

CESNET Technical Report 9/2009

Enhanced UDP Packet Reflector – *Erum*

IVAN PÚDELKA¹, TOMÁŠ REBOK^{1,2}

¹ Faculty of Informatics, Masaryk University, Brno

² CESNET, z.s.p.o., Praha

Received 11.12.2009

Abstract

This technical report deals with an enhanced unicast packet reflector (called *Erum*), which extends the previous version of the UDP/RTP packet reflector (known as *rum*). Among the main improvements of *Erum* belong the possibilities of dynamic interconnecting the reflectors into a peer-to-peer network supporting the 2D full-mesh communication model, which allows it to provide a communication environment for more clients against the case of using just a single reflector. This technical report describes the design and implementation of such a reflector and demonstrates its usage according to different users' and/or network's requirements.

Keywords: Erum, UDP packet reflector, 2D full-mesh, peer-to-peer reflector network, scalability

1 Introduction

If one needs to establish a communication environment among a group of clients, he or she has a few alternatives. For example, the multicast – a network technology for the delivery of information to a group of destinations using the most efficient strategy (just a single copy of the data through a single link) – could be used. However, this technology is not supported by most of the contemporary networks, and thus a different technology making use of the unicast-based communication has to be used.

An example of such an alternative technology is the UDP/RTP packet reflector (also known as *rum*), which reflects/replicates the traffic being received from each connected client to all the others. Even though the reflector simulates the functionality of the multicast in unicast networks, it cannot ensure that a single copy of particular data will be passed over every link just once. This limitation obviously leads to less efficient behavior in the sense that the reflector can interconnect far less clients than the native multicast can. Nevertheless, besides this limitation, such an alternative approach can provide many new and interesting features as compared to the multicast – for example, an individual approach to every client, video stream composition and/or synchronization, data recording, secure communication individualized for every client, etc.

The less efficient reflector's behavior implies from the fact, that the number of clients it can serve is naturally limited by the throughput of the reflector's output network link(s) – for every client the reflector sends a single copy of the passing data. To cope with that, a peer-to-peer network of reflectors (*peers*) can be

established – the clients could be distributed across the reflectors (each client communicates with just a single reflector) and the reflectors ensure the proper data delivery among themselves (and subsequently, among their connected clients). In this case, a single copy of the data passing between each pair of reflectors is dedicated to multiple clients served by the receiving reflector. Such a reflector network then forms so-called *overlay network* – a virtual network that consists of nodes and logical links, which is designed above existing physical network in order to implement new network services and/or functionalities.

Obviously, such a reflector network should ensure the lowest communication latency possible. Moreover, between each pair of clients, the network should ensure more or less the same communication latency – the network topology itself should not introduce unequal latencies between the clients depending on the part of the network they are connected to. And even further, there are also another needs on the employed communication model (e.g., the robustness in the sense that a link/reflector failure does not affect (or affects minimally) the communication environment), which require sophisticated topologies to be used [5].

In this technical report we present *Erum* – a Java-based version of the original UDP/RTP packet reflector improving the number of servable clients by employing a peer-to-peer network communication model – and present its behavior in the overlay network satisfying the *2D full-mesh topology* [5]. At first, we present and study the performance of a single *Erum* and compare its efficiency with the original C-based reflector. Later, we present an example of an established network of *Erums* and study its practical behavior in the face of maximum number of attachable clients.

2 *Erum* Design

This section presents *Erum* (for the sake of simplicity, later depicted as reflector as well) and its behavior from a high-level point of view; a more precise description of its architecture and implementation details can be found in the following section.

Since *Erum* extends the original reflector (*rum*), it has to be able to operate independently in the network in the same way as *rum* does. However, as soon as there are more *Erums* in the network, it must be able to cooperate with them, to form the 2D full-mesh topology, and to ensure the proper data delivery to all the connected clients.

The process of forming the 2D full-mesh topology proceeds as follows: once a new *Erum* appears in the network, it contacts any *Erum* being already connected to the reflector network, which subsequently takes care of introducing it to all the other members of the network (details are provided in Section 3.1).

As soon as the 2D full-mesh topology is formed (see an example in Figure 1), the clients are allowed to connect the network. During the connections, the network tries to uniformly distribute the clients over the network in order to distribute the network load introduced by the passing/reflecting data. This distribution allows to satisfy the primary *Erum*'s goals – to maximize the number of servable clients and to minimize the communication latency among the communicating clients.

Every connected client communicates with just a single *Erum* (the one which it

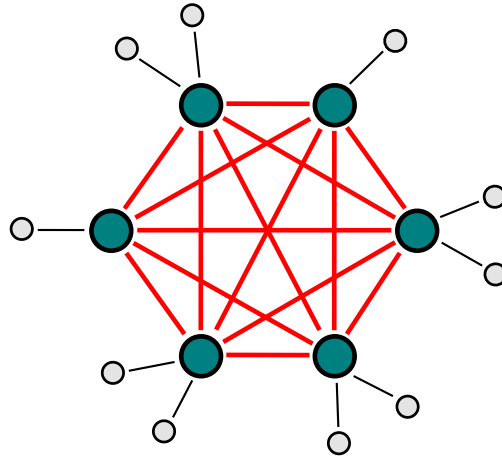


Figure 1. 2D full-mesh topology (red lines) in the network consisting of 6 *Erums* and 10 clients.

becomes informed about as soon as it becomes connected) – the client sends data to that reflector, which simultaneously forwards them to all its other clients (i.e., without the sending one) and to all *Erums* in the network. As soon as the other reflectors receive the data, they forward them to all the clients they serve. Nevertheless, to overcome reflectors’ failures, every client is also periodically informed about all the reflectors in the network, so that it is able to connect to another one if a reflector failure appears.

All the reflectors as well as all the connected clients communicate both via UDP (*User Datagram Protocol*) and TCP (*Transmission Control Protocol*) transmission protocols. While the UDP is dedicated for data transmissions only, the TCP is used for control communications among all the reflectors and connected clients – for example, a registration information about new reflectors and/or clients, errors’ notifications, an information about clients’ re-distributions, etc.

This requires the clients to be enriched with a control service in order to be fully usable in the network of *Erums* – at least, such a service has to be able to perform and answer the requests from the reflector network. For the clients not easily enrichable with such a service, a general service can be used – such a service may run on the client’s machine and may take care of performing and answering the requests from the reflector network. In this case, the particular client does not directly communicate with the reflectors, but it communicates with this service, which forwards/receives the data to/from a relevant reflector this client should be connected to.

3 Architecture and Implementation Details

The basic functionality of both the original and enhanced reflectors is to replicate traffic being received from every connected client to all the others. This functionality should be ensured on the userspace level, i.e., the reflectors should be able to operate without any administrative privileges both to the host machine and to the network. To provide *Erum*’s independence on the hosting system preserving the mentioned features, we have decided to implement it in the Java language (Java

Standard Edition 6¹).

Regarding *Erum*'s data replicating part, our first attempt was to follow the idea used in the first Java-based *rum* implementation (created by Petr Holub) – two individual threads, one of which receives the packets from the network saving them to a buffer, while the other reads the packets from the buffer and forwards them to appropriate clients. However, because of an overhead introduced by the JVM (*Java Virtual Machine*), the sending thread was unable to achieve the throughput we needed. This problem was overcome by multiplying the sending threads, which, however, introduced some packet reordering in the output network stream(s) and thus became unusable as well.

Thus, after all these attempts we have decided to implement the *Erum*'s data replicating part as a single thread (denoted as *Mirror* in the rest of this report), which has provided the best results regarding all the basic indicators – the maximal throughput, minimal latency, and minimal jitter.

Besides the data replicating part, *Erum* consists of another two important parts: *Reflector-Admitter* thread – the part communicating with another *Erums* in the network, which is mainly responsible for forming the 2D full-mesh topology, connecting/disconnecting the reflectors, re-distributing the clients, etc., and *Client-Admitter* thread – the part communicating with the end clients, which admits or rejects clients' requests for joining the network. See the Section 3.4 for further details.

3.1 Reflectors Management

As already depicted, *Erum* can operate in both an independent and networked mode. If an *Erum* is intended to join an existing reflector network, it contacts an already connected reflector and asks it for joining the network. If allowed, it is provided with a list of reflectors already forming the network by the contacted reflector, which simultaneously informs all the other members about the new peer. As soon as a new reflector joins the network, all the already connected clients are redistributed in order to uniformly distribute the load over all the members of the network (see details in Section 3.2).

Besides that, all the reflectors are making sure all the others about their availability (by sending keep-alive messages). Once an *Erum* does not announce itself for a specified time period, it is considered to become disconnected from the network – every *Erum* erases it from its reflector peers list.

3.2 Clients Management

If a client wants to use the communication environment provided by *Erums*, it has to connect a single *Erum* in order to announce itself's interest. No matter which reflector is chosen, the network itself takes care of choosing the proper *Erum* for the particular client – for all the clients, this guarantees as powerful communication environment as possible.

From the startup, the reflector network itself tries to distribute the clients in

¹ <http://java.sun.com/javase/6/docs/api/>

order to distribute the network load – each reflector serves n or $n - 1$ clients. Thus, once a new client appears, the asked reflector tries to forward it to the one(s) serving the lowest number of clients, where the process of connecting repeats; if all the reflectors serve the same number of clients, the client is accepted on the asked reflector. Once a proper reflector is known and the client is connected, it can start sending/receiving the data.

If a client becomes inactive for a specified time interval, it is automatically purged from the network. Let us point out, that in such a case, no clients' re-distributions occur – the main goal of such a reflector network is not to balance the load (even if it tries to perform it in special cases²), but to maximize the number of servable clients and to minimize the communication latency among them. Since both goals keep satisfied in such a case, no re-distributions are necessary.

3.3 *Erum* Startup

From the command line, *Erum* could be started as follows:

```
java -jar rum.jar [-rtp] port ipaddress [-ref reflectorAddress reflectorPort]
```

The parameter `-rtp` is optional. If present, *Erum* supposes the data to be sent in accordance with the RTP³ protocol format – the parameter makes *Erum* to listen and operate on both the specified port and the `port+1` network ports. The `ipaddress` parameter, which is mandatory as well, specifies the interface, which the reflector should listen on (defined by its IP address or hostname). The other optional parameter – `-ref reflectorAddress reflectorPort` – specifies the IP address/hostname and the port number of the reflector, which the starting one should connect to (in the case of joining an already established reflector network). In the case of more reflectors already forming the reflector network, any of them may be chosen.

For the proper functionality, *Erum* needs two another ports serving as the control ones – so-called `purgerPort` and so-called `controlPort` (see details later in this section). To simplify the communication process, these ports are chosen as `port+2` (the `purgerPort`) and `port+3` (the `controlPort`), where `port` is the port number specified on the command line.

3.4 *Erum* Operation and Threads in Detail

As soon as the reflector is started, the `Main` class checks the correctness of entered parameters. If the reflector is intended to become a member of an existing reflector network, it subsequently contacts the reflector specified on the command line asking it for joining the network. As already depicted, if allowed, it is provided with a list of reflectors already forming the network, so that it can monitor the network as well as provide the list of reflectors to its clients.

In the list, every reflector is represented as an object having specified its network address and port number. Besides this, there is one another list maintained by

² The clients are more or less uniformly distributed across the reflectors from the network startup and after new reflector's join – see Section 3.1.

³ *Real-time Transport Protocol* [3].

every *Erum* – the `clientList`, which serves for maintaining the clients connected to the particular reflector. Every client in the `clientList` is then represented as an object having specified its network address, port number, and last communication time, which is used for purging inactive clients.

Erum's operation is performed by several threads running simultaneously:

Mirror

This thread receives UDP datagrams from the network and sends them to the reflector's peers/clients. The sending process itself depends on the data sender – if the data have been sent by another reflector (peer), they are forwarded to all the clients only, while if the data have been sent by reflector's client, they are forwarded to all the reflector's clients (except the sending one) and to all the reflectors (peers) forming the reflector network. Simultaneously, the last communication time for the sending client is appropriately updated.

ReflectorAdmitter

This thread serves for admitting new reflectors as well as for communications among all the reflectors/peers in the network. For the communication, it uses a TCP channel created across the `controlPort` – when a connection with another reflector (initiator) is established, an object representing the initiator is received. Once received, this thread checks whether the initiator is already a member of the network (by checking the `reflectorList`).

If not, the initiator is a new reflector trying to join the network – it is provided with the `reflectorList` (all the members of the reflector network) and all the other peers, in fact their `ReflectorAdmitter` threads, are informed about the new member. Simultaneously, the contacted reflector initiates the process of discovering the number of all the connected clients in the network, which is later provided to all the other peers as well. Using this information together with an information about the number of all the reflectors in the network, all the members are able to compute the number of clients they should serve, and forward the additional clients to less used reflectors (load re-distribution, minimizing the communication latency).

ClientAdmitter

The main purpose of this thread is to make decisions about admitting new clients to the particular *Erum* or to the whole network. Once a client contacts a reflector (in fact, this thread) through the `purgerPort` (via TCP), the main goal of this thread is to make a decision, whether the new client can be admitted to this *Erum*, to another *Erum* in the network, or not admitted at all (too many clients in the network). Besides that, this thread is also used for another necessary control messages, namely:

- **keep-alive messages** – there is no reaction invoked after receiving this message – as already mentioned, these messages serve for notifying all the peers about reflector's availability.
- **number-of-your-clients** – once this message is received, the reflector sends the number of its connected clients to the message initiator (another reflec-

tor performing the process of computing the number of clients connected to the network).

- **i-am-new-client** – this message has to be sent by a new client before it is allowed to send data to the reflector. In consequence, the number of clients served by all the reflector peers is discovered, and the new client is forwarded to the reflector serving the lowest number of clients (or accepted on the asked reflector, if all the reflectors have the same number of clients).

4 *Erum* Functionality and Performance Analysis

4.1 Testbed

We have tested *Erum* functionality and performance by a set of various tests. The network environment used during the tests is depicted in Figure 2 – it consisted of a cluster of common PCs interconnected via the *HP Procurve 5406zl* network switch. The nodes in the cluster have had the configuration summarized in Table 1.

Table 1. Hardware and software configuration of the used nodes.

	Configuration
Processor	2× Dual-Core Xeon 5160 3.0 GHz
Memory	4 GB
1 GE NIC	Broadcom Corp. NetXtreme BCM5703X (rev 02)
Operating system	Debian 4.0 (Etch) kernel 2.6.22.17-0.1-xen SMP x86 64

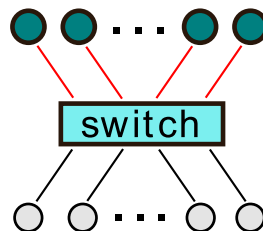


Figure 2. The network architecture used during the tests.

To measure the throughput, latency (average time delay) and packet loss, the tool called *generator7* has been used. The *generator7* tool is a UDP packet generator and receiver based on our previous implementation of the *RTPgen/RTPsink* toolset [2] – *generator7* uses RTP-like time-stamps and sequence numbers to measure the bandwidth, packet loss, latency, and jitter of the communication (the jitter is measured according to the methods described in [3]). However, the tool had to be slightly modified in order to be able to communicate with *Erums* – it had to be enriched with a layer performing the TCP control communications and managing the reflectors, which the generated data have to be sent to.

For the scalability and clients’ redistribution tests, 30 Mbps UDP streams with 1440 B packets have been used. The streams have simulated a videoconference employing the digital video (DV) [1] streams – an example of the typical applications

Erum could be used for. Last, but not least, all the streams have been generated and analyzed on the clients' sides.

4.2 Throughput tests

The main goal of this test is to determine the maximum throughput *Erum* is able to achieve. The importance of this test lies in the fact, that the maximal achievable throughput limits the number of the clients making use of the reflector network – see the scalability tests later in this section.

Erum's performance is further compared to the performance of the original C-based *rum* in order to find out the performance degradation introduced by the Java overheads.

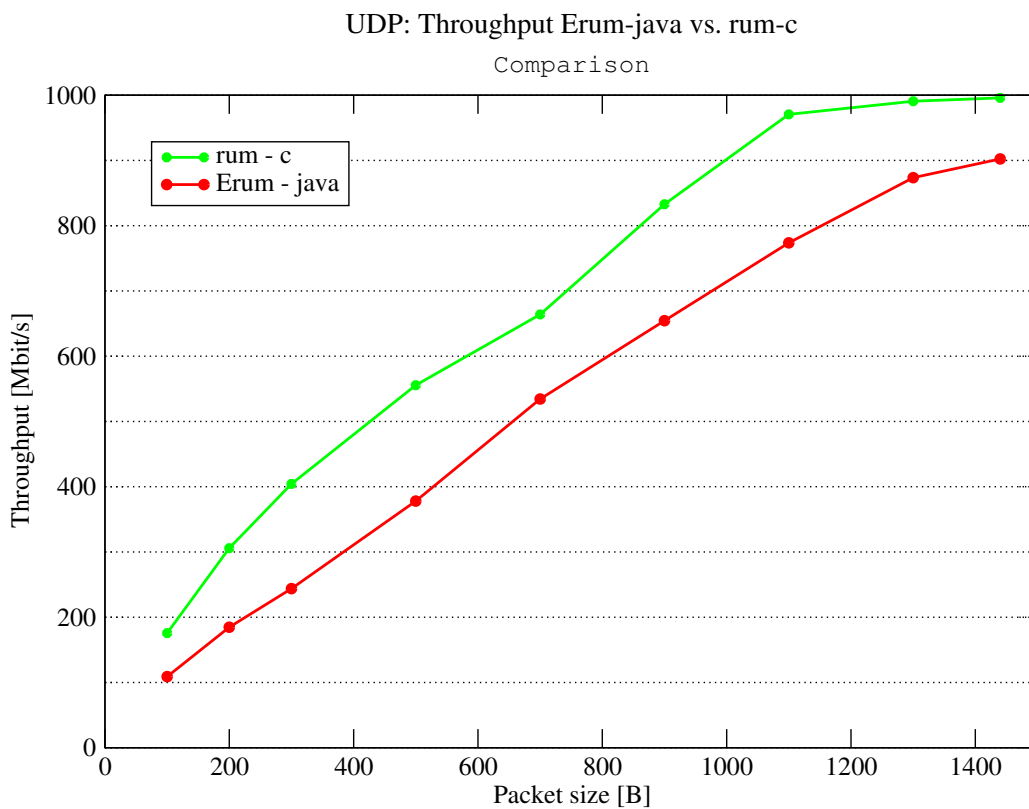


Figure 3. Maximal achievable throughputs – Java-based *Erum* vs. C-based *rum*.

The achieved results are depicted in Figure 3. *Erum*'s throughput, which obviously increases with increasing packet size, achieves its maximum of 905 Mbit/s for 1440 B packets. The graph also depicts the expected performance degradation introduced by the JVM, which is approximately 9.5 percent.

4.3 Scalability tests

The scalability tests intend to determine the total number of clients that can simultaneously make use of the reflector network of a specific size. During the tests, we have created a reflector network consisting of the particular number of *Erums* and have observed the maximal number of clients, that could be interconnected. Each connected client has been sending a 30 Mbit/s data stream (consisting of 1440 B packets) and has been receiving the data streams of all the other clients simultaneously.

4.3.1 Theoretical scalability analysis

Nevertheless, before the practical measurements, we have performed a theoretical scalability analysis [4], [5] as well. Let's assume a 2D full-mesh network consisting of m *Erums* (as shown in Figure 1) having exactly n connected clients. In an ideal case, these clients are distributed in the way that every *Erum* serves n_r or n_{r-1} clients:

$$n_r = \left\lceil \frac{n}{m} \right\rceil. \quad (1)$$

Since every *Erum* receives at most as many streams as the number of reflector network's clients is (exactly one stream from each client), the inbound load *in* (the number of inbound streams the reflector has to serve) of the reflector having n_r clients is

$$in = n. \quad (2)$$

The maximal outbound load (again, the number of outbound streams), which limits the number of the reflector network's clients, then applies for the reflectors having n_r clients⁴ and can be computed as (s is the number of *Erum*'s clients):

$$out = s \cdot ((m - 1) + (s - 1)) + s \cdot (n - s). \quad (3)$$

where the first part of the formula ($s \cdot ((m - 1) + (s - 1))$) denotes the fact that every reflector client's stream has to be forwarded to all its other clients (whose count is $s - 1$) and to all the other reflectors in the network (whose count is $m - 1$). The second part of the formula denotes the fact, that the particular reflector has to forward all the other clients' incoming streams (whose count is $n - s$) to all its clients (whose count is s).

Since s takes the value of n_r or n_{r-1} according to the number of particular reflector's clients, and since the reflectors having n_r represent the network's limits, the formula (3) representing the maximal outbound load can be rearranged into the formula

$$out = n_r \cdot (m + n - 2). \quad (4)$$

For example, let's assume the following situation: there is a reflector network consisting of 4 *Erums* and 8 clients. In this case, $n_r = \lceil 8/4 \rceil = 2$. According to the analysis, we can compute the number of output reflector streams as

$$out = n_r \cdot (m + n - 2) = 2 \cdot (4 + 8 - 2) = 20. \quad (5)$$

⁴ The reflectors having n_{r-1} do have the outbound load lower.

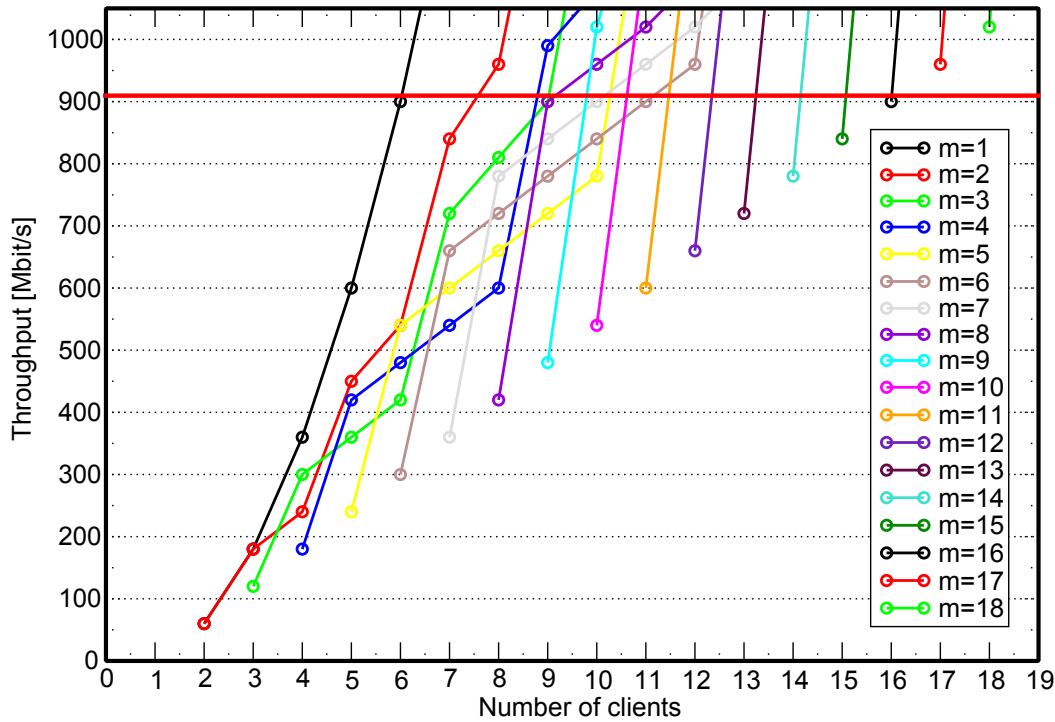


Figure 4. The theoretical throughputs of a single reflector (m indicates the number of *Erums* in the network).

Once using the streams of 30 Mbps (as we expected in our tests), such a reflector has to serve the outbound traffic of $out = 20 \cdot 30 = 600$ Mbps.

The results of the performed analysis are summarized in Figure 4 – the graph shows the maximal output network traffics of the reflectors in the reflector network for the particular number of interconnected clients. Besides that, the limiting value indicating the maximal *Erum*'s throughput measured (905 Mbit/s) is depicted as well.

4.3.2 Practical scalability analysis and measurements

However, the theoretical analysis results do not exactly correspond to our practical measurements, since, during the measurements, a single client has been used as a receiver/analyzer only (it has not sent any data)⁵. Thus, we have adapted the analysis to conform with these conditions – see the graph in Figure 5.

Under these circumstances, the goal of the performed practical measurements was to prove/disprove the reflector network's scalability analysis under a real network conditions. During the tests, we have been monitoring several important char-

⁵ The theoretical analysis depicted in Figure 4 assumes all the clients to operate both as senders and receivers simultaneously.

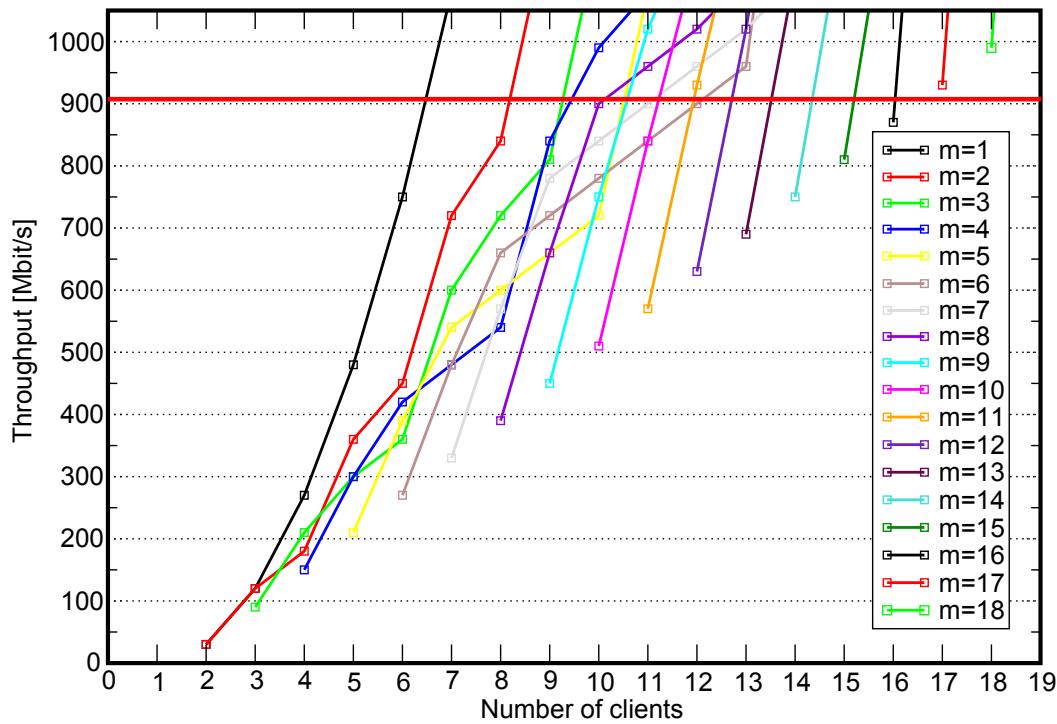


Figure 5. The adapted theoretical throughputs (conforming to practical measurements) of a single reflector (m = number of *Erums*).

acteristics indicating the reflectors network behavior – the latency (average time delay), the average packet loss and the throughput. Every measurement has lasted for 60 seconds and has been repeated 3 times. The presented values represent the computed average values of all the three measurements.

The performed practical measurements have confirmed the adapted theoretical assumptions summarized in Figure 5 – see a few boundary measurements depicted in Figures 7, 8, 9 and 10. Thus, let us summarize the maximal numbers of servable clients by a reflector network of a specific size – see Table 2.

Table 2. Measured maximal number of clients in the reflector network of a specific size.

# <i>Erums</i>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Max. clients	6	8	9	9	10	12	11	10	10	11	11	12	13	14	15	16

Note that increasing the size of the reflector network sometimes does not lead to an increase of the number of servable clients (e.g., from 3 to 4 *Erums*, from 8 to 9 *Erums*, etc.). Even further, increasing the size of the reflector network both from 6 to 7 *Erums* and from 7 to 8 *Erums* does lead to decreasing the total number of servable clients. These situations are caused by the saturation of the network by the data

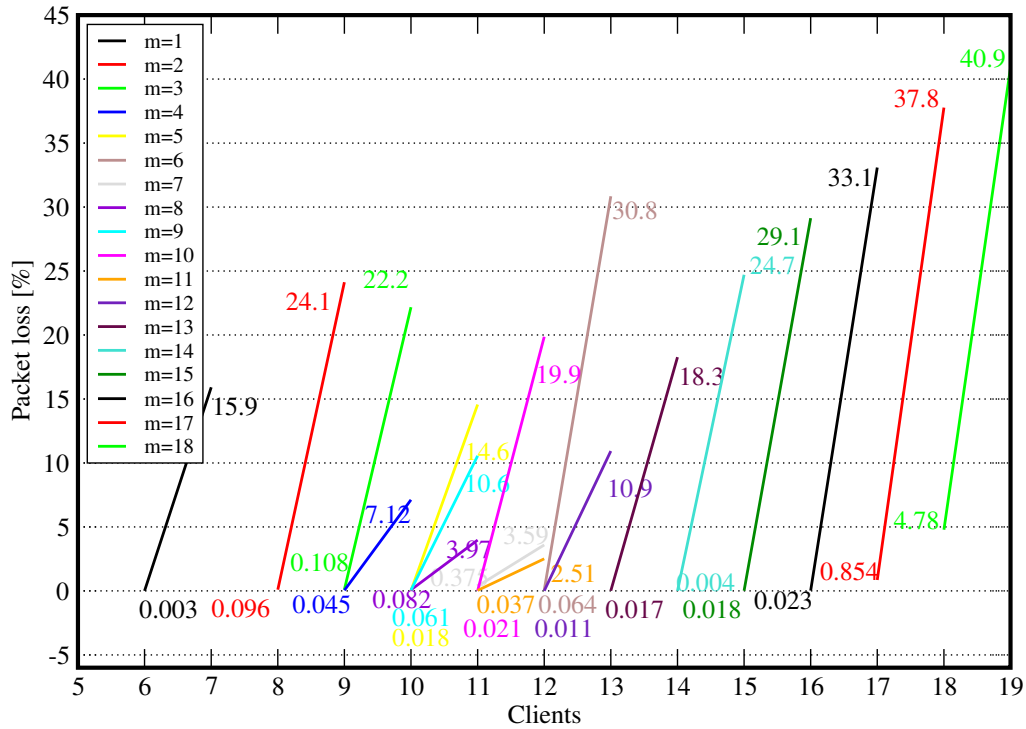


Figure 6. Packet loss summary (m – the number of Erums in the network)

being exchanged among the reflectors – because of the 2D full-mesh topology, once the number of *Erums* increases, each of the original ones has to additionally forward the data of all its clients to one another peer (to the new one), which obviously increases its output load – the limiting factor influencing the number of clients, which the particular *Erum* is able to serve.

As the boundary graphs indicate, connecting more clients than stated have obviously led to an increased packet loss⁶ and communication latency – see Figure 6 summarizing the losses for all the boundary measurements.

Nevertheless, these results correspond to the adapted analysis, which supposes one client acting as a data receiver only. Thus, let us summarize the maximal numbers of servable clients by a reflector network of a specific size, where all the clients act as data senders and receivers simultaneously – see Table 3.

Table 3. The maximal number of clients in the reflector network of a specific size.

# Erums	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Max. clients	5	7	8	8	9	11	10	9	9	10	11	12	13	14	15	16

⁶ For the successful measurement, we have tolerated the average packet loss below 1%.

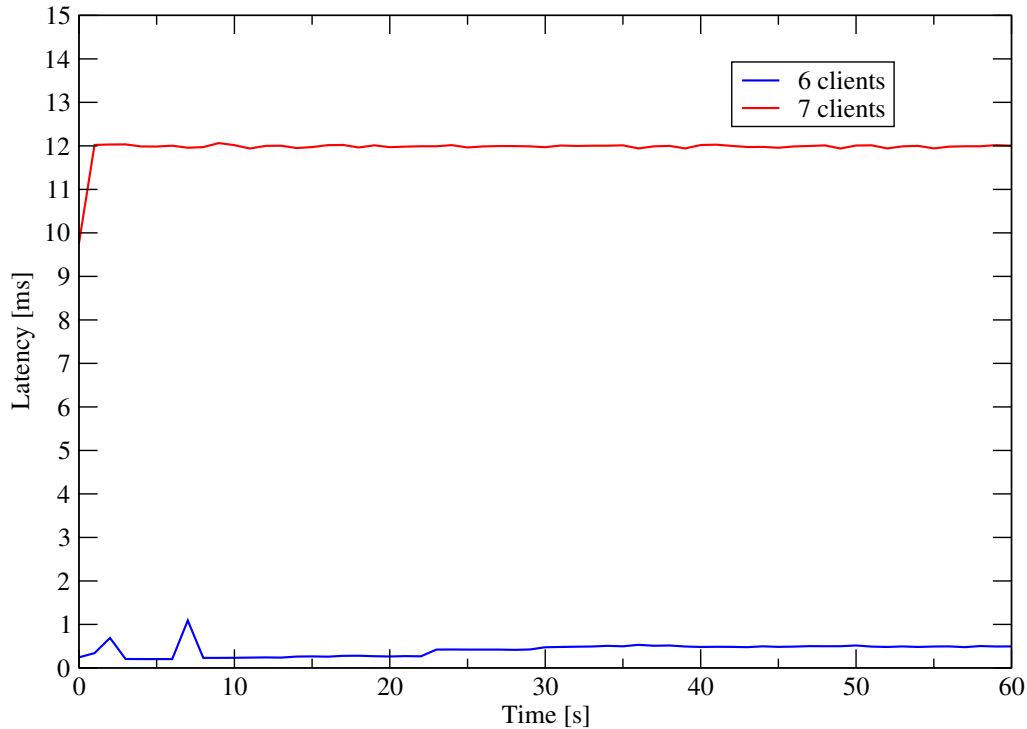


Figure 7. Packet loss and latency for 1 *Erum*.

4.4 Clients' redistribution tests

Since the reflector network performs clients' re-distributions in the case of connecting new *Erum*s, it is necessary to have an idea how much such a migration affects migrated client's communication. Thus, we have performed a set of tests focusing on determining these influences.

We have set up a reflector network consisting of a defined number of *Erum*s and clients. After that, a new *Erum* has joined the network, which has caused clients' re-distributions. We have tested reflector networks of different sizes; the migration time was determined as the average migration time of all the migrated clients. Every test was performed 3 times and the presented values represent the average values measured.

After the migrations, all the clients were redistributed correctly and able to communicate. The migration time of a single client was about 25-85 ms without any dependency on the reflector network size – see the following table, which depicts several measured migration times for the particular reflector network size (before connecting the new reflector) and particular number of connected clients.

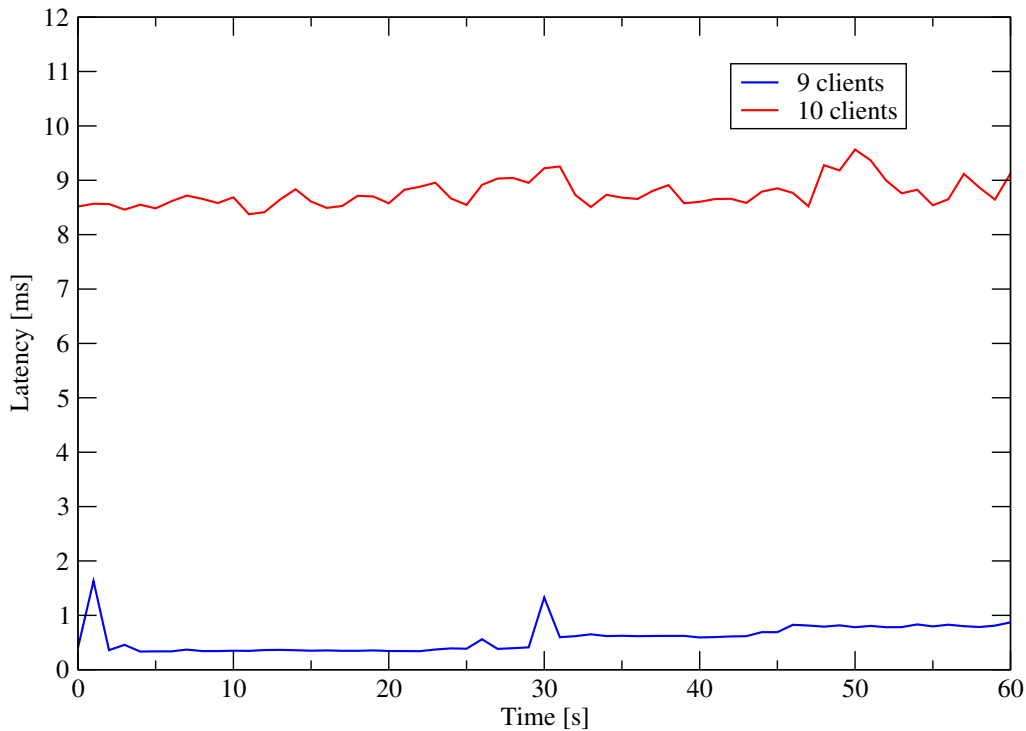


Figure 8. Packet loss and latency for 4 *Erums*.

5 Conclusions

In this technical report we have presented the enhanced unicast packet reflector (called *Erum*). We have described its design and basic functions including the possibilities of interconnecting *Erums* into a peer-to-peer networks serving as a communication environment for higher number of clients than in the case of a single reflector. By employing the 2D full-mesh communication model such a reflector network further provides a communication environment with minimal communication latency possible (the data packets traverse the minimal number of reflector nodes).

Moreover, the basic idea behind the original *rum* – user-empowered-ness and its independence on any specific network features except for simple unicast routing – remained unaffected. Moreover, since *Erum* is implemented using the Java language, there is no need for using any specialized network components in the systems it has to run in.

The measurements, which we have performed, have proven that the scalability of the network of *Erums* corresponds to the theoretical expectations. Even though the JVM introduces some performance degradation (about 9.5 percent based on our measurements), the performed tests have shown that the idea behind *Erum* is fully usable and applicable in real networking environments supporting users' collaboration.

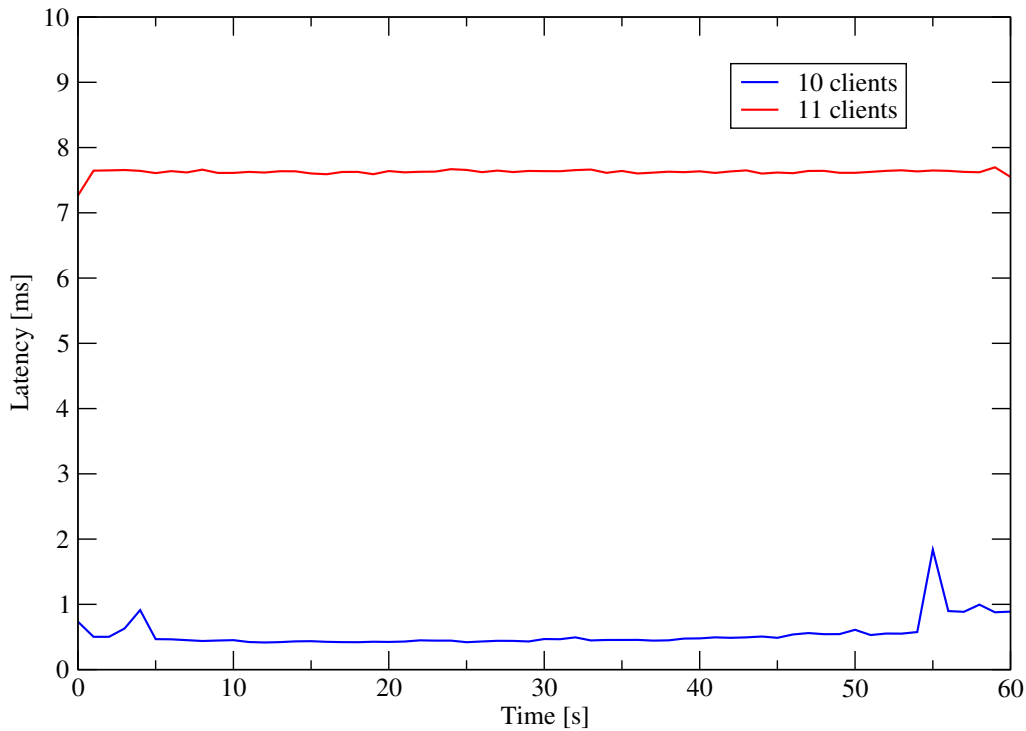


Figure 9. Packet loss and latency for 8 *Erums*.

Currently *Erum* supports 2D full-mesh communication model only. For the future we would like to implement the 3D layered-mesh model described in [5] as well. This model creates k layers, where each of them is similar to the 2D full-mesh model except for the fact that just a single *Erum* of the particular layer both sends and receives the data. The other *Erums* in the particular layer are used just for data reception.

Nowadays, we are implementing secure control communications both between *Erums* and their clients and among *Erums* themselves. Moreover, the implemented security mechanisms should ensure the safeness of the whole reflector network so that unauthorized *Erums*/Clients are unable to connect it and affect the ongoing communication.

For the future we would also like to implement *Erum* in C language in order to overcome the performance overheads introduced by the JVM.

Acknowledgement

This work has been supported by the research intent *Optical Network of National Research and Its New Applications*, MŠM 6383917201, funded by the Ministry of Education, Youth and Sports of the Czech Republic. Moreover, we would like to express our deep thanks to the METACentrum project⁷ for enabling us to use its

⁷ <http://meta.cesnet.cz/cms/opencms/en/index.html>

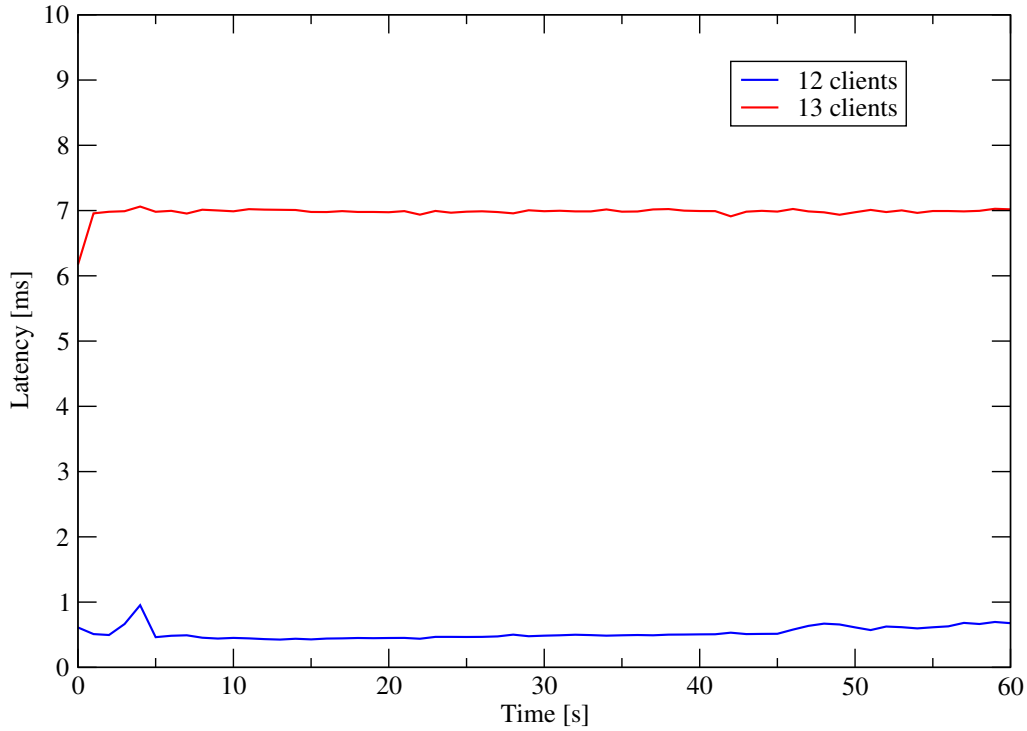


Figure 10. Packet loss and latency for 12 *Erums*.

Table 4. The average migration times for the reflector network of a specific size (before connecting the new reflector) and for the particular number of connected clients.

# Erums	# clients	# migrated clients	average migration time
1	4	2	33 ms
2	7	3	65 ms
4	8	4	57 ms
5	9	4	48 ms

network and computing resources in order to perform all the necessary measurements.

References

- [1] KOBAYASHI, K.; OGAWA, A.; CASNER, S.; BORMANN, C. *RTP Payload Format for DV (IEC 61834) Video*. RFC 3189⁸, IETF, January 2002.
- [2] HLADKÁ, E. *User Empowered Collaborative Environment: Active Network Support*. PhD Thesis, Brno: Masaryk University, 2004.

⁸ <http://tools.ietf.org/html/rfc3189>

- [3] SCHULZRINNE, H.; CASNER, S.; FREDERICK, R.; JACOBSON, V. *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889⁹, IETF, January 1996.
- [4] PÚDELKA, I. *Data Models for Reflector Network*. Bachelor Thesis, Brno: Masaryk University, 2008.
- [5] HOLUB, P. *Network and Grid Support for Multimedia Distribution and Processing*. PhD Thesis, Brno: Masaryk University, 2004.

⁹ <http://tools.ietf.org/html/rfc1889>