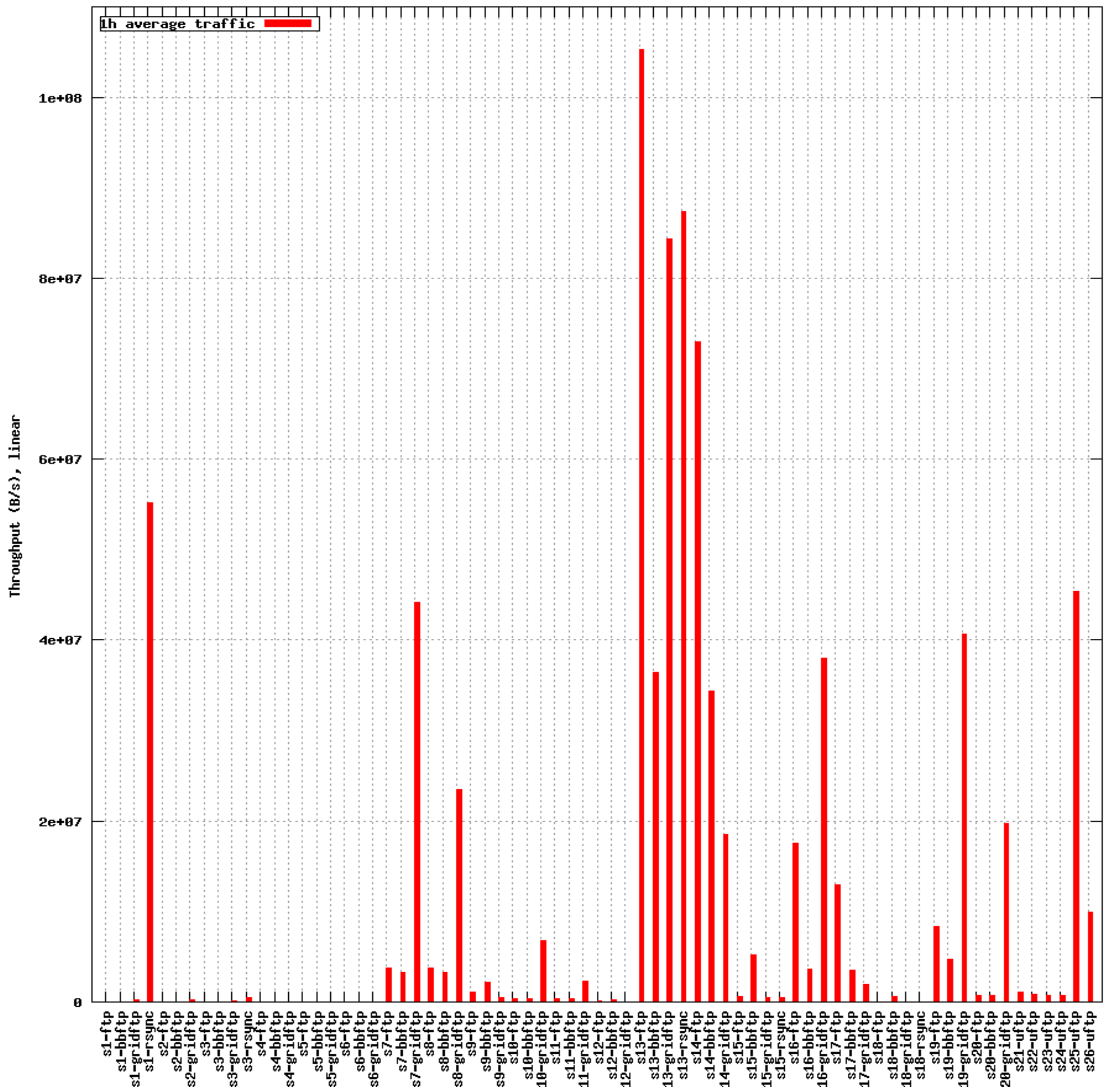

Figure 25. Throughput of all applications in all scenarios, B/s

Table 18 shows the 1 hour statistical summary for all scenarios for all tested applications. The scenarios that indicate higher than 350Mbps (43,7MBps) throughput are marked bold. This performance level is called *target* in this document as it was specified by the EUMETSAT as the targeted minimum throughput for next generation applications.

Table 18. 1 hour statistical summary for all scenarios

Scenario	*Total traffic*	*Total through*	*Avg thr B/s*	*Min thr B/s*	*Med thr B/s*	*Max thr B/s*
s1-ftp	98798152	90206490	25057	23997	24576	218190
s1-bbftp	102490936	83732480	24130	18432	24576	24576
s1-gridftp	700547176	657526898	183666	3818440	147701	147701
s1-rsync	205428000000	198306000000	**55238440**	10056900	65973300	91705876
s2-ftp	98798377	90206490	25057	23997	24576	218190
s2-bbftp	99509732	81283568	23424	17563	24421	24576
s2-gridftp	700765796	657730590	183721	3625710	147701	4481836
s3-ftp	94617442	87565575	24876	22683	24576	210150
s3-bbftp	96062232	78449776	22871	12288	23552	24721
s3-gridftp	535031529	503686874	140693	69253	115049	831007
s3-rsync	1921907444	1855223060	515337	1057330	493768	918679
s4-ftp	11628345	10713370	2976	2048	2338	187739
s4-bbftp	8951708	7219200	2080	0	2048	2627
s4-gridftp	193750018	185222819	51875	0	14360	1315034
s5-ftp	11628345	10713370	2976	2048	2338	187595
s5-bbftp	8951708	7219200	2080	0	2048	2627
s5-gridftp	193739012	185212562	51879	0	14360	1307070
s6-ftp	11541802	10634554	2970	2048	2338	115116
s6-bbftp	9014204	7301016	2110	0	2048	3072
s6-gridftp	152888272	147152083	41216	0	42571	367502
s7-ftp	14058200000	13569400000	3769257	3232990	3764570	4082696
s7-bbftp	12202800000	11612700000	3225741	2827120	3226960	3497648
s7-gridftp	165765000000	158981000000	**44161389**	14530900	44435500	44655597
s8-ftp	13944200000	13459700000	3738774	2655150	3768770	4058804
s8-bbftp	12167600000	11579300000	3216472	2621730	3211500	3461348
s8-gridftp	83686300000	80249800000	23533459	10275400	23668000	37197661
s9-ftp	4045300000	3904640000	1087625	1000650	533266	2087740
s9-bbftp	8321520000	7905410000	2208215	1083650	2367490	3130368
s9-gridftp	1601828134	1531701768	426655	200841	406743	821595
s10-ftp	1539232114	1485711010	412623	959	471870	523998
s10-bbftp	1227398068	1167998936	324410	0	351542	516758
s10-gridftp	25200100000	24174200000	6770309	2761130	6816186	7340508
s11-ftp	1538606614	1485107194	412454	959	472015	523998
s11-bbftp	1223461064	1164301632	323406	0	351252	516758
s11-gridftp	7594800000	7285420000	2290580	1038220	7181200	7340508
s12-ftp	411241463	396941154	110568	959	68023	520234
s12-bbftp	794257696	754380472	218638	0	170719	513996
s12-gridftp	169144213	161574630	45132	15494	42716	112771
s13-ftp	392482000000	378875000000	**105243056**	100053000	93516700	117526920
s13-bbftp	137731000000	131100000000	36416667	28825300	37572600	37664724
s13-gridftp	314815000000	298728000000	**84386441**	101080000	84471800	85917804
s13-rsync	325725000000	314432000000	**87342222**	100207000	69221000	111056124
s14-ftp	271341000000	261934000000	**72962117**	100081000	75159000	117175056
s14-bbftp	129899000000	123648000000	34346667	10551700	37446200	37648356
s14-gridftp	69517300000	65963500000	18477171	10219700	21481400	29673140
s15-ftp	2158030000	2083217328	578670	1036330	557914	1038216
s15-bbftp	19760100000	18780600000	5216833	1058110	5583200	7071204
s15-gridftp	1636452264	1564478976	437002	1918210	418617	739349
s15-rsync	1939736281	1872490967	520136	1322280	504773	974359
s16-ftp	65272400000	63009300000	17551093	10063700	9438090	75146104
s16-bbftp	13704400000	13057300000	3626977	0	3779100	3907984
s16-gridftp	136204000000	129286000000	38025294	20176800	38586900	40267432
s17-ftp	48487000000	46806100000	13001478	10736100	2774220	46955020
s17-bbftp	13087600000	12468500000	3463039	0	3743170	3909432
s17-gridftp	7097600000	6732550000	1951373	1009840	2310720	5377872
s18-ftp	198686616	180468168	50971	0	40978	500718
s18-bbftp	2258790000	2141604224	594885	0	589770	782499
s18-gridftp	171775944	163918352	47374	2358	46481	98609
s18-rsync	208761781	201523591	55978	9551	52128	358494
s19-ftp	30919200000	29844100000	8290005	1037390	336264	113591252
s19-bbftp	18035100000	17163000000	4767500	10322100	212992	37637212
s19-gridftp	154274000000	146395000000	40665278	10029700	6515670	113391432
s20-ftp	2693230000	2599590000	722090	1009570	21441	3753976
s20-bbftp	2674410000	2547000000	707376	0	30708	3828472
s20-gridftp	19809300000	19121900000	19706614	11781500	22724100	33289108
s21-uftp	4011410000	3934480000	1092893	1073430	53270	5416540
s22-uftp	3063190000	3004440000	834543	5370140	53270	78210112
s23-uftp	2845590000	2791020000	775262	1538110	3150	19194528
s24-uftp	2786690000	2733250000	758979	0	3150	19199540
s25-uftp	166613000000	163418000000	**45393889**	11363200	75301900	78357608

For every application, the impact of different file sizes and packet drop rates at fixed RTT was demonstrated. Observations are based on fixed RTT because from all the condition variables particularly RTT is outstanding for two reasons:

1. Both RTT values (70ms and 700ms) represent different real life usage scenarios. The first is a representative RTT for a global terrestrial network. The second is extremely high RTT appropriate for geostationary satellite communication;
2. All the protocols and applications tested in this study demonstrated substantial performance decrease in 700ms RTT link. The best case at this latency was GridFTP in scenario 16 reaching throughput of about 38MBps (304Mbps).

6.3.1 FTP throughput at 70ms RTT

Figure 26 shows seven FTP scenarios: 1, 2, 7, 8, 13, 14, 19. They represent all three file sizes for 70ms RTT and 0 or 10^{-6} loss. The graphs have been grouped in 3 pairs – each for a different file size. The performance differs dramatically: the large files (2GB) are transferred at about 100MBps while small files (10kB) are transferred at about 20kBps. It can also be concluded that low or no packet loss does not impact the average performance much due to Fast Retransmit [69]. Transfer of mixed file set in scenario 19 shows varying average performance as it increased during the transfer of larger files and decreased during the transfer of small files.

Figure 27 shows all three file sizes for 70ms RTT and 10^{-3} packet loss. At this packet drop rate large and medium file sizes show degraded performance while small files show no difference. The last observation can be described by the fact that small files contain only 8 packets and by far most files get sent without a dropped packet. Also at RTT as high as 70ms a rare NACK does not disturb much the weakly performing FTP protocol at small file sizes.

There is another interesting observation to be made in Figure 27 – performance with large files is slightly lower than performance with medium files. It can be described by the fact that 5MB files at 10^{-3} packet loss can often be transferred with just one or few dropped packets. It allows a fast starting TCP connection to complete a 5MB file while maintaining a larger window. However, during sending a 2GB file it gets enough NACKs, often unfortunately close one to another, to be obliged to reduce the window to a smaller size, thus degrading performance.

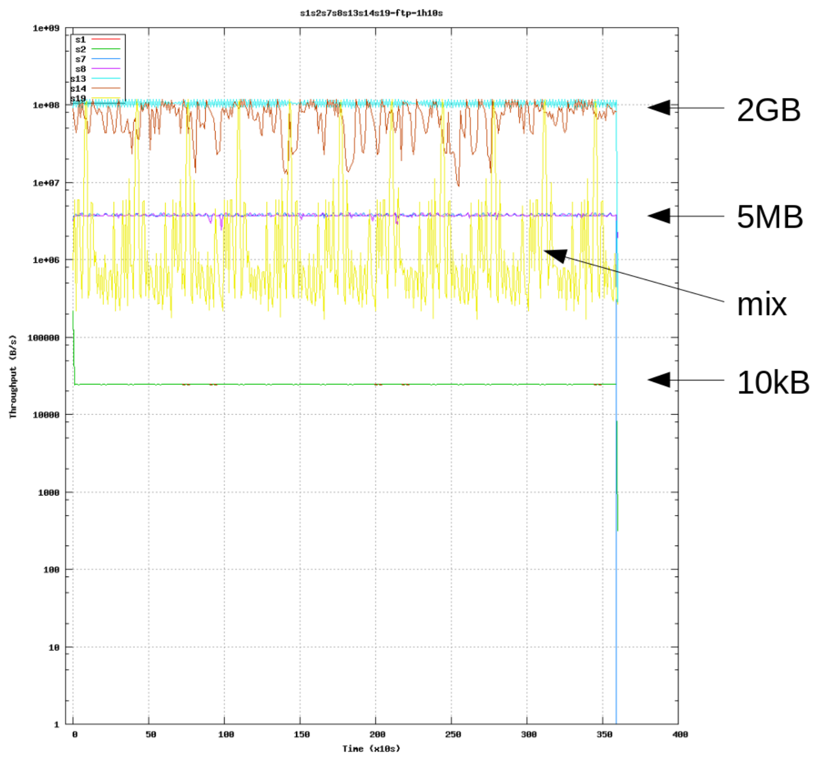

Figure 26. FTP protocol, little or no packet loss, 70ms RTT

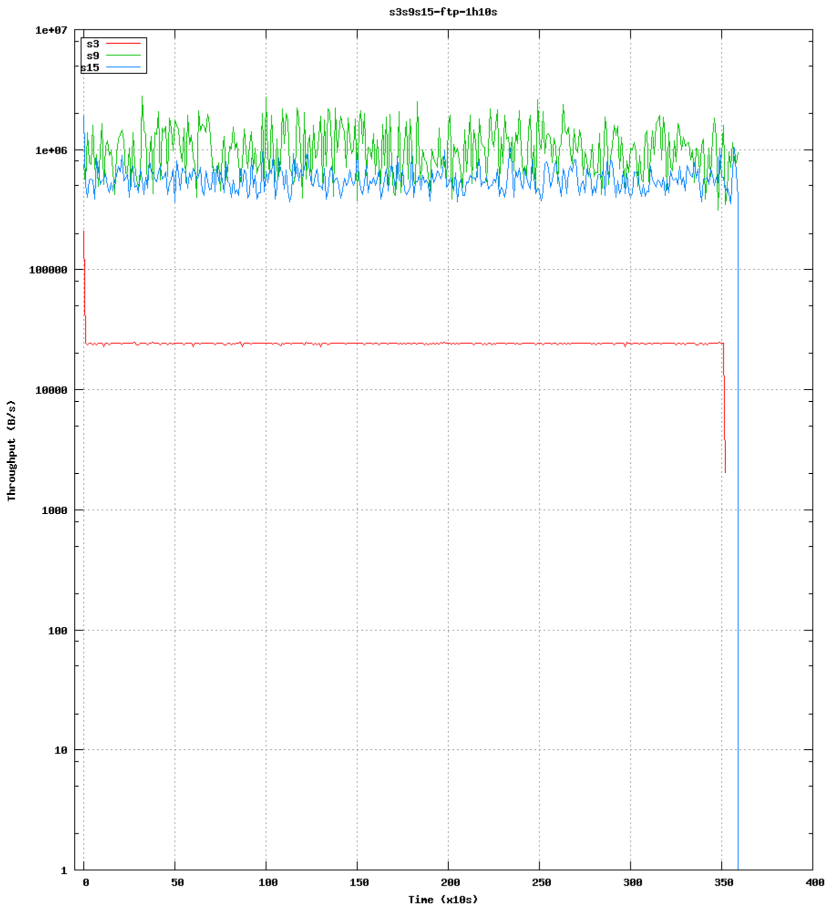

Figure 27. FTP protocol, significant packet loss, 70ms RTT

6.3.2 UFTP throughput at 70ms RTT

Figure 28 shows the performance of UFTP in scenarios 21, 22 and 25.

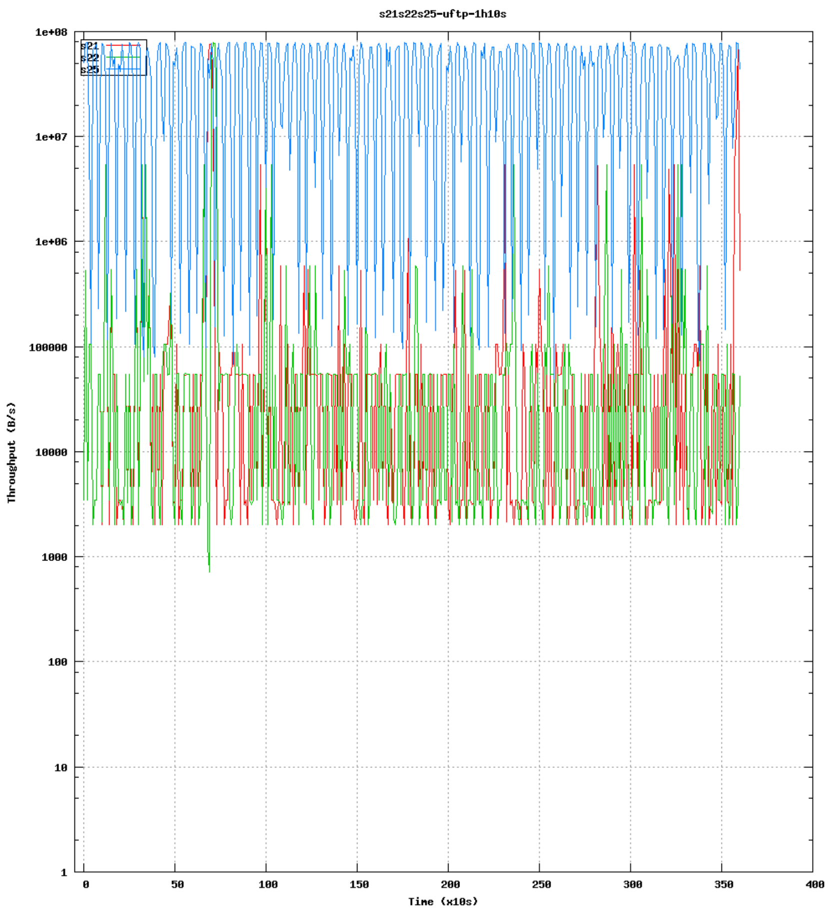

Figure 28. UFTP in scenarios 21, 22 and 25, 70ms RTT

It can be seen that sending mixed file set puts average UFTP performance to about 1MBps.

Only sending large files in a row in scenario 25 allows average UFTP performance to exceed *target*. The graphs are highly fluctuating because of the lengthy session initiation process.

6.3.3 bbFTP throughput at 70ms RTT

Both bbFTP figures (31, 32) are very similar to those of FTP. Throughput patterns in the first graph also have grouped into 3 pairs – each for a different file size. Also, the performance differs quite dramatically depending on a file size: the large files (2GB) are transferred at about 36MBps while small files (10kB) are transferred at about 25kBps. Similarly as with FTP, bbFTP also does not experience any significant impact on average performance in case of low packet loss. Note, however, that in all of scenarios with 70ms RTT and 0 or 10^{-3} packet loss ratio the bbFTP perform markedly worse in case of small or medium files, and considerably worse in case of large file transfer.

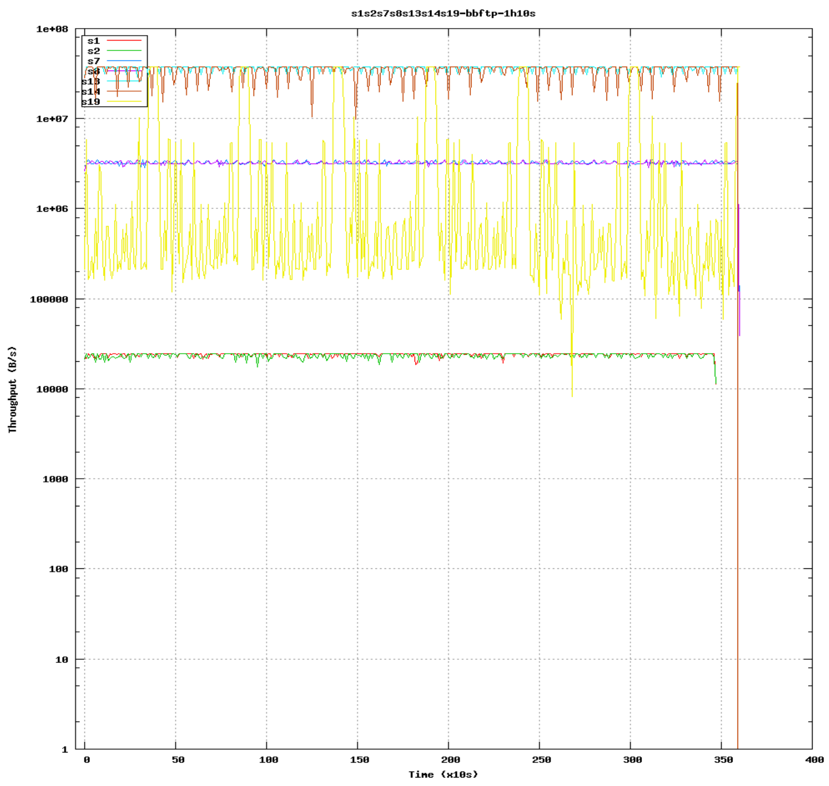

Figure 29. bbFTP protocol, little or no packet loss, 70ms RTT

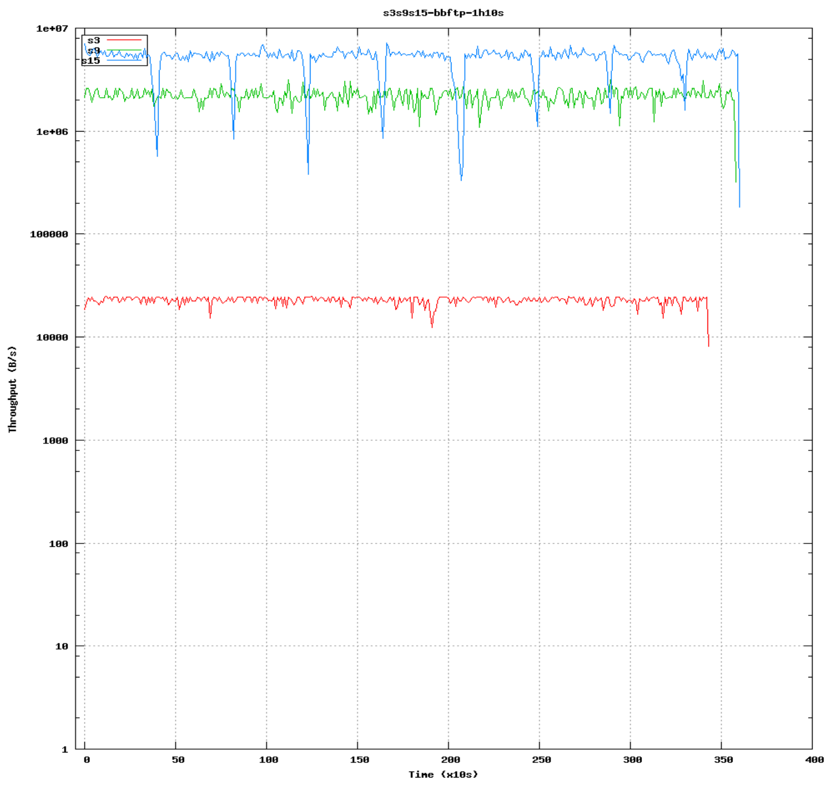

Figure 30. bbFTP protocol, significant packet loss, 70ms RTT

The reason for bbFTP performing almost identically, only slightly worse that FTP, in small file transfer is that bbFTP shares with FTP the same inefficient concept of opening new data connections for each next file. Also, for small files bbFTP is not sending them through parallel data connections, even if usage of parallel streams is specified at command line launch of bbFTP. Thus, in case of small files, bbFTP effectively is the same FTP protocol by design – it uses single control connection for command issuing, and transfers data through a single data connection, which has to be reopened for each next file. The same holds true in comparing FTP and bbFTP performance at 700ms RTT and 0 or 10^{-3} packet loss ratio. In fact, bbFTP results are consistent with FTP results through scenarios 1-6.

As for transfer of medium size (5MB) files in scenarios 7 and 8, bbFTP again reaches slightly smaller throughput. In this case, however, bbFTP is using 10 parallel streams for data transfer. The inefficiency is caused by bbFTP using fixed TCP buffers, either default value of 256KB, or one specified in bbFTP execution parameters. Since for measurement of 5MB file transfer the default buffer size proved to be the most optimal and stable configuration, bbFTP is unable to scale each of the ten parallel TCP data connections enough to achieve higher throughput. If bbFTP would use automatic TCP window size scaling provided by the operating system, it could achieve higher throughput as the size of transferred file is larger.

Lastly, the results for scenarios 13 and 14 show that bbFTP is at least two times slower in transferring 2GB files than FTP. This bottleneck is also caused by bbFTP using fixed and static TCP buffers, rather than relying on automatic TCP window and buffer scaling provided by the operating system. Due to bbFTP instabilities it was not reliably usable with higher count of parallel streams or larger TCP buffers specified in command line parameters.

Completely contrary are the bbFTP results for file transfer in maximum specified packet loss ratio scenarios 3, 9 and 15. The second picture shows all three file sizes and bbFTP transfer throughput for 70ms RTT and 10^{-3} packet loss. As in case of FTP, bbFTP protocol transfers 10kB files with no significant performance penalties as compared with no packet loss scenario. Reasons for this are identical as in case of FTP – the small files are composed of only 8 packets and are thus very unlikely affected by even quite high packet loss. However, bbFTP achieves twice the throughput of FTP in medium file transfer, and ten times the throughput of FTP in transfer of large files. This is due to the same fixed TCP window size used by bbFTP, which caused bbFTP to under-perform compared with FTP in scenarios with no or low packet loss. But in high packet loss scenarios automatic TCP window scaling in Linux is much more cautious and conservative, causing TCP window to be decreased rapidly and keeping it small, thus limiting high throughput. BbFTP does not change TCP window size despite the lost packets and thus maintains much higher throughput.Also, usage of parallel streams help isolate packet loss to one stream at a time, thus decreasing probability of receiving several NACKs in a row that would cause slow restart of TCP window scaling and considerably decrease throughput, as it is for other tested applications, that only use one data connection.

6.3.4 GridFTP throughput at 70ms RTT

Figures 33, 34 show GridFTP performance graphs. Although at first they may seem similar to those of FTP and bbFTP, there are significant differences. Whereas FTP and bbFTP results at 70ms RTT showed 3 distinct pairs of throughput curves depending on transferred file size, the two upper throughput curves for GridFTP are actually determined by the packet loss ratio.

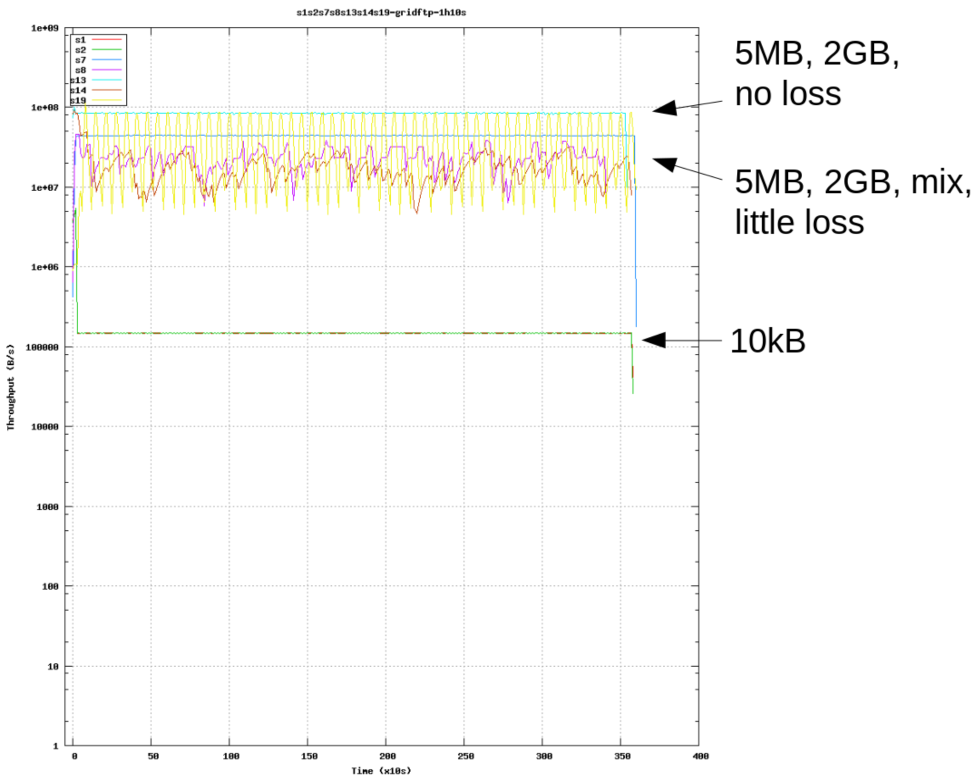

Figure 31. GridFTP protocol, little or no packet loss, 70ms RTT

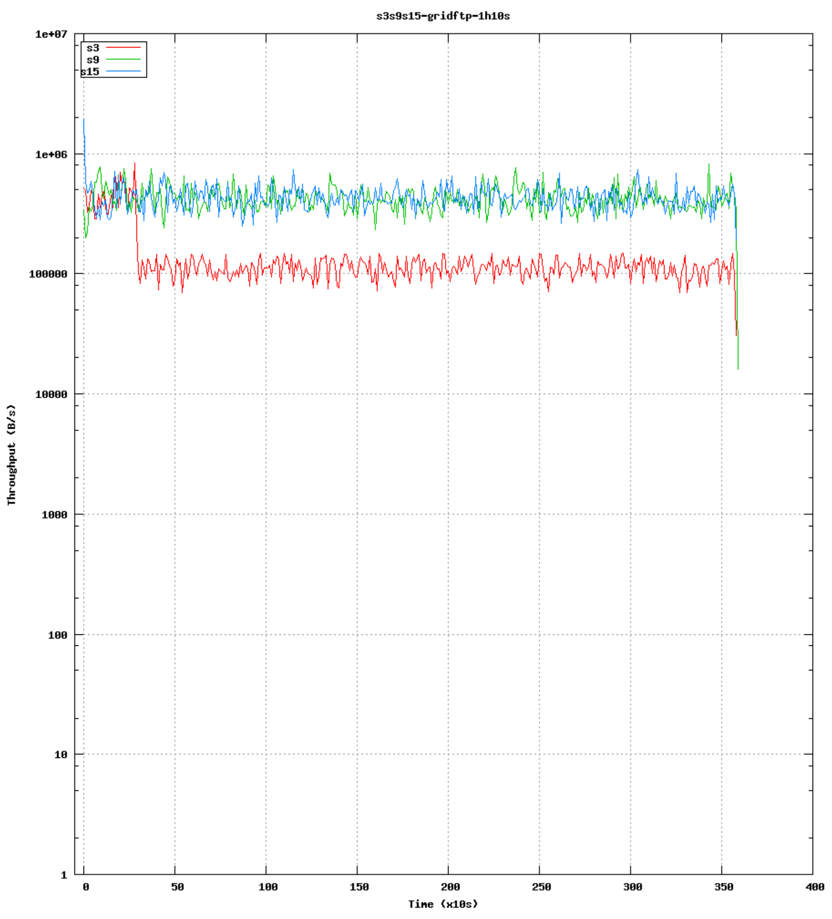

Figure 32. GridFTP protocol, significant packet loss, 70ms RTT

For transfer of small files GridFTP has similar results to FTP and bbFTP. As it can be seen, the file transfer throughput is very low, and after initial peak caused by transfer of directory file list the file transfer session reaches an RTT caused throughput limitation. However, due to more efficient protocol design, GridFTP is able to reach ten to twenty times the throughput of FTP or bbFTP on all small file scenarios. This is because GridFTP reuses already established TCP data connections for sending each subsequent file. Although GridFTP was used with 10 parallel connections, the small files are not striped across multiple connections or sent several simultaneously. The gain in performance is solely due to re-usage of data connection(s). Similarly as in case of FTP and bbFTP, GridFTP protocol does not experience performance degradation in transferring small files at high packet loss. But, even though GridFTP achieves up to twenty times the throughput of the other two unicast applications in scenarios 1-6, it is still results in only 183 kBps for 10kB files on 70ms RTT links with no packet loss.

As for transfer of medium size files in scenarios 7 and 8, GridFTP is able to reach significantly higher throughput and again outperforms FTP and bbFTP by more than ten times. While the other two applications are able to achieve only 3.2-3.7 MBps in scenario 7, GridFTP achieves 44 MBps and reaches target throughput of the study. Due to relying on TCP window scaling provided by the operating system, usage of parallel streams, as well as re-usage of data connections that have already reached large TCP window size, GridFTP is capable of high-throughput file transfer even for file size of 5MB.

Scenarios 13 and 14 show that the full benefits of GridFTP protocol were achieved when transferring large files over parallel streams. Usage of multiple streams and TCP window scaling proved to be most advantageous for medium and large files on links with no packet loss, since GridFTP has the opportunity to accelerate and reuse each data connection. But, this same approach seems to be less effective in presence of packet loss. The larger is the file transferred over a TCP connection, the more likely is that some packets will be dropped for that file. Thus, the longer a TCP connection is used on a lossy link, the more likely this connection will experience degrading throughput over time due to several close packet drops that cause decrease of TCP window. Hence, re-use of existing data connections in case of high packet loss is undesirable. It is much more effective to use a connection only for a lifetime of singe file transfer, and then reopen new connection with the highest possible initial TCP window.

Thus, the strategy of whether to re-use existing data connection for the length of whole file transfer session or to create a new data connection for each subsequent file is dependant on packet loss rate on the link. If there is no packet loss, re-usage of open data connections is the most effective approach. In case the packet loss is high, each TCP connection should be used for as short duration as possible.

This explains why throughput curves are almost identical for scenarios 8 and 14 – because if the same data connection(s) are reused for the whole duration of file transfer session, the TCP window in both cases will converge to the same size depending on packet loss rate, and irrespective of whether medium or large files are being transferred. As it can be seen, in case of high-throughput of large files packet loss rate is much more important than RTT.

6.3.5 FTP throughput at 700ms RTT

Figure 31 shows seven other scenarios for FTP: 4, 5, 10, 11, 16, 17, 20. They represent all three file sizes for 700ms RTT and 0 or 10^{-6} loss. It can be easily seen that the graphs are grouped in 3 pairs again – each for a different file size. The performance also differs dramatically but the throughput is significantly lower. Rare packet loss impacted average performance of the large file transfer in scenario 17. Occasional packet drops close one to another made TCP to reduce window and drop performance in few of the transfers. This is indicated by the "ladders" in the graph. Transfer of mixed file set in scenario 19 again shows varying average performance. Comparing Figure 33 to the Figure 26 (FTP, 70ms) an observation can be made that traffic patterns are "stretched" about 10 times as RTT becomes 10 times longer. At the same time, performance decrease was about 10 times as the RTT was increased 10 times. As mentioned previously, it complies with findings in [68].

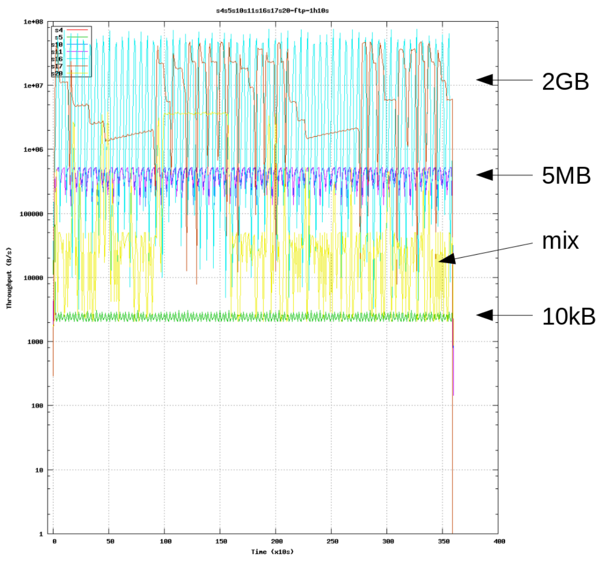

Figure 33. FTP protocol, little or no packet loss, 700ms RTT

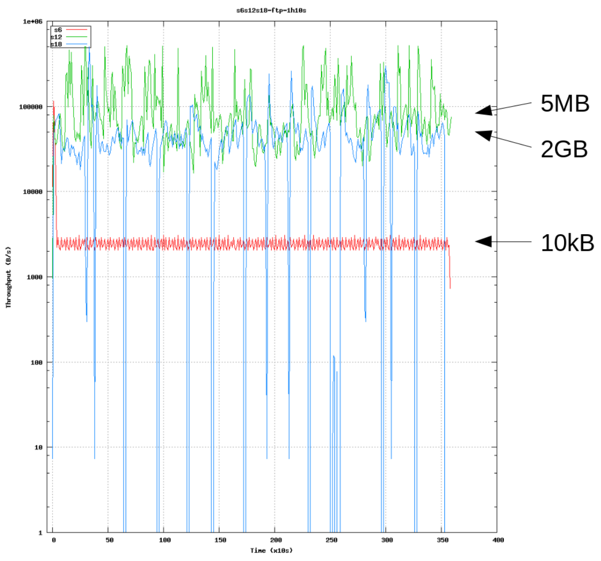

Figure 34. FTP protocol, significant packet loss, 700ms RTT

Figure 33 shows all three file sizes for 700ms RTT and 10^{-3} packet loss. At this packet drop rate, large and medium file sizes show degraded performance while small files show no difference. Larger RTT decreased performance even more. As a result, 2GB file transfer at worst conditions was possible only at average throughput of 50kBps.

As in case with 70ms RTT here, too, performance with large files is slightly lower than performance with medium files. It reason is the same – fast starting TCP can transfer most 5MB files with few NACKs and does not need to decelerate.

As in case with 70ms RTT, also here performance with large files was slightly lower than performance with medium files. The reason is the same – fast starting TCP could transfer most 5MB files with few or no lost packets and did not need to decelerate [70].

6.3.6 UFTP throughput at 700ms RTT

Figure 35 shows UFTP in scenarios 21, 22 and 25, 700ms RTT.

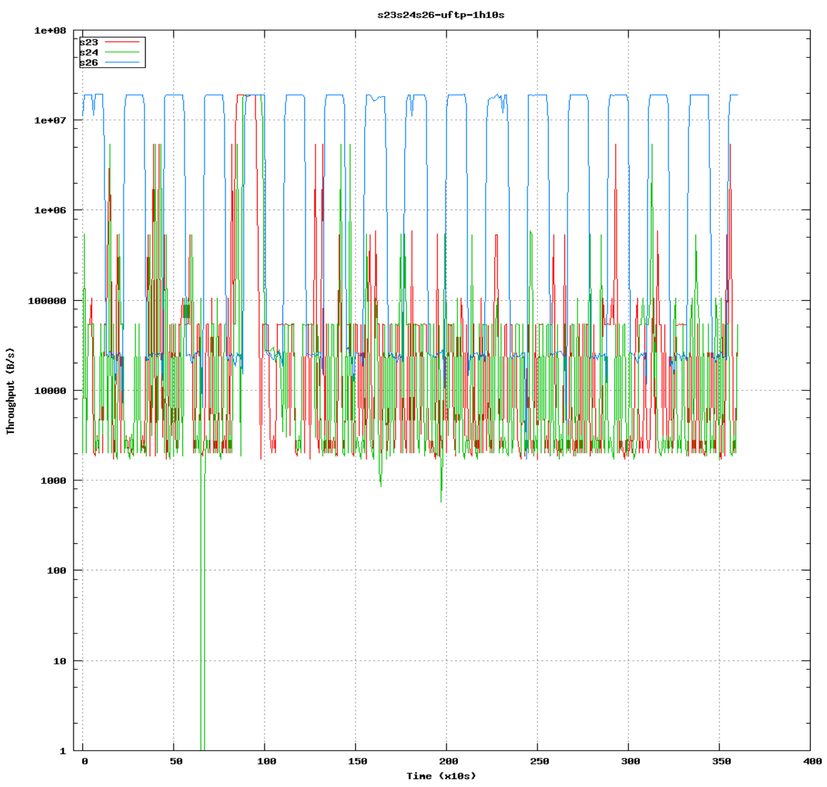

Figure 35. UFTP in scenarios 21, 22 and 25, 700ms RTT

Interesting to see in Figure 35 that average UFTP performance of mixed file set at 700ms RTT is not much lower than at 70ms. The reasons are that most of the time is spent in fixed length initiation process. Also most of the files in the file set are less than 8MB in size and can be transferred in one burst.

Sending large files in a row in scenario 26 is impacted by higher RTT. The reason is that UFTP protocol waits for ACK after every 8MB burst. UFTP performance here does not reach target.

6.3.7 bbFTP throughput at 700ms RTT

bbFTP graphs at 700ms RTT once more show similarities to results from corresponding FTP measurements. As previously explained in analysis of bbFTP results at 70ms RTT, in case of small file transfer bbFTP behaves almost identically to FTP, and they both reach almost the same throughput as in scenarios 1-6.

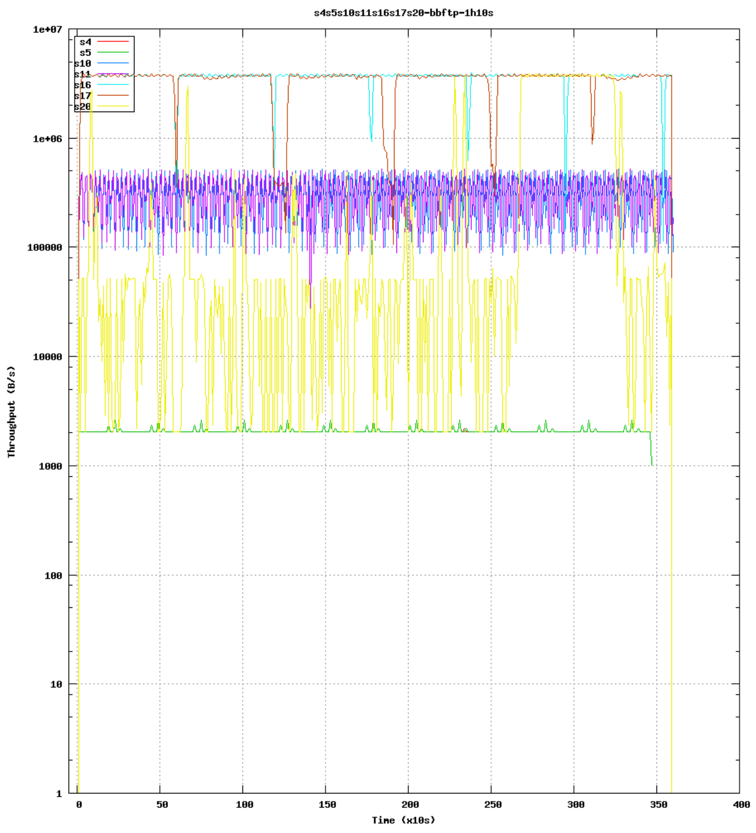

Figure 36. bbFTP protocol, little or no packet loss, 700ms RTT

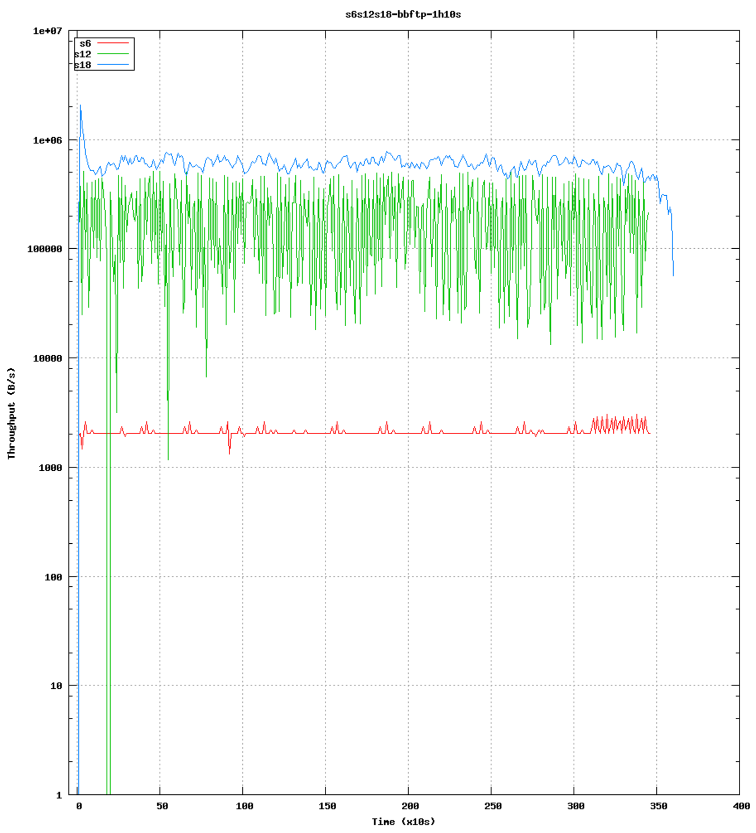

Figure 37. bbFTP protocol, significant packet loss, 700ms RTT

Throughput patterns in the first graph are once again grouped into 3 pairs – each for a different file size. Again, the performance differs quite dramatically depending on a file size: the large files (2GB) are transferred at about 3.6MBps while small files (10kB) are transferred at about 2kBps. In case of 700ms RTT, just as for FTP, RTT has a direct impact on throughput of protocol. By increasing RTT 10 times from 70ms to 700ms, throughput drops accordingly from 36Mbps to 3.6Mbps for large files, and from 25kBps to 2kBps for small files. Once more the results suggest that at such a low throughput packet loss has almost no impact on the results. This clearly shows that for transfer of small files network latency is crucial limiting factor.

Especially noteworthy is the observation that, indeed, as the mathematical models of latency limited protocols predict, achievable throughput is inversely proportional to RTT. These results fully support initial predictions that FTP and bbFTP throughput will be limited by RTT, and that throughput results at 700ms RTT should be 10 times lower than at 70ms RTT. This clearly suggests limitations in the protocol, and not any inefficient implementation as initially suggested in evaluation of various FTP servers (vsftp, proftpd).

As for transfer of medium size (5MB) files in scenarios 10 and 11, bbFTP again reaches slightly smaller throughput than FTP. As previously noted in analysis of 70ms RTT results, the fixed TCP buffers used by bbFTP prohibit bbFTP from accelerating parallel data streams to significant throughput. Instead, automatic TCP window scaling of the operating system should have been used.

Finally, the results for scenarios 16 and 17 show that bbFTP is now five times slower in transferring 2GB files than FTP, as opposed to being just two times slowed at 70ms RTT. As it can be seen, since the TCP buffers used by bbFTP are of fixed size, the disadvantage in throughput scale linearly as the RTT is increased, just as in case of small files. Whereas standard FTP is finally able to fully utilize large TCP window scaling of the operating system and thus provide significantly better performance (smaller performance drop).

Figure 37 shows all three file sizes and bbFTP transfer throughput for 700ms RTT and 10^{-3} packet loss. Just like in case of 70ms RTT, bbFTP is at least able to get some benefit from fixed TCP buffers at very high packet loss. As does FTP, bbFTP transfers 10kB files with no significant performance penalties in comparison with no packet loss scenario. For 5MB files bbFTP again achieves twice the throughput of FTP, as well as previous tenfold improvement in large file transfer due to non-existing TCP window scaling.

By comparing 700ms RTT results for bbFTP and FTP it can be seen that, although in most scenarios producing lower file transfer throughput, bbFTP transfers data with much more smoother traffic pattern, whereas FTP experiences visible jitter in transfer throughput. This is direct visualisation of using fixed TCP window versus TCP window scaling. For large enough files FTP is able to accelerate data transfer much further than bbFTP, at the cost of dropping transfer speed significantly in case of severe packet loss. bbFTP in contrast, achieves maximum throughput instantly, and maintains it very consistently despite packet loss at the expense of maximum throughput. If bbFTP would have been implemented with better stability and ability to scale with parallel streams and large TCP buffers, it could potentially achieve very high throughput in all scenarios with medium and large files, irrespectively of packet loss. Unfortunately, due to poor implementation of bbFTP application, the full potential of bbFTP protocol is not reached, and bbFTP fails in high-speed transfer of large files on WAN networks.

6.3.8 GridFTP throughput at 700ms RTT

The results for GridFTP at 700ms RTT (Figure 38 and 39) are consistent with previously described operation of GridFTP protocol.

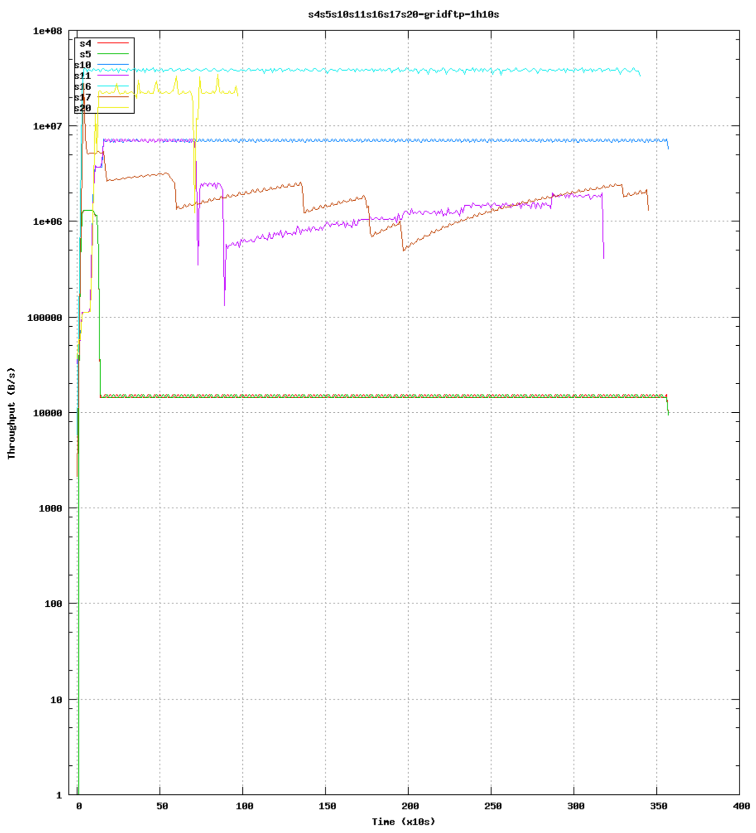

Figure 38. GridFTP protocol, little or no packet loss, 700ms RTT

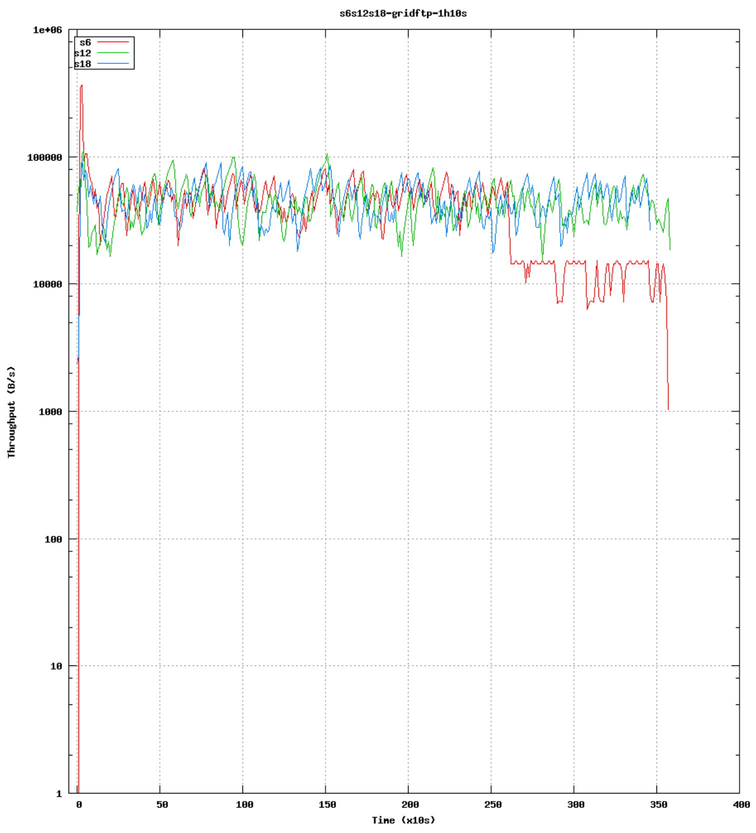

Figure 39. GridFTP protocol, significant packet loss, 700ms RTT

In scenarios with no or low packet loss rate GridFTP significantly outperforms FTP and bbFTP, whereas in scenarios with high packet loss rate, the larger files are transferred, the more throughput is degraded. The first graph clearly shows decrease of TCP window during file transfer for scenarios with moderate file sizes. As the TCP window is decreased due to subsequent close packet drops, throughput is not restored back to previous level before the next packet drop takes place. In case the packet loss rate is significant, as shown in Figure 39, throughput for all connections will converge to a very low value due to ever shrinking TCP window.

6.3.9 RSYNC throughput

The RSYNC application was added to the research because it uses single TCP connection for the whole transfer session. Figure 40 summaries all five RSYNC scenarios.

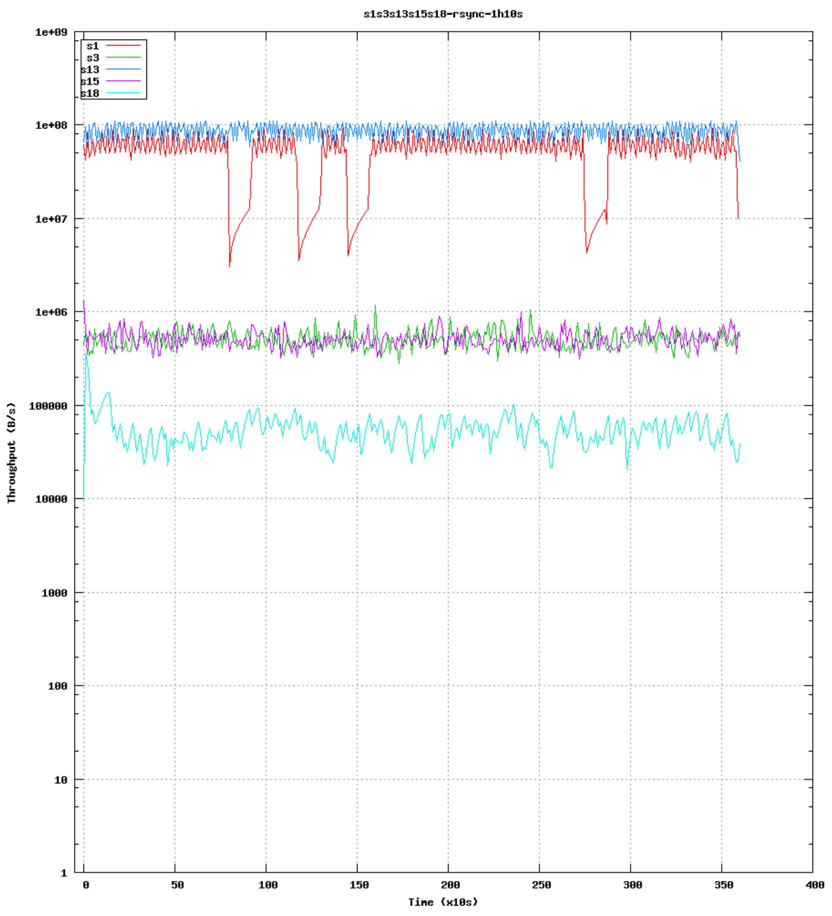

Figure 40. RSYNC protocol performance

RSYNC application was tested in the following selected scenarios with extreme parameters:

- s1 – best conditions at 70ms, small files
- s3 – worst conditions at 70ms, small files
- s13 – best conditions at 70ms, large files
- s15 – worst conditions at 70ms, large files
- s18 – worst conditions at 700ms, large files

Figure 40 shows that RSYNC performs in the range of 50-100MBps in both best condition scenarios. The file sizes make relatively small difference. There is no other application in this study that shows that high performance with 10kB files. At that throughput it is sending around 10000 files every second which was possible only by pre-caching files before tests and dumping them to RAM disk during tests.

In real life user would need high performance storage system if files need to be fetched from disks on demand at that rate.

In worst condition scenarios RSYNC heavily looses performance as TCP needs to retransmit packets and shrink window in response to frequent packet drops. Scenarios 3 and 15 show around 500kBps, while scenario 18 shows around 50kBps. Again, 10 times increased RTT makes 10 times decreased performance.

6.4. Conclusions and recommendations on the protocol performance

6.4.1 Conclusions on application suitability

The criterion of application suitability in this study was the earlier defined *target* performance. The target performance level was specified by the EUMETSAT as the minimum throughput of 350Mbps (43,7MBps) for the next generation real time content delivery.

6.4.2 FTP suitability

FTP protocol mandates a separate TCP connection for control session and a new TCP connection for every data stream. A data stream is either a data file or a directory listing. FTP commands get sent over single permanent control connection. Inefficiency of the multiple file sending process can be clearly seen in the analyzed data and histograms given earlier in this document. It took at least two round-trips to initiate a new file download in the presence of an already open session. In case of 70ms RTT, it meant at least 140ms lost in protocol "chat" for every file regardless of its size.

The analyses show that FTP reached *target* only in scenarios 13 and 14, which were sending 2GB files at 70ms RTT and no or rare packet loss. None of the other FTP scenarios reached *target* as a consequence of either smaller files, higher RTT, more frequent packet loss, or a combination of these.

6.4.3 UFTP suitability

UFTP application spent at least 4 seconds for every session initiation in the tests. The sender was always explicitly provided with IP addresses of both receivers. Session initiation took even longer time in case of open client participation. UFTP had to start a new session for every file it sends. As a result, UFTP application is unsuitable for sending files of small or medium size. Those are even out of scope in a UFTP design and performance study [71] that focuses only on large files.

UFTP reached *target* only in scenario 25, which was sending only large (2GB) files. The other parameters were: 70ms RTT and high packet loss. One of the reasons why this scenario was added to the test plan was to show at least one multicast scenario when UFTP reaches *target*. None of the other UFTP scenarios (mixed file sizes) reached *target* as a consequence of either smaller files, higher RTT, or a combination of both.

Scenario 25 shows another interesting point. UFTP does not suffer much from packet drops. Its delivery process does not retransmit a lost packet immediately following a NACK. Instead it continues to transmit file at the given rate (900Mbps in all tests) and collects NACKs for the next phase. During a subsequent phase, it transmits only the lost fragments at the same given rate. It repeats phases until every receiver has received a complete file. Another welcome feature of UFTP phased delivery process is that any receiver that has received a complete file finishes the session with the sender while other receivers may continue with more phases in case of NACKs.

Scenario 25 was run in presence of worst packet loss. UFTP was the only application that reached *target* at worst packet loss rate (10^{-3}).

6.4.4 bbFTP suitability

Although bbFTP protocol allows sending large files over multiple parallel streams, it has the same protocol limitations as standard FTP. bbFTP reopens data connections for transfer of each subsequent file, thus it was not possible to send files of small or medium size at a high throughput. Moreover, bbFTP uses fixed TCP window size, and has several implementation restrictions on stable settings for the number of parallel streams and TCP window sizes used. The usage of fixed TCP window size may have advantages only in high packet loss scenarios. Even then, bbFTP was unable to achieve high enough throughput.

bbFTP did not reach *target* throughput in any scenario, it performed worse than even standard FTP in all scenarios except ones with packet loss ratio of 10^{-3}. Best case throughput for bbFTP – scenario 13: 36MBps (288Mbps).

Also, bbFTP was poorly implemented and crashed periodically during deployment, configuration and execution of test scenarios. If properly implemented, ensuring more stable and reliable operation, as well as allowing usage of more than ten parallel streams and optimal performance with non-default TCP buffer parameters, bbFTP could possibly be considered for use in large file transfer in high packet loss cases. But, considering the state of bbFTP at the time of this study, it was more perspective to research on the possibilities of achieving the same benefits of using fixed TCP windows during transfers with high packet loss by tuning GridFTP operation specifically for high packet loss scenarios.

bbFTP cannot be suggested based on the data gathered in this study.

6.4.5 GridFTP suitability

GridFTP protocol opens permanent data connection(s) that can be reused to transfer multiple files. This feature resembles protocols like RSYNC and clearly allows achieving higher throughput with small files. For files large enough (bigger than what can be sent within one TCP window), GridFTP was able to utilize parallel transfer of single file over several streams – a feature common with bbFTP. However, it could not be confirmed how scalable GridFTP was in capability to send multiple files simultaneously over the open parallel connections, as it was outside the scope of this study.

Due to GridFTP capability to reuse open data connections, utilize TCP window scaling provided by operating system, as well as parallel transfer of large files through several streams, GridFTP is highly suited for medium and large file transfer on WAN networks with no packet loss. But in case of packet loss on the network, GridFTP will experience dramatically decreased throughput depending on packet loss rate, amount of parallel streams used, and duration of file transfer session.

GridFTP reached *target* only in scenarios 7 and 13, which were sending 5MB and 2GB files at best conditions (70ms RTT and no packet loss). None of the other GridFTP scenarios reached *target* as a consequence of either smaller files, higher RTT, more frequent packet loss, or a combination of these.

6.4.6 RSYNC suitability

RSYNC protocol opens single TCP connection for the duration of whole session. This connection carries all the control commands and file data, including multiple file transfer. Upon starting the session, RSYNC application compares the given local directory with the given remote directory, calculates differences and only then starts to send actual files. This initial comparison makes the average throughput of the tests lower than the performance that can be observed during actual file transfer.

Still, single TCP connection process gives good results with all tested file sizes in case of no significant packet loss. RSYNC reached *target* in scenarios 1 and 13, when sending the smallest as well as the largest files (10kB and 2GB, respectively) at best conditions (70ms RTT and no packet loss). RSYNC was the only application that reached *target* with 10kB files. The other RSYNC tested scenarios (3, 15, 18) had worse conditions and did not reach *target*. This limitation on achievable throughput at high RTT or packet loss is common for all tested applications that rely on TCP window scaling provided by the operating system (FTP, GridFTP, RSYNC).

6.4.7 Conclusions on Applications and Protocols

Exceptional results produced within the study:

- Highest performance – FTP in scenario 13: 105MBps (840Mbps);
- For small files (10kB) only RSYNC reached *target* (scenario 1: 55MBps (440Mbps));
- At worst packet loss (10^{-3}) only UFTP reached *target* (scenario 25: 45MBps (360Mbps));

- At 700ms RTT only GridFTP came close to *target* (best case – GridFTP in scenario 16: 38MBps (304Mbps)).

The applications that can surpass target are:

- FTP, scenarios: 13, 14
- UFTP, scenario: 25
- GridFTP, scenarios: 7, 13
- RSYNC, scenarios: 1, 13

Only FTP application could surpass 2x *target* mark (700Mbps, 87,5MBps) (scenario 13). GridFTP and RSYNC were close to that mark in that scenario reaching 84MBps and 87MBps, respectively.

It can be concluded that four of the five applications tested have shown their best performance at some specific scenarios. Any of them may be considered for use depending on the anticipated file sizes and infrastructure or data dissemination process constraints. Only bbFTP was unable to reach *target* and can be excluded from further consideration.

6.5 Recommendations Based on the Analysis of Data

Several recommendations were given in the study considering various possible assumptions about the infrastructure. Summarizing all recommendations at different assumptions the following general but not strict recommendation was made: use UFTP for multicast or if packet loss is high, otherwise use RSYNC or “tar+nc”.

The “tar+nc” as a very simple recommendation emerged as an afterthought after the detailed tests contracted by EUMETSAT and that was described here. Tar+nc is a combination of archiving tool “tar” and network session tool “nc”. It packs together a given set of files in a single network session. The author believes the performance patterns of such solution to be similar to rsync, but without the need to compare directories at session beginning. These tools are present in any mature network operating system and have evolved to be very powerful, yet simple and achieving same top throughput rates.

6.5.1 Recommendations on Dissemination process

The end-to-end dissemination can utilize simple one tier structure with one or few sources of information acting as senders (or servers) and retrievers of information acting as receivers (or clients).

If the throughput capabilities, delay and quality (packet drops and jitter) parameters throughout the infrastructure are very disperse a two tier dissemination structure is recommended. It suggests to have Tier 1 stations that have capabilities to quickly and reliably retrieve information from the one or few master sources. Tier 2 stations may not have the resources available to Tier 1 stations, so they retrieve information from Tier 1 stations.

The tiers can be organized in tree structure according to geographical location. For example, a tier 1 station might be located in every region or larger country utilizing short latency / high throughput connections to major Internet Service Providers (ISPs) in that area. Then all tier 2 stations might benefit from retrieving information from the local Tier 1 station and also decrease load on master sources.

In any of these dissemination structures the receivers would need to periodically check for presence of newer files on the source or agree on some timed retrieval scheme. Another and more deterministic way to make sure new information is quickly disseminated throughout infrastructure is to use push dissemination. Such a system uses agents that can initiate information retrieval on a remote system. All applications tested in this project can be used in such way. The security risks can be minimized to generally acceptable level.

Another valuable feature of multiple tier structure is that every tier may employ radically different transfer methods. For example, tier 1 stations could use push dissemination while tier 2 stations could use traditional pull dissemination. Also some of the tiers may use multicast transfers while other – unicast.

7 Conclusions

A new language has been proposed in this thesis – Packet Transformation Language (PTL). It is a universal language for defining reversible transformations of protocol data units (PDUs) for tunneling and network virtualization scenarios. PTL concept, theoretical groundwork, originality, syntax and a set of initial functions have been described.

The PTL engine is proposed. It processes *forward transformations* (*FTs*) on egress PDUs and *reverse transformations* (*RTs*) on ingress PDUs, thus restoring the original content of PDUs that were before the *FTs*. The *reversibility principle* is proposed and reasoned about. A lemma is proved: *A forward transformation is reversible if all its functions are reversible.* Upon this a Reverse Transformation Generator (RTG) is proposed that can construct an *RT* from any given *FT*, given that all functions in the *FT* have known reverse functions.

The ZERO protocol for efficient Ethernet-over-IP tunneling has been presented in this theses along with formal proof of its transparency, efficiency, and convergence. The core ZERO protocol is suitable for controlled service-provider networks where guaranteed transparency and efficiency is required. The core protocol has been developed with satellite service-provider networks in mind, but it could equally benefit also other infrastructures where true L2 transparency is required for the Internet of things or other purposes.

The overhead-less nature of ZERO tunneling enables new IP network design patterns, where user IP addressing and routing is fully isolated from the service provider IP addressing and routing through the L2 abstraction. This design principle extends also to the ZERO protocol capability of tunneling IPv6 without any overhead over IPv4 legacy infrastructure thus providing an easy migration path.

The ZERO protocol extensions are discussed that disrupt full transparency or efficiency guarantee, but enable ZERO protocol use over un-controlled public Internet, including support for NAPT gateway traversal. The extended ZERO protocol is aimed at end-users ready to tolerate non-essential frame modification to achieve overhead-less L2 connectivity through public Internet.

The extended ZERO protocol can operate on top of service-provider core ZERO protocol – the overhead-less operation is preserved for both thanks to their reliance on modifying different (*identification/port* or FO/EB, respectively) header fields.

Two ZERO protocol prototype implementations (user-land and kernel) have been demonstrated and tested both in lab and across public Internet. The test results confirm nearly zero overhead efficiency (99,94% packets have no overhead) of the ZERO protocol. The kernel implementation also demonstrates high performance on 1Gbps infrastructures.

The new technologies (PTL, ZERO protocol, stream processing in a virtualized environment) and protocol performance over global networks research described in this thesis can play significant role in the National federated cloud.

Existing cloud solutions like OpenStack, CloudStack [59], OpenNebula [60], Eucalyptus [61] have some sort of network layer separation for different users, user groups or projects. The ZERO protocol is capable of nearly zero overhead Ethernet-over-IP tunneling disregarding the complexity of cloud core networking PDUs. The existing Scientific cloud at IMCS UL [62][63], for instance, uses VLAN tagging to distinguish separate subnets for distinguished user groups. The Ethernet frames that travel on the cloud core network have VLAN tags and hence are larger than in simple LAN environments. This fact has no impact on the size of the tunneled packets that would travel through networks that connect two or more cloud installations, since ZERO protocol would strip away the whole Ethernet header, however large or complex it would be.

Thus, ZERO tunneling protocol may link together not only distinguished subnets, but also core networks of clouds making cloud federation traffic-wise free of tunneling overhead. This also means that high performance cloud interconnections do not need to be dedicated high MTU lines, non-fragmenting tunneling can be achieved on standard MTU Internet connections.

The proposed stream processing system is planned to be implemented in the next generation Scientific cloud at IMCS UL. It is planned to be a true high throughput computing (HTC) system as described earlier with capacity to process data at a rate of 10Gb/s and more. The horizontal flexibility on bare metal layer would be achieved by provisioning computing, I/O and networking resources on demand. At times when demand for cloud applications will be higher than for HTC ones some of the HTC dedicated nodes would reboot into cloud mode and join the cloud resource pool. But when the demand would balance towards the HTC, some of the cloud nodes would reboot into HTC mode, hence shrinking the cloud resource pool and enhancing the HTC pool. Such automation needs to be planned and executed with strict policy and integrity checks. The resource selection and provisioning algorithms of the existing Scientific cloud have been planned and implemented by the author giving notable experience in this field. That research needs to be continued and discussed further.

While ZERO protocol addresses Layer3 fragmentation issues for tunnels over consumer

Internet connections, two research projects have been done by the author and colleagues for the European meteorological union (EUMETSAT) to evaluate the performance of five file transfer protocols (FTP, UFTP, bbFTP, GridFTP, RSYNC) under widely varying conditions characteristic to various WAN scenarios. Namely, performance at 70ms and 700ms RTT typical to intercontinental terrestrial Internet and geostationary satellite communications were studied. Additionally, various packet loss patterns were involved. The designed test lab allowed to find the optimal TCP protocol and file transfer application settings reaching the EUMETSAT set target data rate 350Mbps at 70ms RTT and 10^{-6} packet loss, typical to terrestrial networks. These results also encourage federated and heterogenous cloud systems to be set up even on data centers distributed on global scale. Meanwhile none of the surveyed applications were able to reach the target data rate at 700ms RTT, typical to satellite distribution networks. The results obtained in those studies could be of interest to the much wider audience, as there are much ungrounded myths about the performance of underlying TCP protocol and data transfer applications built on top of it.

References

[1] VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-02 (online, 2012-10-15)

[2] NVGRE: Network Virtualization using Generic Routing Encapsulation, http://tools.ietf.org/html/draft-sridharan-virtualization-nvgre-00 (online, 2012-10-15)

[3] A Stateless Transport Tunneling Protocol for Network Virtualization (STT), http://tools.ietf.org/html/draft-davie-stt-01 (online, 2012-10-15)

[4] RFC2003, IP Encapsulation within IP, http://tools.ietf.org/html/rfc2003 (online, 2012-10-15)

[5] RFC3378, EtherIP: Tunneling Ethernet Frames in IP Datagrams, http://tools.ietf.org/html/rfc3378 (online, 2012-10-15)

[6] RFC2661, Layer Two Tunneling Protocol "L2TP", http://www.ietf.org/rfc/rfc2661.txt (online, 2012-10-15)

[7] OpenVPN, http://openvpn.net/ (online, 2012-10-15)

[8] iptables, a NetFilter project, http://www.netfilter.org/projects/iptables/index.html (online, 2012-10-15)

[9] RFC1702 Generic Routing Encapsulation over IPv4 networks, http://www.ietf.org/rfc/rfc1702.txt (online, 2012-10-15)

[10] ipfw firewall, FreeBSD Handbook, http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/firewalls-ipfw.html (online, 2012-10-15)

[11] Cisco Access Control Lists, CCNA Study Guide, Todd Lammle, ISBN: 0-7821-2647-2

[12] RFC 4447 Pseudowire Setup and Maintenance - Using the Label Distribution Protocol (LDP) , http://www.ietf.org/rfc/rfc4447.txt (online, 2012-10-15)

[13] RFC 4448 Encapsulation Methods for Transport of Ethernet over MPLS Networks, http://www.ietf.org/rfc/rfc4448.txt (online, 2012-10-15)

[14] IEEE Std. 802.1Q-2005, Virtual Bridged Local Area Networks

[15] Carrier Ethernet, http://en.wikipedia.org/wiki/Carrier_Ethernet (online, 2012-10-15)

[16] C. Benvenuti, Understanding Linux Network Internals, O'Reilly Media, 1st edition, 2006

[17] RFC 1027 Using ARP to Implement Transparent Subnet Gateways,

http://www.ietf.org/rfc/rfc1027.txt (online, 2012-10-15)

[18] RFC3514 The Security Flag in the IPv4 Header, http://www.ietf.org/rfc/rfc3514.txt (online, 2012-10-15)

[19] Linux man-pages project, release 3.35, packet (7) function manual, http://man7.org/linux/man-pages/man7/packet.7.html (online, 2012-10-15)

[20] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition), Addison Wesley, 2003.

[21] The Linux Kernel Module Programming Guide, http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html (online, 2012-10-15)

[22] RFC 791, INTERNET PROTOCOL, http://tools.ietf.org/html/rfc791 (online, 2012-10-15)

[23] Linux Source Code, http://lxr.linux.no/#linux+v3.2/include/net/ip.h#L269 (online, 2012-10-15)

[24] Linux Source Code, http://lxr.linux.no/#linux+v3.2/net/ipv4/route.c#L1366 (online, 2012-10-15)

[25] R. Stephens, A Survey Of Stream Processing, University of Surrey, GU2 5XH, 1995

[26] Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, Brian Towles , Stream Scheduling , Stanford University, Stanford, CA 94305

[27] Jayanth Gummaraju and Mendel Rosenblum, Stream Processing in General-Purpose Processors, Stanford University, Stanford, CA 94305

[28] Steven Whitehouse, The GFS2 Filesystem, Red Hat Inc., Proceedings of the Linux Symposium, Volume Two, Ottawa, Ontario, Canada, 2007

[29] Scott Fadden, An Introduction to GPFS Version 3.2, IBM Corporation, 2007

[30] Mark Fasheh, OCFS2: The Oracle Clustered File System, Version 2, Oracle, 2006

[31] Sage A. Weil, CEPH: RELIABLE, SCALABLE, AND HIGH-PERFORMANCE DISTRIBUTED STORAGE, a dissertation, University of California, 2007

[32] Gluster Documentation http://www.gluster.org/community/documentation/index.php, (online, 2012.02.01)

[33] Feiyi Wang Sarp Oral, Galen Shipman, National Center for Computational Sciences, Oleg Drokin, Tom Wang, Isaac Huang, Sun Microsystems Inc., Understanding Lustre Filesystem

Internals, 2009

[34] Bruno G., Stok R., Adventures with clustered filesystems, Inf. Technol. Support Unit, Bank of Italy, 2011

[35] Shinji Sumimoto, An Overview of Fujitsu's Lustre Based File System, Fujitsu Limited, 2011

[36] Jeffrey Dean, Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Google Inc., USENIX OSDI '04: 6th Symposium on Operating Systems Design and Implementation , 2004

[37] Hadoop, http://hadoop.apache.org/, (online, 2012.04.01)

[38] Leonardo Neumeyer, Bruce Robbins, Anish Nair, Anand Kesari, S4: Distributed Stream Computing Platform, Yahoo! Labs, Santa Clara, CA, 2010

[39] Gearman, http://gearman.org/#introduction, (online, 2012.04.01)

[40] Gigabit European Advanced Network Technology (GÉANT) http://www.geant.net, (online, 2012.04.01)

[41] European Grid Infrastructure Integrated Sustainable Pan-European Infrastructure for Researchers in Europe (EGI-InSPIRE) http://www.egi.eu/projects/egi-inspire/, (online, 2012.04.01)

[42] Common Language Resources and Technology Infrastructure http://www.clarin.eu, (online, 2012.04.01)

[43] ELIXIR http://www.elixir-europe.org, (online, 2012.04.01)

[44] The Hidden Costs of Open Source Rethinking the Economics of HPC Infrastructure Software. A Platform Computing White Paper, December 2010. Gord Sissons (gsissons@platform.com) Louise Westoby (lwestoby@platform.com).

[45] Open source software for building private and public clouds. http://openstack.org/, (online, 2012.04.01)

[46] Maarten Steurbaut. The return of Rubik's famous Magic Cube. http://users.skynet.be/maarten.steurbaut/Rubik_Cube.htm, (online, 2012.04.01)

[47] Open source CMS – Drupal http://drupal.org/, (online, 2012.04.01)

[48] Trans-European Research and Education Networking Association (TERENA) Task Force - Computer Security Incident Response Team www.terena.org/activities/tf-csirt, (online, 2012.04.01)

[49] European Middleware Initiative (EMI) http://www.eu-emi.eu, (online, 2012.04.01)

[50] The Cloud Computing Association http://www.cloudcom.org, (online, 2012.04.01)

[51] Asia Cloud Computing Association http://www.asiacloud.org, (online, 2012.04.01)

[52] Stephanie Silvius. Internet Exchange Points. A closer look at the differences between continental Europe and the rest of the world, 2011 www.euro-ix.net/resources/ixp_research.pdf, (online, 2012.04.01)

[53] Business cluster - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Business_cluster, (online, 2012.04.01)

[54] European Strategy Forum on Research Infrastructures. Strategy Report on Research Infrastructures, Roadmap 2010, http://ec.europa.eu/research/infrastructures/pdf/esfri-strategy_report_and_roadmap.pdf, (online, 2012.04.01)

[55] Dr. Beatrix Vierkorn-Rudolph. Towards full utilization of European intellectual potential – ESFRIs strategy for a more balanced landscape of Research Infrastructures in Europe. Stimulating economic and social development: Research Infrastructure development and clusters, 2011 http://www.wire2011.eu/upload/presentations/6/07062011%20-%20WIRE2011%20-%20Beatrix %20Vierkorn%20Rudolph%20-%20Debrecen.pdf.

[56] Lenka Hebakova and Ondrej Valenta. ERAWATCH country report 2010: Czech Republic. Technology Centre ASCR, http://erawatch.jrc.ec.europa.eu/erawatch/opencms/information/reports/countries/cz/report_0006, (online, 2012.04.01)

[57] LinkSCEEM Linking Scientific Computing in Europe and the Eastern Mediterranean. http://www.linksceem.eu/joomla/, (online, 2012.04.01)

[58] HP-SEE. High-Performance Computing Infrastructure for South East Europe's Research Communities. 2011, http://cordis.europa.eu/fp7/ict/e-infrastructure/docs/hp-see.pdf.

[59] Apache CloudStack software, http://cloudstack.apache.org/, (online, 2012.04.01)

[60] OpenNebula – A simple but feature-rich, customizable solution to manage private clouds and datacenter virtualization, http://opennebula.org/, (online, 2012.04.01)

[61] Eucalyptus – Open Source AWS compatible private clouds software, http://www.eucalyptus.com/, (online, 2012.04.01)

[62] *Jaunas paaudzes skaitļošanas infrastruktūras principi* (Next generation computing infrastructure principles), Leo Trukšāns, major thesis, 2008.

[63] E-spiets – Scientific cloud system, http://e-spiets.lv/, (online, 2012.04.01)

[64] RFC3022, Traditional IP Network Address Translator (Traditional NAT), http://www.ietf.org/rfc/rfc3022.txt (online, 2012-10-15)

[65] RFC5128, State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs), http://www.ietf.org/rfc/rfc5128.txt (online, 2012-10-15)

[66] K.Sataki, B.Kaskina, G.Barzdins, E.Znots, M.Libins: BalticGrid-II project final report on Network Resource Provisioning, 2010. URL: http://www.balticgrid.org/Deliverables/pdfs/BGII-DSA2-9-v1-2-FinalReport-IMCSUL.pdf

[67] M.Carbone, L.Rizzo: Dummynet Revisited, SIGCOMM CCR, Vol. 40, No. 2, April 2010.

[68] Lee J., Cha H., Ha R.: A Two-Phase TCP Congestion Control for Reducing Bias over Heterogenous Networks, In: Proceeding of Information networking: convergence in broadband and mobile networking : international conference, ICOIN 2005, Jeju Island, Korea, January 31-February 2, 2005, LNCS Vol.3391, Springer, 2005.

[69] M.Allman, V.Paxson, W.Stevens, RFC 2581: TCP Congestion Control, April 1999.

[70] M.Mathis, J.Mahdavi, S.Floyd, and A.Romanow. RFC 2018: TCP Selective Acknowledgment Options, October 1996.

[71] J. Zhang, R.D.McLeod: A UDP-Based File Transfer Protocol (UFTP) with Flow Control using a Rough Set Approach, submitted to IEEE Transactions on Networking, 2002.

[72] D.Barbosa, J.P.Barraca, A.Boonstra, R.Aguiar, A.Ardenne, J.Santander-Vela, L.Verdes-Montenegro: A Sustainable approach to large ICT Science based infrastructures; the case for Radio Astronomy, Accepted to the IEEE EnergyCon 2014, Croatia 2014, IEEE Xplorer, 2014

[73] S.Bourke, H.J.Langevelde, K.Torstensson, A.Golden: An AIPS-based, distributed processing method for large radio interferometric datasets, Experimental Astronomy, August 2013, Volume 36, Issue 1-2, pp 59-76.