

# Performance Analysis of Chainsaw-based Live P2P Video Streaming

P.B.J. Duijkers



Delft University of Technology



# Performance Analysis of Chainsaw-based Live P2P Video Streaming

Master's Thesis in Computer Science

Parallel and Distributed Systems Group  
Faculty of Electrical Engineering, Mathematics, and Computer Science  
Delft University of Technology

P.B.J. Duijkers

December 5th, 2008

**Author**

P.B.J. Duijkers

**Title**

Performance Analysis of Chainsaw-based Live P2P Video Streaming

**MSc presentation**

December 12th, 2008

**Graduation Committee**

prof. dr. ir. H.J. Sips (chair) Delft University of Technology

ir. dr. D.H.J. Epema Delft University of Technology

ir. J.J.D. Mol Delft University of Technology

dr. M.M. de Weerd Delft University of Technology

## **Abstract**

Due to the growing popularity of viewing media over the Internet, content servers are suffering from more and more stress every day. This problem is traditionally solved by enhancing the server infrastructure at the content provider, which is effective but also costly. A more cost effective solution would be to use P2P technology to distribute the media stream in real-time. For this purpose, the Chainsaw algorithm has been proposed, which performs very well in simulations. However, Chainsaw has not been implemented in a real video player yet. We have built our own version of Chainsaw called Kettingzaag, and we have added some improvements and features which make it more resilient to errors, such as multiple description coding. Kettingzaag is put to the test in our own video player called Lumberjack, on the DAS-3 supercomputer in Delft. Our experiments show that the Kettingzaag algorithm performs well for network sizes up to a hundred nodes, and is likely to perform just as well for larger network sizes.



# Preface

This thesis is the result of my graduation project on real-time video streaming. This project is done in the context of the I-Share project, which is a joint effort between the Universities of Technology in Delft, Twente and Eindhoven, the Vrije Universiteit in Amsterdam, and Philips. One of the goals of I-Share is to develop a P2P system which can stream video in real-time over the Internet.

I did this graduation project at the Parallel and Distributed Systems group, of which I would like to thank my advisors, ir. dr. D.H.J. Epema, and J.J.D. Mol. Furthermore, I would like to thank my parents and girlfriend for their patience and support during my thesis project.

P.B.J. Duijkers

Delft, The Netherlands  
5th December 2008



# Contents

<b>Preface</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 I-Share . . . . .	1
1.2 Content Distribution . . . . .	2
1.3 Modular Player Design . . . . .	2
1.4 Problem Statement . . . . .	3
1.5 Thesis Outline . . . . .	3
<b>2 The Design of Kettingzaag</b>	<b>5</b>
2.1 Three Approaches to Live Video Streaming . . . . .	5
2.1.1 Traditional Approach . . . . .	6
2.1.2 Network Level Multicast . . . . .	6
2.1.3 Application Level Multicast . . . . .	6
2.2 Multiple Description Coding . . . . .	8
2.3 The Chainsaw Overlay Network . . . . .	9
2.4 The Kettingzaag Overlay Network . . . . .	11
<b>3 The Delft-37 Testbed</b>	<b>15</b>
3.1 Introduction to Delft-37 . . . . .	15
3.2 The Delft-37 Controller . . . . .	16
3.2.1 Executing Commands on Delft-37 . . . . .	17
3.2.2 Copying Files to the Delft-37 Network . . . . .	18
3.2.3 Copying Files from the Delft-37 Network . . . . .	18
3.2.4 Delft-37 Network Measurements . . . . .	18
3.3 Delft-37 Infrastructure . . . . .	18
3.3.1 Problems . . . . .	19
3.4 Delft-37 Network Performance . . . . .	21
3.4.1 Ping Tests on Delft-37 . . . . .	22
3.4.2 Bandwidth Tests on Delft-37 . . . . .	22
3.5 Conclusion . . . . .	24

<b>4</b>	<b>Testing Lumberjack</b>	<b>25</b>
4.1	System Parameters . . . . .	25
4.2	Measurements . . . . .	27
4.2.1	Basic Functionality . . . . .	27
4.2.2	User Behaviour . . . . .	27
4.2.3	Basic P2P Behaviour . . . . .	28
4.2.4	Video Behaviour . . . . .	29
4.3	Performance Metrics . . . . .	30
4.4	Test Set 1 . . . . .	31
4.4.1	Test Setup . . . . .	31
4.4.2	Unlimited Upload Results . . . . .	32
4.4.3	Limited Upload Results . . . . .	33
4.5	Test Set 2 . . . . .	34
4.5.1	Test Setup . . . . .	34
4.5.2	Test Results . . . . .	35
4.6	Conclusion . . . . .	38
<b>5</b>	<b>Conclusions and Future Work</b>	<b>39</b>
5.1	Conclusions . . . . .	39
5.2	Future Work . . . . .	40
<b>A</b>	<b>Delft-37 Test Results</b>	<b>45</b>
<b>B</b>	<b>Lumberjack Test Results</b>	<b>53</b>
B.1	Lumberjack Unlimited Client Upload Results . . . . .	53
B.2	Lumberjack Limited Client Upload Results . . . . .	67

# Chapter 1

## Introduction

Since the introduction of peer-to-peer (P2P) networks, downloading has become increasingly popular. The first P2P networks were primarily used for music and compressed video files. Today's networks allow users to download large files such as DVD and game images within a few hours. The main reason for the popularity of P2P is that the system overcomes bandwidth limitations of a single node at almost zero cost. This has inspired many people to share their music and video collections on the Internet. Unfortunately, most of these music and video files are copyrighted, and so P2P has become a synonym for illegal downloads. However, more and more people are willing to watch legal content online, such as the news and TV programs. A recent development that has become very popular are the so called *Broadcast Yourself* websites, which allow users to share their videos online with other people. A well known example is *youtube.com*, and a less known Dutch variant is *123video.nl*. One of the main reasons why these sites are so popular is the click-and-watch experience. Most P2P systems do not yet provide real-time streaming, and therefore content providers still use the client/server model. This means that each client downloads its entire stream from the server, so the bandwidth stress and cost at the server grows linearly with the number of clients. Recently, Ellacoya Networks has performed a study [6] on the Internet usage data of approximately one million north-american broadband subscribers. This study shows that traffic over HTTP has been increasing in the past years, and now consumes 46% of the total bandwidth, against 37% for P2P. This makes HTTP bigger than P2P for the first time in four years. The main reason for this shift is real-time video streaming over HTTP. With the increasing demand of real-time media, server costs will get even higher in the future. So, the logical next step is to extend P2P technology with real-time media streaming.

### 1.1 I-Share

The Universities of Technology in Delft, Twente and Eindhoven, the Vrije Universiteit in Amsterdam, and Philips participate in the I-Share [7] project, which is a

research project on sharing resources in virtual communities. The underlying idea is that people participate in a group, and help other group members to benefit from the group's resources by sharing bandwidth, storage and CPU cycles. Setting up algorithms for resource sharing is not simple because of three major issues. First, finding available resources at other nodes is not easy. Second, nodes that have limited capabilities or do not share their resources degrade overall system performance. And third, it is hard to determine which nodes can be trusted. Therefore, I-Share research concentrates on mechanisms for *resource discovery*, *willingness*, *trust*, and *resource sharing*. One of the major achievements in the I-Share project is Tribler [9, 22], a BitTorrent-compatible P2P client which implements most of the above mechanisms. For example, Tribler allows users to setup virtual communities by adding trusted nodes which may help with a torrent download by sharing unused bandwidth [9]. Although many more features have been implemented in the Tribler client, it has no real-time streaming features yet.

## 1.2 Content Distribution

So, how to setup a P2P system that handles real-time media streaming? The answer to this question is not trivial. One of the most difficult aspects is to create a good content distribution algorithm (CDA) for video. Although many CDAs exist, they can all be categorized into three main categories. The first category consists of tree-based CDAs [4, 13, 19]. The big advantage of tree-based CDAs is that their node addition and routing schemes are fairly easy. However, tree recovery can be both difficult and time consuming when multiple nodes in the network fail. Furthermore, leaf nodes waste bandwidth because they do not upload to any of the other nodes. The second category is made up by flooding-based CDAs [20, 21]. The advantages and disadvantages of these CDAs are similar to the tree-based versions. Although failing nodes are less likely to split up the network, flooding-based CDAs suffer from higher end-to-end delays than tree-based solutions when the number of nodes becomes large. The final category contains the swarm-based CDAs [12, 25, 27]. The main advantage of swarm-based overlays is that they do not enforce a strict network topology. Since every node connects to multiple neighbours, overlay recovery problems are reduced when nodes fail. However, packet routing is far from trivial. One of the first swarm-based algorithms is Chainsaw [25]. Although there is not much practical experience with Chainsaw yet, its theory and the first test results reported in [25] look promising. Therefore, we have chosen to base our own CDA on the ideas of Chainsaw.

## 1.3 Modular Player Design

Live P2P video players can be very different in design, but they always share some common functionality. Every system needs a video encoder module that generates a stream of packets at the server side. Furthermore, the system needs a network

module that connects nodes to each other and handles messages between them. The network module should only contain very basic functionality, which means that the node behaviour and content distribution algorithm must be implemented in a separate CDA module. This makes it easier to re-use the code with different kinds of CDAs. Finally, every system needs a video decoder module that displays the received stream on screen. Optionally, the video en-/decoder can use an error correction module based on multiple description coding (MDC) [14]. The idea behind MDC is that the video encoder splits the original video stream in two or more substreams called descriptions. The video decoder can reproduce the original stream from any of the descriptions, but the playback quality depends on the number of received descriptions. All of these modules (video encoder/decoder, network, and MDC) have already been implemented within the PDS group in a similar research project on a tree-based CDA called Orchard [10], and are reused in the Lumberjack player.

## 1.4 Problem Statement

In this thesis we will focus on the following research question: How does a Chainsaw-based implementation perform in a real video player and network, and in particular does it perform as well in a real environment as it does in simulations? Since the authors of Chainsaw only simulated the distribution algorithm, there is no Chainsaw implementation freely available. Therefore, we have implemented our own version, which we call *Kettingzaag*, and embedded it in the Orchard video player by replacing the Orchard CDA module with *Kettingzaag*. Since this CDA replacement removes the relationship with trees, we renamed the player to *Lumberjack*. Additional questions to be answered in this thesis are:

- Do we need changes to the original Chainsaw algorithm to create a working implementation?
- Which parameters are most important in *Kettingzaag*, and what are good settings for various network sizes?

## 1.5 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 gives a more detailed overview of the categories of CDAs, MDC, and the technology behind *Kettingzaag*. In order to test our Lumberjack player, we need a controlled test environment and a test plan. Because our goal is to use Lumberjack in a real-life scenario, we have set up a number of virtual servers on rented hosts all over the globe. Since these hosts are controlled by Delft University of Technology, and the total number of hosts was planned to be 37, the network is called *Delft-37*. Unfortunately, the Delft-37 network turned out to be too slow to host multiple Lumberjack instances. The Delft-37 setup, tests and measurements are discussed in Chapter 3. Because

Delft-37 turns out to be unsuitable we will run Lumberjack on the DAS-3 [24] supercomputer in Delft. Since the DAS-3 nodes are extremely well connected, the Lumberjack software limits the upload rate at 1 Mbps and simulates round-trip times between nodes based on the Delft-37 test data. Chapter 4 is about the actual testing of Lumberjack. The chapter starts with an overview of the Kettingzaag parameters, and the measurements that we want to perform. This is followed by a discussion of the first test set that operates on small 10-node networks. These tests provide good parameter settings for the second test set, which operates on larger 50- and 100-node networks. The second test set shows that Kettingzaag successfully reduces stress at the content server, but that further work is needed before it can be used in a commercial environment. Our conclusions and future work are presented in Chapter 5.

## Chapter 2

# The Design of Kettingzaag

Live video streaming is becoming more and more popular, which means that costs are rising for content providers. This is inherent to the client/server distribution model that is used by most content providers. We will show that the peer-to-peer (P2P) model is the most cost effective on public networks, such as the Internet. Although many P2P algorithms exist for live video streaming, we have chosen for a fairly new swarm-based algorithm called Chainsaw. Chainsaw performs very well in simulations, but has not yet been implemented in a real video player so far. We have implemented our own version called Kettingzaag, and added some features that are not available in the original algorithm. In this chapter we will present the background of Chainsaw and the design of Kettingzaag.

In Section 2.1 we give an overview of the three approaches that are available for live video streaming. Section 2.2 explains the idea behind multiple description coding, and how this is implemented our Lumberjack video player. This is followed by Section 2.3 which describes the technology behind Chainsaw. We will finish this chapter with Section 2.4 which discusses the differences between Chainsaw and our own implementation called Kettingzaag.

### 2.1 Three Approaches to Live Video Streaming

In this section we will describe the three possible network models that can be used to stream video to multiple clients [1]. These models are the *client/server* model, the *network layer multicast model* (NLM) and the *application layer multicast* (ALM) model. Section 2.1.1 points out that the client/server model is undesirable due to high costs. Section 2.1.2 shows that NLM is only possible when all network devices support it, which is not the case on the Internet [1]. And finally, Section 2.1.3 explains why ALM is the best solution to distribute video to a large number of viewers.

### **2.1.1 Traditional Approach**

The traditional solution to video streaming is to let each user download from a server of the content provider. This solution has two major drawbacks [3]. First, this approach is not scaling well. Each content server can serve a limited number of users, depending on server and network loads. When the user limit is exceeded, the quality of the stream drops, or the stream cannot be viewed at all. The only solution to this problem is to add more content servers and share the user load. Furthermore, more users lead to higher bandwidth usage. Since bandwidth is charged per gigabyte, this solution gets more expensive as the service becomes more popular [2]. The second drawback is that this solution is vulnerable to an abnormal number of users connecting in a very short period of time. This phenomenon is called a flash crowd and can happen in case of a big news event. To support flash crowds, the content provider needs to setup a large number of extra servers, which are not used (fully) under normal operation, or deny some users service when the user limit is reached.

### **2.1.2 Network Level Multicast**

The best solution would be to have the server multicast the stream to all subscribed clients. This can be done in two different ways. Either network level multicast, or application level multicast can be used. Network level multicasting lets the network's routers handle packet duplication where necessary. Nodes subscribe to their local router for a multicast group, which in turn tries to subscribe to a router which is closer to the multicast source. Eventually the path to the source is complete and the node starts receiving the stream. This approach sounds ideal, but has the following four drawbacks, which are described in detail in [1]. First, the media source has no idea which nodes are subscribing to the multicast group, as subscriptions are handled by the network routers. This prevents an easy method of billing customers for content that is not free. Second, not all Internet Service Providers (ISPs) and Internet routers support network level multicasting, which prevents at least some users from viewing the stream. Third, the network level multicast protocol does not guarantee that packets arrive within a certain timespan, or arrive at all. Finally, when routers crash, they lose their routing tables, which results in breaking the network topology. These last two problems are very undesirable in real-time streaming applications, because time constraints are very important. Since these problems are very hard to tackle without access to the routers, network level multicast is not a realistic option for any multicast application that uses the Internet.

### **2.1.3 Application Level Multicast**

Application Level Multicast (ALM) means that the application sets up and controls its own network. The big advantage is that the application has full control over the network topology, routing decisions, and rules within the network. For

instance, one of the most important rules in the Kettingzaag network is that nodes should exchange data with each other. Since the application defines a network on top of another network, the application's network is also called an overlay (network). Many different overlay network solutions exist, but most of them can be categorized as either flooding-based (e.g., CAN [20]), or tree-based overlay types (e.g., Scribe [13], NICE [19]). Simplistic versions of the overlay categories are shown in Figure 2.1, where  $S$  represents the content server, and the other nodes the clients. A fairly new overlay solution, called Chainsaw [25], drops the strict node relationships that exist in the other categories. Instead, nodes will barter with each other, similar to the BitTorrent protocol.

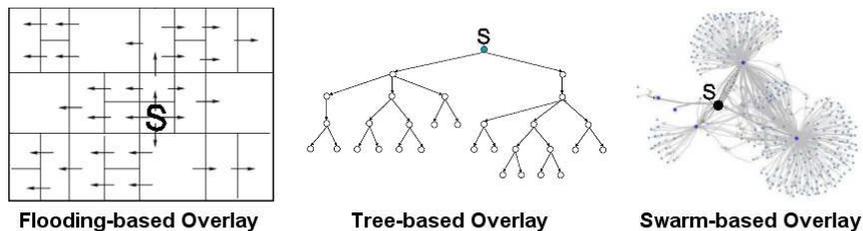


Figure 2.1: Flooding-, tree- and swarm-based overlay types.

Although overlay solutions can be very different, they must all meet the following three requirements to be successful:

1. the end-to-end delay between source and receiver must be reasonable.
2. joining and leaving of nodes must be handled quickly and locally.
3. the overlay network must be scalable.

First, keeping the end-to-end delay reasonable is just a matter of keeping the number of nodes between the content distributor and receiver small. Furthermore, nodes that are geographically close can be grouped together into so called Geo-Clusters [5, 15]. Second, the joining and leaving of nodes must be handled locally to prevent stress on the server. Furthermore, it must be handled quickly to ensure continuous playback of the media stream. This may lead to problems in flooding and tree-based approaches, as a leaving node disconnects a part of the network. This may lead to problems when many nodes fail within a short timespan. Finally, an overlay is scalable when the load on the content server does not increase with the number of clients in the overlay network. In theory, the server needs to insert its media stream only once when the total upload rate of the clients is higher than or equal to the download rate. This is easily observed in Figure 2.2, where each client passes the stream to the next client (upload rate equals download rate). Note that this figure also shows the worst end-to-end delay possible between the source and the last node. Although this requirement is met for all three overlay

categories, the flooding and tree-based algorithms have a drawback. In a tree, the leaf nodes are not uploading any data at all. However, they make up approximately 50% of the network under normal operation. This is compensated by the inner nodes, which upload to at least two children. This is no problem if the inner nodes have a fast enough upload. However, our solution aims at a bit rate that is close to the maximum upload capacity of the majority of home users in the Netherlands (1 Mbit/s). Because most of the nodes in our network will be home users, the tree-based solution will not work, unless the media stream's bit rate is decreased. The disadvantages of flooding overlays are comparable to those of the tree-based solutions. Although the number of strained inner nodes, and non-uploading outer nodes is smaller than in a tree-based solution, they still make up a considerable amount of the network. Furthermore, the end-to-end delay in flooding-based solutions can become a problem for a large number of nodes.



Figure 2.2: Simplistic P2P setup where every client uploads as much as it downloads.

Experimental results [25] show that Chainsaw does not suffer from packet loss under normal operation. When half of the nodes in a 10,000 node network fail simultaneously, less than 1% of the remaining nodes suffer from packet loss (packet loss ranging from 0.1% to 17.5%, with mean 3.74%). These numbers can be further improved, since the simulated nodes did not update their neighbor list upon failures. Chainsaw also provides for quick startup times. New nodes can start playback within a few seconds from joining, without suffering from packet loss. So, a Chainsaw overlay does not seem to suffer much from a high node failure rate. Furthermore, the drawback with leaf and outer nodes does not occur with Chainsaw, because all nodes are allowed to barter with each other.

## 2.2 Multiple Description Coding

Multiple description coding (MDC) is a technique that is used to make the multicasting of a video stream more robust to errors. The idea is that the original stream is split up in two or more sub-streams, called descriptions. The original stream can be reproduced at a node from any number of received sub-streams. The quality of the reproduced stream depends on the number of sub-streams that are received, where a higher number of sub-streams lead to a better quality. When all of the sub-streams are received, the reproduced stream is exactly the same as the original stream. Normally MDC is used as an error correction tool in the video en-/decoder only, meaning that the underlying network is not affected by it. However, we have chosen to allow clients to switch to half-quality mode by deliberately ignoring one of the two video descriptions. This means that bandwidth can be saved when the

network knows which stream to download and which to ignore. Chainsaw does not provide this functionality, but can be augmented without many problems.

Our Lumberjack player uses a simple form of MDC, which divides the original stream in three sub-streams. The first sub-stream is called the even stream, which contains all video packets with an even number. The second stream contains video packets with an odd number, and is called the odd sub-stream. The last stream contains all of the audio packets. Audio is not divided in two streams. Since audio bit rates are small compared to the video bit rate, and also can be given a higher download priority, they are less likely to be lost. A schematic overview of the Lumberjack MDC algorithm for video is shown in figure 2.3. Decoder 1 repairs the missing odd frames from the received even frames. The decoded stream is called a half quality stream. Decoder 3 is comparable to decoder 1, except that even and odd frames are switched. Decoder 2 is used when both the even and odd frames are received.

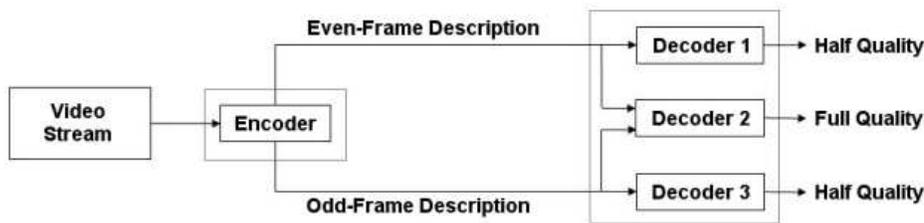


Figure 2.3: Lumberjack MDC Setup.

More complex MDC solutions exist, and are described in [5]. More sophisticated MDC schemes exist [5, 26], but they are more difficult to implement and have more overhead. Since our research aims at network performance, we have chosen to keep the simple MDC solution for now.

## 2.3 The Chainsaw Overlay Network

The Chainsaw overlay does not have a strict network topology as is found in tree- and flooding-based networks. Basically, every new node connects to a set of random neighbour nodes. The network does not maintain a global state with node information either. This makes the network very resilient to node failures and communication errors. However, since a node cannot easily predict which of its neighbours downloaded which packets, data pushing is practically impossible. Therefore, Chainsaw is based on data pulling in combination with *local* state gossiping. This means that every node maintains both its neighbours' states and its own state which is gossiped to its neighbours. The fact that nodes make their own download decisions raises the following three questions for each node:

1. Which packets are desired by the node?

2. Which packets are available for upload by the node?
3. Which neighbour(s) can upload the packets that the node desires?

In order to answer these questions, each node maintains three data structures. The first data structure is a list of desired packets, called the *window of interest*. Obviously, the contents of the window of interest depend on the playing position of the node. It seems reasonable to add all future packets that have not yet been downloaded. However, packets that are very close to the node's playing position will most likely not be downloaded in time. Therefore, the window of interest slides along somewhat ahead of the node's playing position. This means that packets that are too close to the playing position are removed and considered lost, while new packets are augmented at the stream rate. The second data structure is a list of downloaded packets, called the *window of availability*. The window of availability slides along with the playing position as well, providing a fixed length history of downloaded packets. Since every node's playing position is close to that of its neighbours, old packets can be safely dropped. The final data structure is an array of *status lists*, one list for each neighbour node. During media streaming, every node gossips the state of its window of availability to its neighbours. When a node receives this availability information, it is stored into the status list belonging to that particular neighbour. By searching the array of status lists, a node can determine which neighbour(s) can provide a certain packet. In order to get a good understanding of the Chainsaw principles, we will discuss the protocol in more detail below.

The most important job for a node is to acquire new packets. This can either be realized by the seed node(s) by encoding the media, or by the other nodes by downloading from neighbours. Both events are handled in the same way. So, when node  $A$  receives a new packet with number  $p$  and data  $D_p$ , it removes  $p$  from its window of interest, and adds  $D_p$  to its window of availability. Then node  $A$  gossips the availability of packet  $p$  to its neighbours, by sending a  $NOTIFY_p$  message. The neighbours that receive this notification message will add packet  $p$  to their status list for node  $A$  ( $statuslist_A$ ). Suppose node  $B$  is one of the neighbour nodes of node  $A$ , and  $B$  has selected packet  $p$  from its window of interest as the next packet to download. Node  $B$  will try to find a suitable neighbour by checking all of its status lists for packet  $p$ . If the packet is available at more than one neighbour, a random candidate is chosen. For now, suppose node  $B$  finds packet  $p$  only in  $statuslist_A$ . So, node  $B$  asks node  $A$  to upload the packet by sending a  $REQUEST_p$  message to  $A$ . When node  $A$  receives this request, it can grant it by sending packet data  $D_p$  back to node  $B$ . Although the algorithm is fairly simple, two issues arise when it is implemented exactly as described above.

First, if a node receives a notify message for some packet, it will most likely request the packet immediately. Since the round-trip time for each neighbour differs, the fastest neighbour will probably be the first node to request every time. Because we aim for a stream rate that is close to the upload rate of the nodes in the network,

such behaviour would saturate the upload link of the notifying node. The result is that the other neighbours do not get a chance to request packets from the notifying node. As a result, packets are not well distributed among the nodes, degrading network performance significantly. In order to prevent this from happening, each node has a maximum number of outstanding requests per neighbour. We call this the *maximum parallel request* threshold. The second issue arises at the seed node when it is uploading at its maximum capacity. Since nodes cannot always determine which packets have been uploaded by the seed before, some packets will be requested more than once. The downside is that requests for packets that have not yet been uploaded before may be choked due to the seed's saturated upload link. To prevent this from happening, the seed can override a packet request from a node. What happens is that the seed maintains a *packet overriding* list which contains the numbers of the packets that have never been uploaded before. If an incoming request contains a packet number that is not in this list and the list is not empty, the seed overrides the request. The oldest packet from the packet overriding list is sent back to the requesting node, and the packet number is removed from the list.

## 2.4 The Kettingzaag Overlay Network

Our Kettingzaag algorithm uses the same rules as the Chainsaw algorithm described above, with a few additions. First, we have added a *REJECT* message to the protocol, that notifies a requesting node that its request has not been granted. This speeds up decisions in the requesting node, because it does not have to wait for a request timeout. Second, we have added a ping-pong mechanism between nodes, which determines the round-trip time (RTT) between two nodes. This RTT value is used to prevent requests for future packets that are known to arrive too late. Third, a notion of MDC has been added to Kettingzaag, allowing us to switch a node to half-quality mode by deliberately not requesting one of the two video streams. This degrades video quality for the client, but also reduces the bandwidth usage significantly. Finally, we have added some debug messages to request clients to send their logfiles and to shutdown. Technically, these messages are not affecting network performance at all, but they do make testing on uncontrolled networks easier. We will describe each of these four additions in more detail below.

### Reject Message

In our Kettingzaag network, node  $A$  can deny a request from node  $B$  by sending a  $REJECT_p$  message to node  $B$ . This speeds up the network, because nodes do not have to wait for a timeout before re-requesting packet  $p$  from another node. Packet requests can be rejected for five reasons. First, the requested packet may have been removed from the window of availability of node  $A$ . Second, the incoming request at node  $A$  is a re-request from node  $B$ , but the original request has not yet been processed and is still in queue. Third, the incoming request at node  $A$  is a

re-request of a packet that has already been, or is being uploaded to node  $B$ . This may occur if the timeout at node  $B$  expires at the same time that node  $A$  starts uploading packet data  $D_p$ . Fourth, the request may be coming from a node that has notified that it is about to disconnect from the network. And finally, when node  $A$  has reached its upload limit, requests from  $B$  are rejected, and node  $B$  is choked. This means that node  $B$  will not request more packets from node  $A$  for a short period of time. Obviously, when the request is not rejected,  $D_p$  is sent from  $A$  to  $B$  and the whole sequence starts over again.

### Ping-Pong

Kettingzaag uses a *ping-pong* mechanism to determine the round-trip time between a pair of nodes. Each node periodically sends a *PING* message to all of its neighbours. When a neighbour receives a ping message, it immediately replies with a *PONG* message to get a good approximation of the round-trip time (RTT) between the pair of nodes. The RTT between a pair of nodes is used to determine if future packets should be downloaded or not.

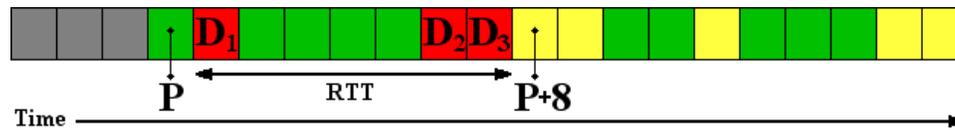


Figure 2.4: Packet download decision based on playback position and round-trip time.

Figure 2.4 shows the node state of node  $B$  at playback position  $P$ , and is used to explain how to determine which packets should be downloaded. The blocked bar represents the video stream, where each block represents a single frame/packet. The green packets have already been downloaded by node  $B$ , and are available for future playback. Both the red and yellow packets are unavailable, and need to be downloaded from neighbours. Suppose that node  $B$  has found neighbour  $A$  from which it can download all of the red and yellow packets, and that the latest ping-pong sequence resulted in a RTT as shown in Figure 2.4 (equal to 7 packets). This means that by the time node  $B$  receives a packet from  $A$ , the playback position will have shifted 7 packets into the future. Hence the red packets  $D_1$ - $D_3$  cannot be downloaded in time, and can be discarded for node  $A$ . The packets that can be downloaded from node  $A$  in time are shown in yellow from position  $P + 8$ . Naturally, packets  $D_1$ - $D_3$  may be downloaded from other neighbours with lower round-trip times.

### Multiple Description Coding

As explained in Section 2.2, MDC is normally used as an error correction tool in the video en-/decoder, which means that Kettingzaag should not be affected by

its presence. However, we have chosen to allow clients to switch to half-quality mode by deliberately ignoring one of the two video descriptions. Therefore, the Kettingzaag network has been augmented with the notion of MDC streams. Since the video stream is played at 30 frames per second (fps), both the even and odd streams play at 15 fps. For simplicity we built the audio stream at 15 fps as well, and add it in between the odd and even frames of the video stream. The total stream consists of three descriptions and its layout is shown in Figure 2.5.



Figure 2.5: Kettingzaag media stream layout.

This stream layout allows us to easily identify to which stream number packet  $P$  with packet number  $p$  belongs to by calculating  $p \bmod 3$ . The relationship between the stream number and the stream description is shown in Table 2.1.

Stream Number ( $p \bmod 3$ )	Description
0	Packet $P$ belongs to the <i>even video</i> description.
1	Packet $P$ belongs to the <i>odd video</i> description.
2	Packet $P$ belongs to the <i>audio</i> description.

Table 2.1: Relationship between the stream numbers and the stream descriptions.

### Debug Messages

We have implemented two additional debug messages. First, we added a *SHUTDOWN* debug message, which allows the seed to request nodes in the network to shutdown. And second, we added a *SEND – LOGS* debug message that allows the seed to order nodes in the network to send their logs. Although these debug messages do not influence Kettingzaag network performance, they may be useful in networks that are not under our full control.



## Chapter 3

# The Delft-37 Testbed

In order to test the Lumberjack player and the Kettingzaag algorithm in a real environment, we have rented some virtual servers on the Internet. All of these servers together form a network, which we call the Delft-37 network. In order to make Delft-37 easy to use and manage, we have implemented a program that allows us to control the network and its contents from a central control point. Since the Delft-37 nodes may be unreliable, we added a few reliable machines at the university network for controlling Delft-37, running the Lumberjack seed and storing test logs. In this chapter we will present the Delft-37 network, and its purpose as a real-life test environment for our Lumberjack player.

Section 3.1 gives a more thorough introduction of the Delft-37 network. Section 3.2 explains why a central control point is needed, and how we have implemented it. Section 3.3 describes the Delft-37 infrastructure at the university network. Section 3.4 discusses the Delft-37 network performance. Finally, Section 3.5 concludes why we have decided to use the DAS-3 supercomputer instead of Delft-37 for testing our Lumberjack player.

### 3.1 Introduction to Delft-37

The Delft-37 network consists of a number of virtual servers, which are running at hosting providers all over the world. The network consists of 6 nodes, which are listed in Table 3.1. The original goal of 37 nodes (hence Delft-37) has been abandoned for two reasons. First, a network of 37 virtual hosts is expensive. Second, most hosting providers which rent virtual servers are located in Europe and North America, and it is not useful to place a total of 37 nodes on these continents alone. Providers in Asia exist, but most of them have placed their servers in Europe or North America. Acquiring an account at providers that have not placed their servers abroad is problematic due to language barriers. South American and African providers do offer hosting, but almost all of them offer web-based accounts only. This type of account is useless for our purpose, since our overlay implementations are written in Python, and need to run on a virtual server. Luckily, peer-to-

peer technology is very popular in the United States and Europe at this moment. The distribution of nodes in the Delft-37 network reflects this situation.



Figure 3.1: Geographic locations of Delft-37 hosts.

Full Name	IP Address	Location
vds-355074.amen-pro.com	62.193.219.68	Paris, France
advantagecom.us.peer-2-peer.org	66.29.146.21	Walla Walla, Washington, USA
d80-237-144-205.dds.hosteurope.de	80.237.144.205	Köln, Germany
adiungo-phoenix.us.peer-2-peer.org	193.192.247.157	Phoenix, Arizona, USA
adiungo-london.uk.peer-2-peer.org	193.192.247.133	London, UK
usonyx.sp.peer-2-peer.org	202.172.255.90	Singapore

Table 3.1: Delft-37 nodes with IP address and location.

Delft-37 can be compared to the PlanetLab [17] network, although PlanetLab is much bigger in size with over 900 nodes. Also, PlanetLab consists mostly of university computers and networks which are far better connected than most of our virtual servers that form the Delft-37 network. Our virtual servers have different connection speeds to the Internet and are heavily used by other users as well. Furthermore, the underlying network is not always of the high quality found at universities. These factors introduce a level of uncertainty which is not found on the PlanetLab network. Since our goal is to test our software in a dynamic environment where problems can arise now and then, the good network properties of PlanetLab make it unsuitable for our research.

## 3.2 The Delft-37 Controller

Having a test environment such as Delft-37 is useless without some form of management. Many activities, such as distributing software and starting programs on the network, are not easy to achieve without a flexible control point. The ssh protocol accepts authentication by using keys, but this is not very flexible for multiple

control nodes. This key-based approach has two drawbacks. First, the keys have to be distributed to all Delft-37 nodes and controlling computers, which is not convenient when multiple control points are available. And second, there is no control over command execution when ssh is running as a background command execution process. The key-based approach allows a user to execute multiple commands on each host, but when a command is finished, the next command will be started immediately. Since not all hosts have the same processing power, some hosts may be executing their fourth command, while others are still executing their second. Furthermore, not all Internet links from the central controller to each of the hosts is equally fast. Hosts with faster links will start their command execution earlier than hosts with slower links. To tackle these problems, we wrote a control program in python. This Delft-37 controller makes the authentication transparent to the user, and allows parallel command execution. The controller monitors and synchronizes command execution for each host. It does so by checking if all hosts are ready. When all hosts have finished their last command, the central controller sends the next command to each of the hosts. The nodes that make up the Delft-37 network are specified in a file. Adding new nodes is merely a matter of downloading the latest host file. The Delft-37 controller can be run from any computer with an Internet connection. The computer that is running the controller program is from now on referred to as *the central controller*. The controller program is only distributed among the people that use the Delft-37 network. All options of the controller are explained in the next sections.

### 3.2.1 Executing Commands on Delft-37

The controller program allows the user to control all of the Delft-37 nodes from the central controller simultaneously. The user provides a list of commands to the controller program. The central controller tries to login using ssh on all of the Delft-37 nodes, using some timeout for failing nodes. When all nodes have either responded to be ready or timed out, the central controller sends the first command from its list to each node. This is done in parallel. Then the central controller waits for all hosts to reply that they have finished their command execution. When all hosts are ready again, the second command from the controller's command list will be sent. This process continues until all commands have been executed. An optional buffer-output option determines the way the output of each node is displayed on the central controller's screen. If the option is omitted, the output will be printed to screen as soon as it is received by the central controller. Since the commands on the Delft-37 nodes are executed in parallel, the output will probably be mixed up with output of other nodes. When the output is buffered, it is stored in a separate buffer for each host. These buffers are printed to the controller's screen after the last command finishes on all Delft-37 nodes. The buffers are printed one after the other, preventing output mixups.

### 3.2.2 Copying Files to the Delft-37 Network

The controller program allows the user to upload files from the central controller to all of the nodes in the Delft-37 network. The central controller uploads the requested files to multiple Delft-37 nodes simultaneously. The reason for multiple connections is simple. Some Delft-37 nodes may have a lower download bandwidth than the central controller's upload bandwidth. In this case the upload capacity of the central controller is not fully utilized. To utilize the remaining bandwidth, another connection will be opened to an available node. The total number of simultaneous connections can be specified by altering a parameter inside the controller program. When some nodes are not available, an error message is shown on the central controller's screen when the controller program terminates.

### 3.2.3 Copying Files from the Delft-37 Network

The controller program allows the user to download files from the Delft-37 nodes to the central controller's hard-drive. This is done using the ssh secure copy program. The files that are requested are searched for on each Delft-37 node. This means that the central controller will receive file  $F_a$  from node  $H_a$ , but also from node  $H_b$ , node  $H_c$  and so on. It is clear that file  $F_a$  will be overwritten if no measures are taken. This problem is solved by having the controller create a new directory for each Delft-37 node. Because host names can be the same on different networks, and URLs can be very long, the directory names chosen are the IP addresses of the nodes. For easy maintenance, these IP-named directories are stored in a special directory. This directory is used for each download action, so multiple downloads might overwrite files as well. It is up to the user to rename the special directory before a new download is started.

### 3.2.4 Delft-37 Network Measurements

The controller program allows the user to run four tests on the Delft-37 network, which are *ping*, *traceroute*, *bandwidthTCP* and *bandwidthUDP*. A major problem of network performance tests is that they can affect each other. Although ping and traceroute tests are affected minimally, bandwidth tests can be influenced to a great extent. When a node is measuring bandwidth speeds with more than one node, it will have to divide its available bandwidth, resulting in erroneous results. To avoid this situation, a scheduling algorithm has been added to the network performance test routine. The next section elaborates on how these measurements are done, and presents the test results that were obtained.

## 3.3 Delft-37 Infrastructure

Although any Delft-37 node can act as the central controller, we would like to assign this job to a few machines that are not only more reliable than the Delft-

37 nodes, but also have more computing power and storage space. Therefore we have setup three superstorage machines in Delft with a total disk capacity of 10 terabytes. These superstorage machines have four roles. First, they act as the Delft-37 controllers for uploading new Lumberjack versions and starting tests. Second, they are used as a central gathering point for downloading test logs from each Delft-37 node. Third, they will encode video streams which can be either distributed in real-time, or saved to disk for later distribution. And finally, they inject a lot of legal torrents which can be downloaded with the Tribler BitTorrent-based client. Each superstorage machine contains 14 hard drives of 320 GB in a RAID-5 array (3,8 TB storage for each machine). Since the motherboard can only store two serial ata (SATA) drives, we added 3 SATA controllers to the PCI bus, which connect the other 12 drives. Although we are aware of commercial storage solutions, we choose to implement our own solution for two reasons. First, the above setup will most likely lead to interesting problems, because it runs on the edge of the hardware's capabilities. And second, setting up a system with hardware from a local computer shop is much cheaper than a commercial solution. The super storage machines run the latest 64-bit version of Debian Linux.

Partition	Disk	sda	sdb	sdc	sdd	sde	sdf	sdg	sdh	sdi	sdj	sdk	sdl	sdm	sdn
1		boot	boot	swap	swap	root									
2		data													

Figure 3.2: Partitioning scheme for each super storage machine.

Each machine uses the partitioning scheme that is displayed in Figure 3.2. The boot and swap partitions are setup as a RAID-1 array (mirrored). The root and data partitions are setup as a RAID-5 array. This setup ensures that the machine can keep on running when at most one drive fails. In case of a failure, the machine needs to be shut down to replace the failed disk, after which the broken arrays can be repaired.

### 3.3.1 Problems

The first problems were discovered soon after installation and setup of the Linux operating system. Some of the drives were (partially) broken on arrival and failed during RAID-5 initialization of the data partition. After replacing the bad disks and reinstalling the system everything seemed to work fine. However, when the disks were put under heavy load, the RAID-5 array of the data partition would break down within an hour (and sometimes even during RAID-5 initialization). Most of the time one or two disks would fail with a 'DriveReady Seek Complete Error'. Testing the failed drive(s) one by one in another system would not lead to errors, and also the hard drive's built-in monitoring system (SMART) did not discover any errors. So with all of the drives being healthy, we needed to solve a different problem. A quick search on the Internet showed some ideas that might lead to a

solution for this problem.

### **Cable Interference**

Although cable interference is no issue under normal circumstances, it can be a problem in our machine. Since there are 14 disks, there are also 14 cables, which makes it almost impossible to prevent them from crossing each other. This could possibly lead to some interference on the data lines, causing the DriveReady error. To make sure that interference is reduced to a minimum level, the drives were removed from their casing and laid out in a star shaped setup. Although this did not solve the problem, it increased overall stability. The DriveReady error still occurred under heavy load, but at a later point in time (mostly after a couple of hours). Since the star shaped layout does make a difference, this setup is used in the rest of our tests.

### **Power Supply Lines**

The Linux kernel mailing list suggested that the power supply lines should be checked, because the DriveSeek error is sometimes caused by bad power supplies. Although the power supply is a high quality 680W unit, we did check the power lines. Although the power supply manual did mention that the unit divided its power to two power lines, it did not mention that the lines were not equally strong. So, some of the disks were transferred from the weaker line to the stronger line. Unfortunately, this did not improve the stability of the machine. Still, the new setup is maintained to be on the safe side.

### **Replacing SATA Cables**

The Fedora Linux forums showed an issue where the SeekComplete error was caused by bad cabling between the SATA drives and its controller. We started replacing cables for every drive that raised the error, but this did not lead to better stability. In fact, testing the replaced cables in another system would not lead to any errors, as long as the number of disks was small. This lead us to the idea that the PCI bus might be saturated during extensive usage of the 14-disk RAID array.

### **Saturated PCI Bus**

So, with the cables seemingly correct in small setups, we tried setups with 8, 9 and 10 drives, where the drives were connected to only 2, 3 or all controllers (including motherboard). The controller setup did not seem to have any influence on the problem, but the number of disks did. All setups with more than 8 disks would fail within a day. The motherboard's PCI bus (32 bits) has a bandwidth of 127.2 MB/s. The sequential write speed of our drives is measured somewhere between 16 and 17 MB/s, which comes close to the PCI bus speed in case of 8 drives. Although an 8 drive setup should be able to saturate the PCI bus, this will only happen for a

short period of time. Fortunately, the kernel driver for our SATA controller had an option to slow it down. After slowing down the kernel driver, a 9 drive setup could not saturate the PCI bus anymore. However, such a setup would still raise errors.

### **Changing Kernel**

Initially the machine was running an unpatched 2.6.15.3 kernel from kernel.org. We decided to try out the newest 2.4 kernel to see what would happen. So, the entire machine was reinstalled with a Debian Linux 2.4 installation disk, and the kernel was patched to 2.4.32. The raid setup was running smoothly, but the same errors occurred during long time write tests. The kernel change had no influence on the stability of our setup.

### **Replacing the Hard Drives**

As a last resort we replaced the Western Digital hard disks with Maxtor drives. The RAID array was built up successfully, and surprisingly the write tests executed for more than a day. We kept stressing the RAID array with extensive write tests for over a week without errors. With one of the super storage systems finally running stable, it is time to investigate the capabilities of the Delft-37 network. Meanwhile the hard drives of the other two machines are interchanged at our suppliers.

## **3.4 Delft-37 Network Performance**

We have executed some network tests to determine the Delft-37 network performance. The I-Share project aims at real-time video streaming, which can be done in different ways. Two popular solutions are multicast trees and the Chainsaw method. Multicast trees are used when a small number of senders must serve a large number of clients. The idea is that the sender(s) serve a small number of clients. Each served client serves a small number of yet unserved clients, and so on. This method allows a server to distribute its data to a much larger audience than would be possible by directly uploading to each client. The resulting tree is called a multicast tree, which becomes deeper as the number of clients increases. Our performance tests are focused on two important network link properties. The first is the delay between sending a packet from a source node and the receiving of that packet at the destination node. This delay is called the ping time. The second important link property is the bandwidth. The link bandwidth is defined as the total number of bits that can be transferred per second between the source node and the destination node. The ping time is important, as it determines the response time of nodes and the total delay between the source and leaf nodes in multicast trees. When the multicast tree gets deeper, the media stream must traverse more network links. When these links have a high delay, the leaf nodes of the tree will receive the stream at a later point in time. Bandwidth is important because video streams are distributed at a certain speed called the stream rate. Nodes that cannot

keep up with the stream rate will not be able to download some packets, which results in a degraded playback quality of the stream. The available bandwidth for a certain node also determines how many child nodes can be served by that node. Ping times and bandwidth measurements between nodes can vary in time. This is mostly due to the fact that the Delft-37 nodes are used by other users who take away bandwidth. However, it is also possible that the network near the source or destination is suffering from congestion.

### 3.4.1 Ping Tests on Delft-37

The ping tests show that the link quality between two nodes is almost constant. The time of day does not seem to affect the time that is needed to send and receive a ping request. This is probably because of the small size of a ping requests (compared to other packets) and the rate at which they are sent (once per second). The slowest node by far is Singapore, with a ping time of around 400 ms from (and to) any of the other nodes. The second slowest node is Washington, which has a ping time of around 190 ms. All of the other nodes have ping times less than 100 ms (mostly around 50 ms), which is quite fast. All results are displayed in Figure 3.3, which shows the minimum, mean and maximum times that were obtained by executing a ping from the host in the left column to the host in the top row. All ping times are in milliseconds.

From	To	Paris	Walla Walla	Köln	Phoenix	London	Singapore
Paris		---	168 168 170	24 26 30	35 38 45	35 42 55	384 391 398
Walla Walla		168 171 175	---	184 187 209	173 181 192	177 188 197	228 231 233
Köln		25 34 46	183 185 189	---	35 39 49	35 38 46	366 369 378
Phoenix		35 36 39	174 174 179	36 37 38	---	1 3 36	375 388 407
London		35 39 54	177 180 186	36 42 54	1 2 24	---	375 401 419
Singapore		384 387 391	227 236 259	366 369 380	374 399 430	397 410 436	---

Figure 3.3: Delft-37 nodes with their minimum, mean and maximum ping times in ms.

### 3.4.2 Bandwidth Tests on Delft-37

The bandwidth tests give some insight in the link capacity that is available between the Delft-37 nodes. Two bandwidth tests were run for each node. First a TCP bandwidth test is executed, directly followed by an UDP bandwidth test. This is done to make the comparison between both tests as fair as possible. Bandwidth tests are executed by a program called *IPerf* [8]. *IPerf* tests the TCP bandwidth by setting up a connection between two nodes. One of the nodes sends as much data as it can for a specified period of time. When the test finishes, the total number of bits sent is divided by the length of the period to obtain the bandwidth speed. In our tests the TCP time period is set to 3 seconds. The UDP bandwidth test is treated differently by *IPerf*. The program generates a stream at the specified rate for a

specified time period. When the test finishes, the receiver of the stream reports to the sender how many packets it received. From this information IPerf calculates the packet loss. Our tests use a time period of 3 seconds, with a stream rate of 1 Mbit per second. In a normal situation, where enough bandwidth is available and no network congestion occurs, the packet loss should be (close to) zero. In case of insufficient bandwidth, the number of datagrams lost per second should be close to the difference between the UDP stream rate and the previously measured TCP bandwidth. If the values differ too much, either other users are using up more (or less) bandwidth, or the network between the two nodes is congested. Network congestion can be investigated by comparing TCP bandwidth with UDP bandwidth (minus datagram loss) speeds.

The test results are shown in Figures A.1-A.6 for all nodes in Delft-37. All test runs started at 12:00 on Friday the 21<sup>st</sup> of April 2006, which is represented by the first test number, and ended at 12:00 on Wednesday the 26<sup>th</sup> of April 2006, which is represented by the last test number. Successive tests are 6 hours apart, meaning that the second test is at 18:00 on Friday the 21<sup>st</sup> of April, and so on. The same timestamp on the next day is found by adding 4 to the selected test number. In our tests, UDP datagram loss is normally well within a 10% range, with most tests not exceeding a 5% datagram loss. Bandwidth tests from other nodes to node London failed from test number 1 to 13 due to a crash in the IPerf server on this node. Bandwidth tests from any node to node Paris always fail, but tests done from this node to other nodes are successful. The test results lead to the following three conclusions.

First, the UDP datagram loss in test five (Figure A.5) for host London is extremely high, with numbers between 14% and 22%. The TCP bandwidth in this interval is also low. At this point in time, the network was probably suffering from congestion. It is a pity that this assumption cannot be proven with the results from the other host on the same subnet (due to its IPerf crash). Even so, the time interval for a certain test number to all other hosts is about 5 to 7 minutes, and it seems unlikely that other users take up that much bandwidth for such a long time; especially when the tests at other points in time do not show the same behaviour.

Second, some UDP bandwidth tests show strange results, like test 2 for host Paris (Figure A.1). The UDP bandwidth is 404 Kbit/s, with a packet loss of only 1.9%! Further investigation shows that the bandwidth tester, IPerf, waits for a last ACK packet to finish the test. If this packet arrives very late, for example three seconds after the test, IPerf takes a test time interval of 6 seconds, instead of 3. The result is that the measured bandwidth is divided by 6, instead of 3, which means that the actual bandwidth is halved. All UDP bandwidth dips below 700 Kbit/s are due to this phenomenon. Actual packet loss in these cases is always below 10%.

Third and last, in about 20% of the tests, the UDP bandwidth exceeds the TCP bandwidth. The fact that UDP bandwidth measurements are exceeding TCP bandwidth measurements is unexpected and may be explained by fluctuations in the bandwidth usage of other users. Another explanation can be that TCP congestion prevention algorithms might have a significant impact on these measurements for

short time intervals, like our 3 second tests. It may be a good idea to increase the TCP bandwidth test interval to 10 seconds, and see if this percentage of 20% drops significantly.

### **3.5 Conclusion**

The test results in this chapter show that most nodes in the Delft-37 network have a bandwidth capacity that is below 2 Mbit/s. Since the bit rate of the video stream will be close to 1 Mbit/s, most of the Delft-37 nodes cannot support more than one Lumberjack instance without saturating their Internet link. Taking the difficulties into account of acquiring nodes on different continents, we have decided to abandon the Delft-37 network. Instead, we will run tests on the DAS supercomputer in Delft.

## Chapter 4

# Testing Lumberjack

The Delft-37 network turned out to be unsuitable as a testbed for our Lumberjack player, so we will use the DAS-3 supercomputer and implement delays based on the Delft-37 test data. In order to get meaningful test results, we have identified the most important parameters in the Kettingzaag algorithm and defined the measurements that we want to perform during live streaming. Since the Chainsaw authors do not elaborate on many of the system parameters that we have defined, we start our tests with small networks and a variety of parameter settings. This will give us a good insight in network behaviour, which is needed for selecting parameters for larger networks. In this chapter we present the Lumberjack tests and the test results obtained on the DAS-3 supercomputer.

Section 4.1 describes the system parameters in the Kettingzaag algorithm. Section 4.2 gives an overview of the measurements that we perform during live streaming with Lumberjack. This is followed by Section 4.3 which explains which measurements we will discuss in our tests, and why we will omit some others. Section 4.4 discusses our first test set, which is used to get a good idea of the Kettingzaag network behaviour. Section 4.5 discusses our second test set, which deals with larger networks. We finish this chapter with conclusions about the two test sets in Section 4.6.

### 4.1 System Parameters

This section gives an overview of the parameters that play an important role in the Kettingzaag network. We can distinguish five parameters, which are the server seeding ratio, the stream's bit rate, the total number of outgoing connections for a node, the maximum number of parallel requests a node can make, and finally the size of the play buffer. Each of these parameters is discussed in more detail below.

## Server Seeding Ratio

The Server Seeding Ratio (SSR) is defined as the total number of times the content server will upload the media stream. The ideal SSR would be 1.0, since that would mean that the server only inserts the media stream into the network once, after which it is completely distributed by the clients. However, people want to view media at the best possible quality. As a result, the media stream rate will be close to the upload capacity of the individual nodes in the network. This makes such a low SSR very challenging to meet. A higher SSR is not necessarily bad, as long as the SSR does not grow (much) as the number of nodes in the network increases.

## Video and Audio Bit Rate

One of the most obvious settings is the total bit rate of the media stream, which is the sum of the video and audio bit rates. As long as the total bit rate remains below the maximum upload capacity of the majority of the nodes, the nodes should be able to distribute the stream amongst themselves. When the stream rate exceeds this threshold, the server needs to augment the missing capacity. In this case, the SSR will be growing linearly with the number of nodes in the network.

## Number of Neighbours

We define  $N$  as the total number of initiated connections to other nodes. When a node receives a connect request, it adds the requesting node to its own neighbour list as well, making the connection a two way communication line. So, each node connects to  $N$  neighbours and on average receives  $N$  incoming connections from other neighbours. This means that on average each node will be connected to  $2N$  neighbour nodes.

## Maximum Parallel Requests

In order to enforce a good distribution of packets in the network, nodes are not allowed to request all of their packets at a single neighbour. To prevent this, each node can only request a certain amount of packets from each of its neighbours. We call this threshold the maximum parallel requests (MPR) threshold. Although it is important to set the MPR threshold to a low value, a too low value will also prevent a good network operation. The problem is that when the threshold is set too strict, nodes with little neighbours cannot request their packets fast enough. Basically, the MPR threshold can be decreased when  $N$  is increased.

## Play Buffer Size

The play buffer is a block of memory that acts as a buffer between the network module and the video playing module. Its purpose is to prevent disturbances in

video playback due to small dips in network speed. When the network is somewhat slower for a short period of time, the buffer will slowly empty. So, it seems reasonable to choose a high play buffer size. However, the play buffer must be filled before playback can start at the startup of a node. So, in order to guarantee a good response time on startup, we would like a small buffer size.

## **4.2 Measurements**

The Lumberjack software will do measurements in four categories. Each of these four categories deals with some measurements which are discussed in detail below. First, the basic functionality category will be discussed in Section 4.2.1. Second, the user behaviour category is outlined in Section 4.2.2. This is followed by the basic P2P behaviour category in Section 4.2.3. Finally, the video behaviour category is discussed in Section 4.2.4.

### **4.2.1 Basic Functionality**

The basic functionality holds two measurements, which are the frame rate and the number of program crashes. These are discussed in the following two subsections.

#### **Frame Rate**

The frame rate specifies the number of frames that the video playback module plays per second. This number depends on the CPU speed of the machine and its video graphics adapter. Slower machines might lose frames which have been received by the network, just because they are lagging behind during playback. Although the DAS-3 supercomputer consists of multiple nodes, we still run more than one Lumberjack instance on a single computing node. This allows us to simulate more nodes, but at the same time prevents the software from actually displaying the video feed. Therefore we have implemented a fake video decoder which just decodes frames at the exact rate of the stream. So, the frame rate measurement is not interesting for our DAS-3 tests.

#### **Number of Program Crashes**

The total number of program crashes are recorded and sent back to our data collection machine. It gives us insight in the stability of the Lumberjack player.

### **4.2.2 User Behaviour**

The user behaviour can be divided in two subcategories, which are the time of connect/disconnect, and the video quality of the stream.

### **Time of Connect/Disconnect**

The moment at which a client connects is recorded, as well as the moment it disconnects. This measurement gives us insight in how many users try the software once, how many users try it more than once, and how long a user stays connected. Obviously, this measurement is not very useful for our tests on the DAS-3 super-computer.

### **Video Quality**

The video quality depends on the download speed of the client. The content distributor splits up the video in two separate streams. One stream consists of the video's even frames and the other of the odd frames. When a client is suffering from bandwidth problems for a longer period of time, the software switches to half-quality mode by dropping one of the streams. The user can also manually switch between half-quality and full-quality mode. During our DAS-3 simulations, all clients will run at full quality.

#### **4.2.3 Basic P2P Behaviour**

The basic P2P behaviour category holds three measurements. First the neighbour failure rate will be discussed. Then the latency to neighbours is explained, and finally the up/down bandwidth usage is outlined.

#### **Neighbour Failure Rate**

The neighbour failure rate (NFR) is defined as the number of neighbour nodes that fail within a certain test time interval. For example, when a node is connected with 23 other nodes (neighbours), of which 3 fail in a test time interval of 2 seconds, then the NFR for that two second time interval is 1.5 per second. The NFR gives insight in how quickly nodes are losing neighbours. Our first DAS-3 simulations will not include failing nodes.

#### **Latency to Neighbours**

The latency to a neighbour is defined as the time that is needed to get a TCP packet from one node to the other. Lower latencies are better, because video frames that are about to be played are more likely to arrive in time. Furthermore, latency to neighbours gives an indication of the geographical distance between two nodes. When the Chainsaw overlay network is full of high latency links, it is probably badly mapped on the Internet. Since the DAS-3 nodes are extremely well connected, we simulate latency between neighbours by delaying incoming packets at the nodes.

## **Upload Rate**

The upload rate is defined as the number of bits that a node uploads per second to all of its neighbours. The upload rate is specified as either Kbits/s (Kbps), or Mbit/s (Mbps).

## **Download Rate**

The download rate is defined as the number of bits that a node downloads per second to all of its neighbours. The download rate is specified as either Kbits/s (Kbps), or Mbit/s (Mbps).

### **4.2.4 Video Behaviour**

The video behaviour category can be divided in 5 subcategories, which are the actual video bit rate, frame loss, number of frame re-requests, timing of frames and the number of frame repairs.

#### **Actual Video Bit Rate**

The actual bit rate at a client is recorded, because a truly fixed bit rate is an illusion. The client's bit rate will fluctuate around the bit rate on which the stream is multicast. These fluctuations are important to monitor, because the mean bit rate should be close to the actual bit rate. Furthermore, a very good network performance can be caused by a lower bit rate at a certain point in time (e.g. during black screens).

#### **Frame Loss**

The number of frames that are lost because a frame has not been downloaded in time. Nodes with a slow connection are more likely to suffer from frame loss than faster clients. This measurement also gives insight in network problems when fast connected clients are suffering high frame loss.

#### **Number of Frame Re-requests**

Whenever a frame is requested from a neighbour, but is not received within a certain time interval, it is requested from another neighbour. The number of re-requests can give an insight into problems or high stress at some neighbour node.

#### **Timing of Frames**

We store the following frame times for a low level insight of the network:

- Frame Availability Time
- Frame Request Time

- Frame Receipt Time
- Frame Playback Time

The frame availability time specifies the moment at which a node is notified by a neighbour that the neighbour has a new frame available for download. From that time a node may decide to download that particular frame from the neighbour. When it decides to do so, it requests the frame at the neighbour and stores this moment as the frame request time. As the neighbour receives the request, it decides if it will upload the frame or not. When it does, the requesting node stores the time at which the frame is received as the receipt time. Every time a frame is played on the client, the node stores this time as the playback time. Getting to know these times gives insight in the delay between availability notifications, requests, receipts and finally playback.

### **Number of Frame Repairs**

Every time a node fails to download a frame from the network in time, it has to repair it during playback. Repaired frames are generated using the previous and the next frame. This results in a frame with degraded quality, but at least the video will not show a black frame (or stop) for a short period of time. Frame repairs are done constantly when the client is set to half-quality mode (see above).

## **4.3 Performance Metrics**

From the measurements that we described in the last paragraph, we define the following four as performance metrics for our tests:

- Frame rate in frames per second
- Frame loss in frames per second
- Upload rate in Kbits per second
- Download rate in Kbits per second

The measurements that will not be discussed are the Number of program crashes, the connect/disconnect time of nodes, the quality mode, the neighbour failure rate, the round-trip time to neighbours, the number of packet re-requests, the timing of frames and the number of frame repairs. The reasons are that some are low level debug information (number of packet re-requests, and timing of frames), others are non-existent (program crashes, and neighbour failure rate), and the rest is set to a fixed value. We will discuss these fixed values in the next paragraph.

We run all our tests at full quality with a round-trip time of 150 ms between any pair of nodes. In each test, all clients start roughly at the same time and stay connected during the entire test run. This makes the (dis)connect time and neighbour

failure rate measurements obsolete for our tests. Since the DAS-3 nodes can only be accessed by a command-line interface, we are unable to show the video feed. Therefore, the feed is handled by a fake decoder which just pops 30 frames per second from the frame stack. This means that the number of frame repairs cannot be measured, but this should not be a problem as long as the frame rate does not drop too much.

## 4.4 Test Set 1

With our first test set we will try to get an understanding of the effects that the server seeding ratio (SSR) and the number of neighbours have on the frame loss/frame rate of the nodes. Since the SSR becomes more important at the mean upload rate of the clients, we start our first test with 10 nodes and a total bit rate of 800 Kbit/s. This is close to the 1 Mbit/s upload limit that most home users have in the Netherlands. So, under perfect operation all nodes should be able to upload enough to support each other. Even though all nodes will upload, we do not expect perfect network behaviour. This will probably result in frame loss for lower SSR values. In Chainsaw, a SSR of 2.0 works well, so we have chosen to keep our SSR values close to 2. We will gradually increase the SSR values in our tests from 1.1 to 1.5, 2.0, 2.5 and finally 3.0. Hopefully, frame loss will decrease rapidly for SSR values of 2 and higher. We do not assume a fully connected network in real-life streaming situations, so we keep the number of initiated neighbour connections ( $N$ ) low. Since we expect that lower settings for  $N$  have a negative impact on network performance, we vary the values for  $N$  from 2 to 4. This leads us to *test set 1*, which is summarized in Table 4.1.

### 4.4.1 Test Setup

Although Table 4.1 shows 15 different test types, we only perform three test runs in which we run five test types each. This means that we dynamically change the SSR value (from 1.1, 1.5, 2.0, 2.5 to 3.0) in each test run to accommodate each test type. So, test run 1 contains test types 1-5 ( $N=2$ ), test run 2 contains test types 6-10 ( $N=3$ ), and test run 3 contains test types 11-15 ( $N=4$ ). The seed will start with an SSR of 1.1, and change it to the next higher setting every 60 seconds. So, each of these three test runs lasts 300 seconds and contains 60 timestamps per SSR setting. Since we have 10 nodes in each test run, there are 10 data points per timestamp. In order to keep our figures organized, we only keep track of the minimum (red data points), mean (green data points) and maximum (blue data points) values for each of these timestamps. We execute test run 1 (test types 1-5), 2 (test types 6-10) and 3 (test types 11-15) ten times to filter out anomalies in the results, so we can get a good insight of the tendencies in the network. We start our tests with ten executions of each of the three test runs without limiting the upload rate for client nodes. This allows us to determine if Table 4.1 provides realistic values for more

Test Run	Test Type	Video Rate (Kbps)	Audio Rate (Kbps)	# of Peers	SSR	$N$
1	1	704	96	10	1.1	2
	2	704	96	10	1.5	2
	3	704	96	10	2.0	2
	4	704	96	10	2.5	2
	5	704	96	10	3.0	2
2	6	704	96	10	1.1	3
	7	704	96	10	1.5	3
	8	704	96	10	2.0	3
	9	704	96	10	2.5	3
	10	704	96	10	3.0	3
3	11	704	96	10	1.1	4
	12	704	96	10	1.5	4
	13	704	96	10	2.0	4
	14	704	96	10	2.5	4
	15	704	96	10	3.0	4

Table 4.1: Tests of test set 1.

common upload boundaries.

#### 4.4.2 Unlimited Upload Results

The unlimited client upload results for each of the four performance metrics (frame rate, frame loss, upload rate, download rate) are shown per test run in Section B.1. The four performance metrics of test run 1 are shown in Figures B.1-B.4, the metrics of test run 2 are shown in Figures B.5-B.8, and the metrics of test run 3 are shown in Figures B.9-B.12. Since every test run is executed ten times, each of the mentioned figures contains 10 graphs. Considering that the results for all performance metrics are recorded simultaneously, graphs that are in the same position correspond to the same execution of a test run.

Remarkably, the test results for the frame rate with 2, 3 and 4 outgoing neighbour connections are almost the same (Figures B.1, B.5, B.9). We show the results for  $N=3$  in Figure 4.1 (a), because these summarize the other  $N$  settings well. The frame rate at an SSR of 1.1 is unacceptable for any value of  $N$ , but stabilizes at SSR values of 1.5 and up. Almost all test runs have a dip in the frame rate in SSR intervals 1.5 and 2.0. It is remarkable that the frame rate drop is worse in the higher SSR interval for all values of  $N$ . These drops are somewhat less deep for higher values of  $N$ , which is normal because nodes have more neighbours to download missing packets from. This also holds for higher SSR values where we notice small dips in the minimum frame rate for  $N$  values 2 and 3, while the frame rate is stable for 4 outgoing connections.

When we compare the node upload speed for different values of  $N$  (Figures B.3,

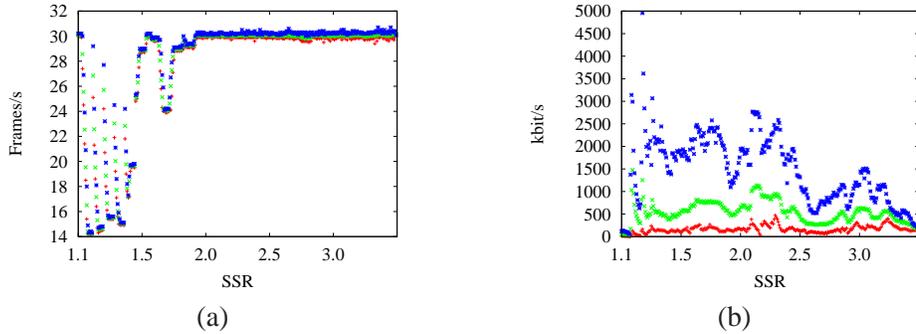


Figure 4.1: Minimum, mean and maximum frame rate and upload rate for test runs with 3 neighbours, an no upload limit. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=unlimited$ )

B.7, B.11), we notice that the mean upload is about the same in all figures and decreases for higher SSR values. This behaviour is shown in Figure 4.1 (b) and is expected, as the nodes can obtain more packets from the server at higher SSR values, and thus exchange less packets amongst themselves. The biggest difference in upload speeds is noticeable in the maximum upload which peaks higher for smaller values of  $N$ . Apparently, when the network is less connected, packets are only available at a select number of nodes, whereas packet availability is more balanced for higher values of  $N$ . With an unlimited upload speed, this is not much of a problem since the nodes just upload a bit more. This is probably the reason why the test results for the three test runs look so similar. Therefore, we will run the same test runs again, but with a more realistic upload limit of 1 Mbit/s for each of the nodes.

#### 4.4.3 Limited Upload Results

The results with a client upload limit of 1 Mbit/s are shown in Section B.2 in Figures B.13-B.24. Naturally, the frame rates at SSR value 1.1 are even worse than in the test runs with an unlimited upload. This time there is a huge difference between test run 1 with  $N=2$  and test runs 2 and 3 (with  $N=3$  and 4). This is shown in Figure 4.2 which contains two representative graphs from the appendix. As expected, the upload limit causes the frame rate to drop significantly for  $N=2$ . This is proven by the results in Figure B.15, which show that the maximum upload is at a steady value of 1 Mbit/s for SSR values of 2.0 and below. From SSR 2.5 and up, the maximum upload rate starts to fluctuate below its maximum value, indicating that the upload limit is not the bottleneck anymore. As a result, the mean frame rate value quickly stabilizes close to the desired value of 30 frames per second (fps). However, small frame rate dips are still common at higher SSR rates, especially for the worst performing node. The situation becomes better as the value of  $N$  increases, although the frame rate dips at SSR values of 2.0 and below are

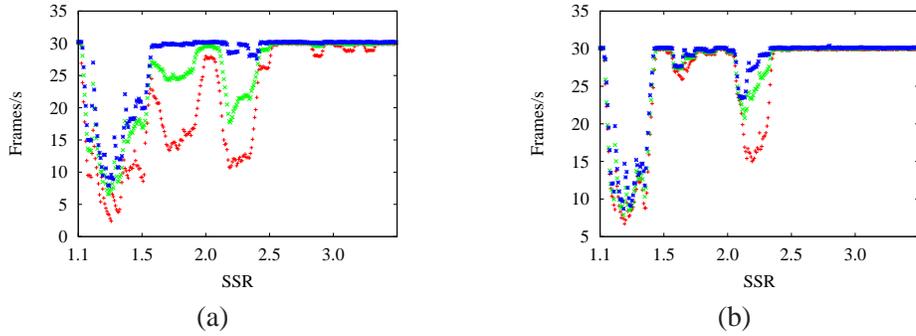


Figure 4.2: Minimum, mean and maximum frame rate for test runs with 2 (a) and 3 (b) neighbours, with a 1 Mbit/s upload limit. (Test Run: 1 (a) and 2 (b),  $SSR=1.1-3.0$ ,  $N=2$  (a) and  $N=3$  (b),  $UploadLimit=1$  Mbit/s)

still deep enough to be noticed by viewers. The frame rate does seem to stabilize in the second half of the  $SSR=2.0$  interval, so we run some extended tests for this interval. For each value of  $N$  we will run three extended tests which each lasts for 10 minutes at an  $SSR$  of 2.0. This will give some insight in the behaviour of the network after the frame rate has stabilized. The extended test results for  $N=3$  are shown in Figure 4.3. The results of the tests with 4 neighbours are practically identical to the  $N=3$  values. For  $N=2$  one of the tests failed with a mean frame rate below 10 fps. In order to check if this is coincidental, we ran another three test runs of which two failed.

## 4.5 Test Set 2

The goal in our second test is to find out if the  $SSR$  and number of neighbours depend on the total number of nodes in the network. The original authors of Chainsaw performed successful simulations with very large networks at low  $SSR$  settings, but they do not elaborate on the number of neighbour connections.

### 4.5.1 Test Setup

In test set 1 we have seen that an  $SSR$  value of 3.0 is working well, and therefore this  $SSR$  setting will be maintained in test set 2. In order to get an idea of the influence of the number of neighbours in larger networks, the total number of nodes is increased to 50. The total number of neighbours will be set to three different values.

First,  $N$  is set to 5, which is an almost constant value in comparison with the  $N$  settings in the 10-node networks from test set 1. Second,  $N$  is set to 10 (20% of the total number of nodes), which yields the same neighbour percentage as  $N=2$  in the 10-node network tests. Finally,  $N$  is set to 20 (40% of the total number of nodes) and can be compared to the  $N=4$  setting in the 10-node network tests. Second, we

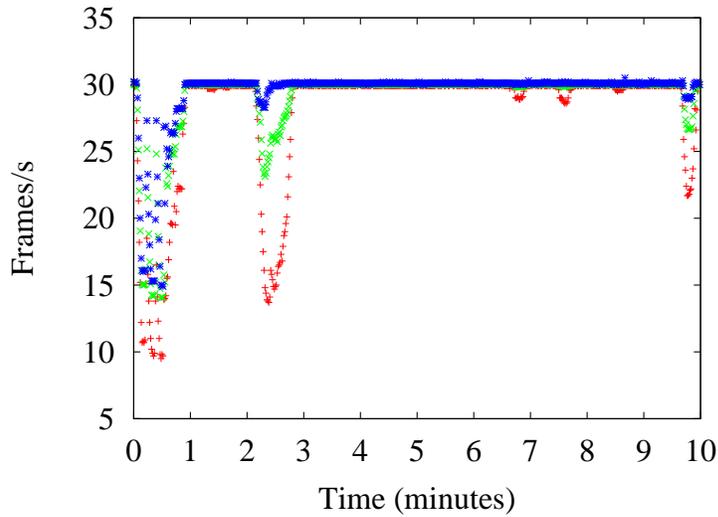


Figure 4.3: Minimum, mean and maximum frame rate for 10-minute test runs at 800 Kbit/s with 3 neighbours and 1 Mbit/s upload limit. (Test Type: 8,  $SSR=2.0$ ,  $N=3$ ,  $UploadLimit=1$  Mbit/s)

Test Type	Video Rate (Kbps)	Audio Rate (Kbps)	# of Peers	SSR	$N$
1	704	96	50	3.0	5
2	704	96	50	3.0	10
3	704	96	50	3.0	20
4	704	96	100	3.0	5
5	704	96	100	3.0	10
6	704	96	100	3.0	20

Table 4.2: The tests of test set 2.

will increase the total number of nodes to 100 while maintaining the number of neighbours at 5, 10, and 20. This leads to 6 tests which are described by Table 4.2. Each of these tests is performed with a client upload limit of 1 Mbit/s and lasts 10 minutes.

#### 4.5.2 Test Results

The results from test types 1, 2 and 3 look promising, and are summarized in Figure 4.4. Since the test results for 5 and 10 neighbours (tests 1 and 2) are similar, we summarize them both in one graph.

The networks with 20 outgoing connections (test type 3) show a slightly better performance than the networks from test 1 and 2. Note that the  $N=20$  networks

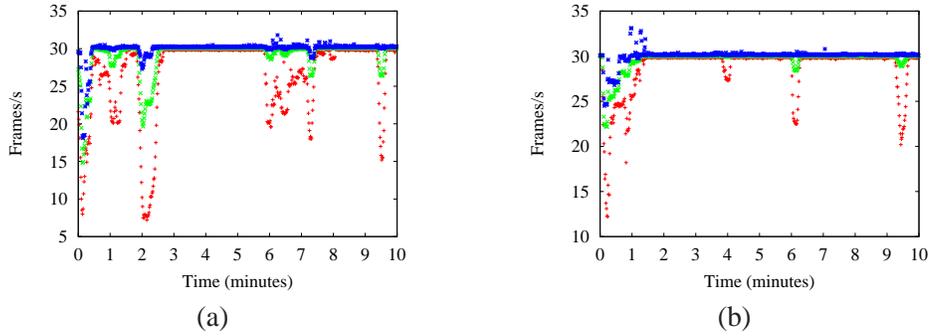


Figure 4.4: Minimum, mean and maximum frame rate for test runs with 50 nodes and 5, 10 (a), and 20 (b) neighbours, with a 1 Mbit/s upload limit. (Test Type: 1,2 (a) and 3 (b),  $SSR=3.0$ ,  $N=5,10$  (a) and  $N=20$  (b),  $UploadLimit=1$  Mbit/s)

have an average of 40 ( $=2N$ ) neighbours, meaning that the network is almost fully connected. Therefore, a node can almost always find the packets that it is missing, since each node knows about almost every other node in the network. This is consistent with the results for the  $N=4$  setting in the 10-node networks of Test Set 1. More interesting are the results of tests 1 and 2. Especially test 1 shows that larger networks can operate well with a small number of neighbours. When we examine the theory this is not completely unexpected. We know that each node in test 1 is connected to 10 ( $=2N$ ) other nodes on average, and has a maximum of 5 outstanding packet requests per neighbour. This means that each node can request a maximum of  $10 \cdot 5 = 50$  packets at a time, which represents just over a second of media playtime. The round-trip time between nodes is set to be a random number between 130 ms and 170 ms, which means that a node is able to request data roughly 6 times faster than it needs to. We therefore expect no problems when we extend the network size to 100 nodes, while keeping the neighbour settings at the same values. Unfortunately, the results from tests 4 to 6 are all equally terrible, and are summarized by Figure 4.5 (a).

These results defy the expectations based on the previous results from the 50-node networks. This could indicate a problem with the distribution algorithm in larger networks. If the problems are caused by the Kettingzaag algorithm itself, we would expect better results when the node bandwidth is increased. Therefore, we decided to rerun the tests without limiting the upload of both the clients and the server, giving each node over 50 Mbit/s of bandwidth. Fortunately these results are just as bad, which means that it is unlikely that the distribution algorithm is the cause of these problems.

Further investigation shows that the server node is suffering from an extremely high CPU load in larger networks. When the server encodes the media stream without any nodes connected, the CPU load ranges from 35% to 40%. The remaining resources are used by a python timer system that allows delayed code execution. These timers are mainly used to play video and audio frames at the correct time,

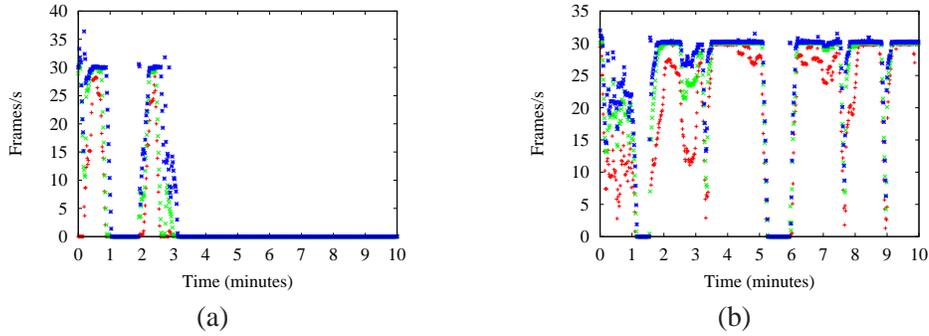


Figure 4.5: Minimum, mean and maximum frame rate for test runs with 100 nodes and 5 neighbours, with a 1 Mbit/s upload limit and 150 (a) or 0 (b) ms round-trip time between nodes. (Test Type: 3, 4 and 5,  $SSR=3.0$ ,  $N=3$ ,  $UploadLimit=1$  Mbit/s,  $RTT=150$  ms (a) and 0 ms (b))

and to delay network packet delivery to simulate round-trip times between nodes. When the number of nodes increases, so does the number of network packets that have to be processed (and delayed) at the server. This leads to a large number of timers, which eventually causes high CPU loads. This prevents the server from inserting new media packets into the network in time. We rerun the  $N=5$  tests with the packet delay system disabled. Since the number of active timers are reduced greatly, we expect better network performance due to less CPU load spikes.

The results of this 0 ms delay test are shown in Figure 4.5 (b). Although CPU spikes still influence network performance, the results are much better than the previous results with packet delaying enabled. In fact, the network performs well from the 3<sup>rd</sup> to the 5<sup>th</sup> minute and from the 6<sup>th</sup> to the 10<sup>th</sup> minute. In these intervals the server CPU load never reached 100%, which leads us to believe that further optimizations would yield to results equal to those of the 50-node networks. However, we will not further optimize the software for two reasons. First, we are close to the maximum number of nodes that we can simulate on the Delft cluster of the DAS-3 supercomputer. So even if the server could handle more nodes, we would not be able to simulate them on the Delft cluster. Using other DAS-3 cluster sites is possible, but starting them at about the same time is extremely difficult (if not impossible) due to job scheduling on each cluster. The second reason is that our research is superseded by a similar research project in the PDS group that uses a give-to-get overlay closely related to the BitTorrent protocol [23], just like [16]. The biggest advantage of this give-to-get overlay is that it implements protection against free-riding (nodes that download packets, but not upload any), something which is not present in our current implementation of Kettingzaag. Since the P2P principle is based on nodes exchanging packets amongst each other, this is an important feature [18]. When too many free-riders enter the network, Kettingzaag performance will drop because the well behaving nodes (the ones that do upload) cannot provide enough upload bandwidth to compensate the free-riding nodes [11].

## 4.6 Conclusion

The system works quite well for a 10-node network for SSR values of 2.5 and up. Even for  $N=2$ , the frame rate rarely drops below 25 fps. For SSR values below 2.0, the results are too unstable to be used in a real environment. Although the initial tests showed bad results for an SSR setting of 2.0, the extended tests show that this is the lowest reasonable SSR value for longer runs. However, the startup time needs to be much improved before an SSR setting of 2.0 can be used in a real situation. We have to conclude that the system does not work well for a 10-node network with  $N=2$  at this setting. The question if this is a result of the low number of outgoing connections, or if a value of  $N$  equivalent to 20% of the total number of nodes is a too low setting for  $N$  is answered by Test Set 2.

The goal in our second test was to find out if the SSR and neighbour settings should depend on the total number of nodes in the network. According to the test results of both the 50-node and the optimized 100-node tests, we can conclude that it is very unlikely that such a dependency exists. Even though we have not tested beyond network sizes of 100 nodes, we know that with  $N=5$  and  $RTT=150$  ms, nodes can request packets up to 6 times faster than required. We therefore do not expect problems with much bigger network sizes, as long as the client nodes keep uploading. Of course, higher values for the SSR and  $N$  will smoothen the occasional frame rate drop, and make the network more resilient against node failures. Unfortunately, we cannot simulate large enough networks to make any meaningful statements about network performance during node failures.

## Chapter 5

# Conclusions and Future Work

Content providers are becoming more and more aware of the high server costs that are inherent to the client/server model for live video streaming. The solution to this problem lies in peer-to-peer technology. The reason is that clients distribute the stream amongst each other, which reduces the amount of data that is downloaded from the content servers. Many P2P solutions exist, but we have chosen to implement our own algorithm called Kettingzaag, which is based on the Chainsaw algorithm [25] because of its simplicity and excellent simulation results.

Chainsaw is a fairly new swarm-based content distribution algorithm that makes use of local state gossiping and data pulling. The main advantage of swarm-based overlay networks over their tree- and flooding-based counterparts is that they do not enforce a strict network topology. This makes them very resilient to errors. We have added some improvements to our Kettingzaag algorithm, such as *REJECT* messages, ping-pong messages and multiple description coding. We intended to test Kettingzaag in our own video player called Lumberjack on Delft-37, a collection of virtual hosts on the Internet.

Delft-37 consists of 6 hosts and was intended to grow to 37 hosts. In order to make Delft-37 easy to use and manage, we have implemented a program that allows us to control the network and its contents from a central control point. We also added a few reliable machines to the university network to host Lumberjack seeds and to store test logs. Unfortunately, Delft-37 performance tests show that the nodes do not have enough bandwidth and processing power available to host multiple Lumberjack instance. Therefore, the idea has been abandoned and we have moved our tests to the DAS-3 supercomputer.

### 5.1 Conclusions

We started our thesis with the following research question: How does a Chainsaw-based implementation perform in a real video player and network, and in particular does it perform as well in a real environment as it does in simulations? We can conclude that our Kettingzaag implementation performs quite well for relatively

small networks. Under normal operation, Kettingzaag manages to deliver the video stream with minimal packet loss at low server seeding rates. However, the first minute after network startup remains problematic for Kettingzaag. If this is also the case in the Chainsaw simulations remains unclear, because its designers state that their network gets time to initialize, without specifying further information. We do believe that Chainsaw still has better network startup times, which is most likely caused by the fact that new Chainsaw nodes can only request packets from neighbours. In Kettingzaag, new nodes can also request packets from the seed, which results in new nodes receiving very new packets due to the seed's packet overriding mechanism.

The second question we asked ourselves in the problem statement is: Do we need changes to the original Chainsaw algorithm to create a working implementation? We found that the Chainsaw description explained in [25] performs alright. However, implementing the *REJECT* message as described in Chapter 2 improved Kettingzaag performance significantly. Although the Chainsaw designers do not discuss a *REJECT* message or another hold-off mechanism, we cannot imagine that they just silently drop requests.

The final question in our problem statement is: Which parameters are the most important in Kettingzaag, and what are good settings for various network sizes? We can conclude that the video bitrate and the mean upload rate in the network are the most important parameters. The video bitrate should not exceed 80% of the mean upload rate in the network. We have shown that the server seeding ratio and the number of initiated neighbour connections do not have to depend on the total number of nodes in small networks. Although the implementation of Lumberjack prevents us from testing networks larger than a hundred nodes, we are fairly certain that a server seeding ratio of 3.0 and 20 initiated neighbour connections are enough for large networks as well.

## 5.2 Future Work

One of the most important issues with our implementation is the fact that it uses a lot of CPU resources, which prevents us from testing network sizes larger than a hundred nodes. In fact, the 100-node network tests failed at first. To address this issue, the python timer implementation should be redesigned and real-time MDC video encoding should be replaced by a pre-encoded stream saved on disk. These changes will allow an increase of the maximum network size and of the number of different videos that can be seeded on a single node.

We have noticed that the most prominent difference between Chainsaw and our Kettingzaag implementation is the startup time. Although the Chainsaw simulation results were taken after the network had time to initialize, we do believe that Chainsaw performs better than Kettingzaag in terms of startup time. In the Chainsaw algorithm, new nodes are encouraged to request packets from neighbours only. This prevents new nodes from receiving overridden packets from the server, which

are most likely to be far away from the playback position. Since Kettingzaag does not do this, new nodes will download the newest packets from the server as well, wasting bandwidth for the packets that they really need.

Another important issue is that we assumed that all nodes donate upload bandwidth to the network. In reality however, many nodes will not do this (free-riding), which negatively affects network performance. In order to minimize the negative effects of free-riding, measures must be implemented that penalize free-riding nodes when the total bandwidth demand exceeds the total bandwidth supply.

In order to improve the performance of Kettingzaag even more, we suggest two changes. First, the mechanism that determines which packets should be downloaded is now based on random selection. This could lead to packet loss when a very new packet is chosen instead of a packet that is about to be played. A more intelligent solution would be to implement an algorithm that prioritizes packets that are within a certain range from the playback position. Second, the maximum number of parallel requests from a node to a single neighbour could be implemented in a better way. For Chainsaw, the authors suggest that the requesting node keeps track of the maximum number of outstanding requests per neighbour. Since our software is open source, everyone can change this parameter at will, which would result in altered nodes that request many packets at the same time. Instead, the maximum number of parallel requests should be implemented in the nodes receiving the requests, for this allows the receiver to choke modified nodes when necessary.



# Bibliography

- [1] C. Diot, B.N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment Issues for the IP Multicast Service and Architecture. *IEEE Network* Vol. 14, Issue 1, p78-88, IEEE, Jan. 2000.
- [2] C. Huang, J. Li, K.W. Ross. Can Internet Video-on-Demand be Profitable? *Computer Communication Review* Vol 37, Issue 4, p133-144, ACM SIGCOMM, Oct. 2007.
- [3] D. DeFigueiredo, B. Venkatachalam, S.F. Wu. Bounds on the Performance of P2P Networks Using Tit-for-Tat Strategies. In *Proceedings of the Seventh International Conference on Peer-to-Peer Computing* Vol 9, p11-18, IEEE, Sep. 2007.
- [4] D.A. Tran, K.A. Hua, and T.T. Do. Zigzag: An Efficient Peer-to-Peer Scheme for Media Streaming. In *Proceedings of IEEE INFOCOM* Vol. 2, p1283-1292, Mar. and Apr. 2003.
- [5] E. Mota-Garcia, R. Hasimoto-Beltran. A fast scheme for simple geographic Internet mapping. *Sixth Mexican International Conference on Computer Science* p230-234, IEEE, Sep. 2005.
- [6] Ellacoya Networks, Inc. <http://www.ellacoya.com/news/pdf/2007/NXTcommEllacoyaMediaAlert.pdf>. Website, 2007.
- [7] I-Share. <http://www.freeband.nl/project.cfm?language=en&id=520>. Website, 2005.
- [8] IPerf. <http://dast.nlanr.net/Projects/Iperf/>. Website, 2006.
- [9] J.A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H.J. Epema, M. Reinders, M.R. van Steen, H.J. Sips. *Tribler: A social-based peer-to-peer system. Concurrency and Computation: Practice and Experience* Vol. 20, p127-138, 2008.
- [10] J.J.D. Mol, D.H.J. Epema, H.J. Sips. The Orchard Algorithm: Building Multicast Trees for P2P Video Multicasting without Free-Riding. *IEEE Transactions on Multimedia* Vol. 9, Issue 8, p1593-1604, IEEE, 2007.
- [11] J.J.D. Mol, J.A. Pouwelse, D.H.J. Epema, H.J. Sips. Free-Riding, Fairness, and Firewalls in P2P File-Sharing. *The Eight International Conference on Peer-to-Peer Computing (P2P'08)* p301-310, IEEE, Sep. 2008.
- [12] J.J.D. Mol, J.A. Pouwelse, D.H.J. Epema, H.J. Sips. Give-to-Get: Free-Riding Resilient Video-on-Demand in P2P Systems. *Fifteenth Annual Multimedia Computing and Networking (MMCN'08)*, IEEE, Jan. 2008.
- [13] M. Castro, P. Druschel, A.M. Kermarrec, and A. Rowstron. Scribe: A large-scale and Decentralized Application-Level Multicast Infrastructure. *IEEE Journal on Selected Areas in Communications* Vol. 20, Issue 8, p1489-1499, Oct. 2002.
- [14] M. Rui, F. Labeau. Error-Resilient Multiple Description Coding. *IEEE Transactions on Signal Processing* Vol. 56, Issue 8, p3996-4007, IEEE, Aug. 2008.
- [15] P. Karwaczynski. Fabric: Synergistic Proximity Neighbour Selection Method. In *Proceedings of the Seventh International Conference on Peer-to-Peer Computing* Vol 9, p229-230, IEEE, Sep. 2007.

- [16] P. Shah, J.F. Paris. Peer-to-Peer Multimedia Streaming Using BitTorrent. *Performance, Computing, and Communications Conference (IPCCC)* p340-347, IEEE, Apr. 2007.
- [17] PlanetLab. <http://www.planet-lab.org/>. Website, 2004.
- [18] R. Krishnan., M.D. Smith., Z. Tang., R. Telang. The Impact of Free-Riding on Peer-to-Peer Networks. *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, Jan. 2004.
- [19] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM* p205-217, Aug. 2002.
- [20] S. Ratnasami, M. Handley, and R. Karp. Application Level Multicast using Content Addressable Networks. In *Proceedings of Third International Workshop on Networked Group Communication (NGC '01)* p14-29, London, England, Aug. 2001.
- [21] S. Ratnasami, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proceedings of ACM SIGCOMM* p161-172, Aug. 2001.
- [22] Tribler. <http://www.tribler.org/>. Website, 2005.
- [23] Tribler. <https://www.tribler.org/StreamingExperiment>. Website, 2008.
- [24] TU-Delft. <http://www.cs.vu.nl/das3/>. Website, 2007.
- [25] V. Pay, K. Kumar, K. Tamilmani, V. Sambamurthy, and A.E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *Proceedings of the Fourth International Workshop on Peer-to-Peer Systems (IPTPS)* p127-140, Stony Brook University, Feb. 2005.
- [26] V.N. Padmanabhan, H.J. Wang, and P.A. Chou. Resilient Peer-to-Peer Streaming. *Technical Report MSR-TR-2003-11*, Microsoft Research, Nov. 2003.
- [27] X. Zhang, J. Liu, B. Li, T.S.P. Yum. Coolstreaming/DONNet: A Data-Driven Overlay Network for Efficient Live Media Streaming. In *Proceedings of IEEE INFOCOM* Vol. 3, p2103-2111, Mar. 2005.

## Appendix A

# Delft-37 Test Results

This appendix shows the Delft-37 bandwidth results that we described in Chapter 3. Every figure contains both the TCP and UDP bandwidth results from one node in the network to all of the other nodes. The top graphs represent the download speed obtained using the TCP protocol, and the bottom graphs show the download speed obtained using the UDP protocol. For UDP we have chosen to create a stream of 1 Mbit/s. The UDP graphs show how many kbits arrive each second at the downloading node. Note that the same symbols in the legend of the graphs do not always represent the same nodes. Table A.1 contains the nodes that are present in the Delft-37 network.

Full Name	IP Address	Location
vds-355074.amen-pro.com	62.193.219.68	Paris, France
advantagecom.us.peer-2-peer.org	66.29.146.21	Walla Walla, Washington, USA
d80-237-144-205.dds.hosteurope.de	80.237.144.205	Köln, Germany
adiungo-phoenix.us.peer-2-peer.org	193.192.247.157	Phoenix, Arizona, USA
adiungo-london.uk.peer-2-peer.org	193.192.247.133	London, UK
usonyx.sp.peer-2-peer.org	202.172.255.90	Singapore

Table A.1: Delft-37 nodes with IP address and location.

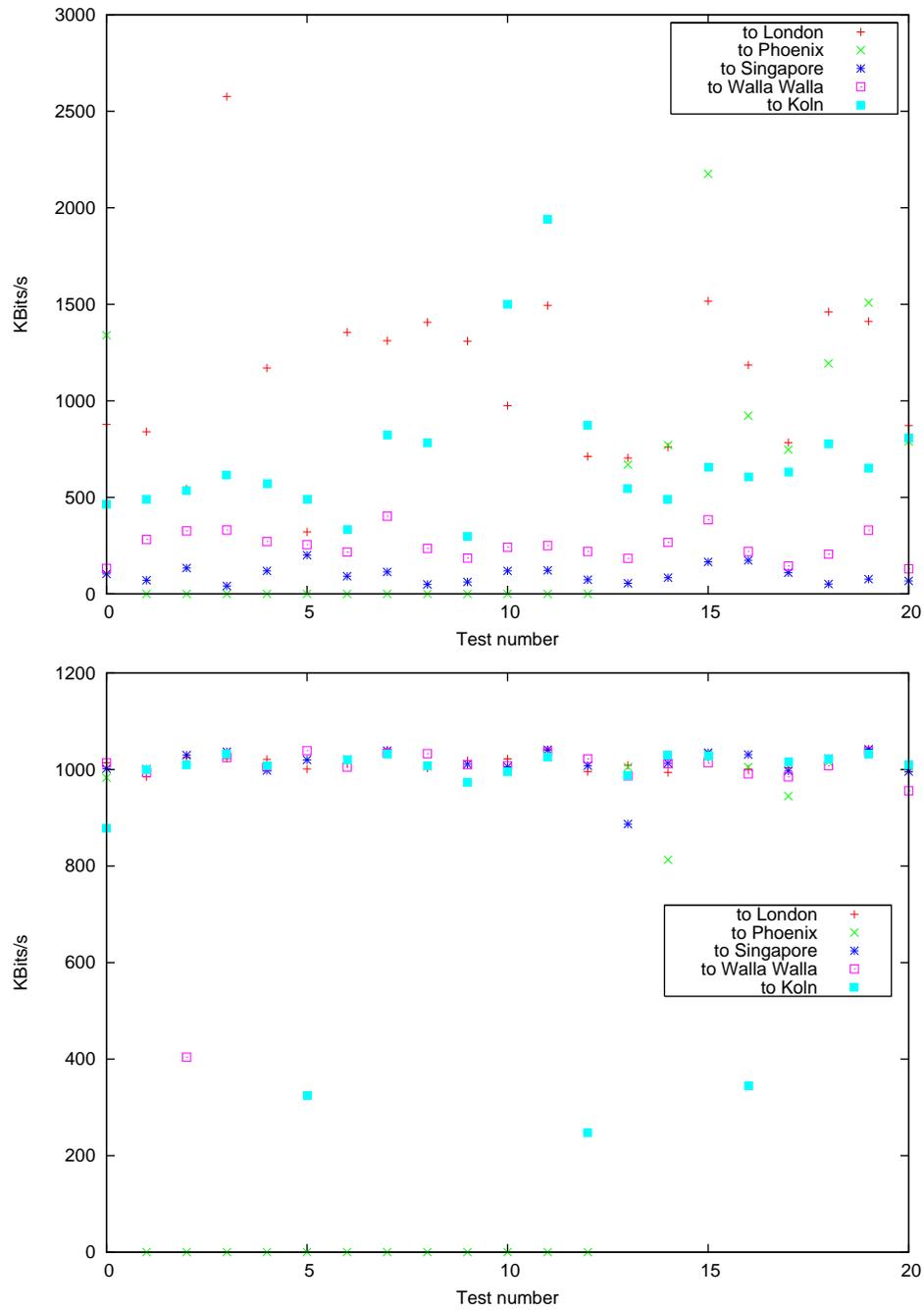


Figure A.1: TCP (top) and UDP (bottom) bandwidth test results for Paris.

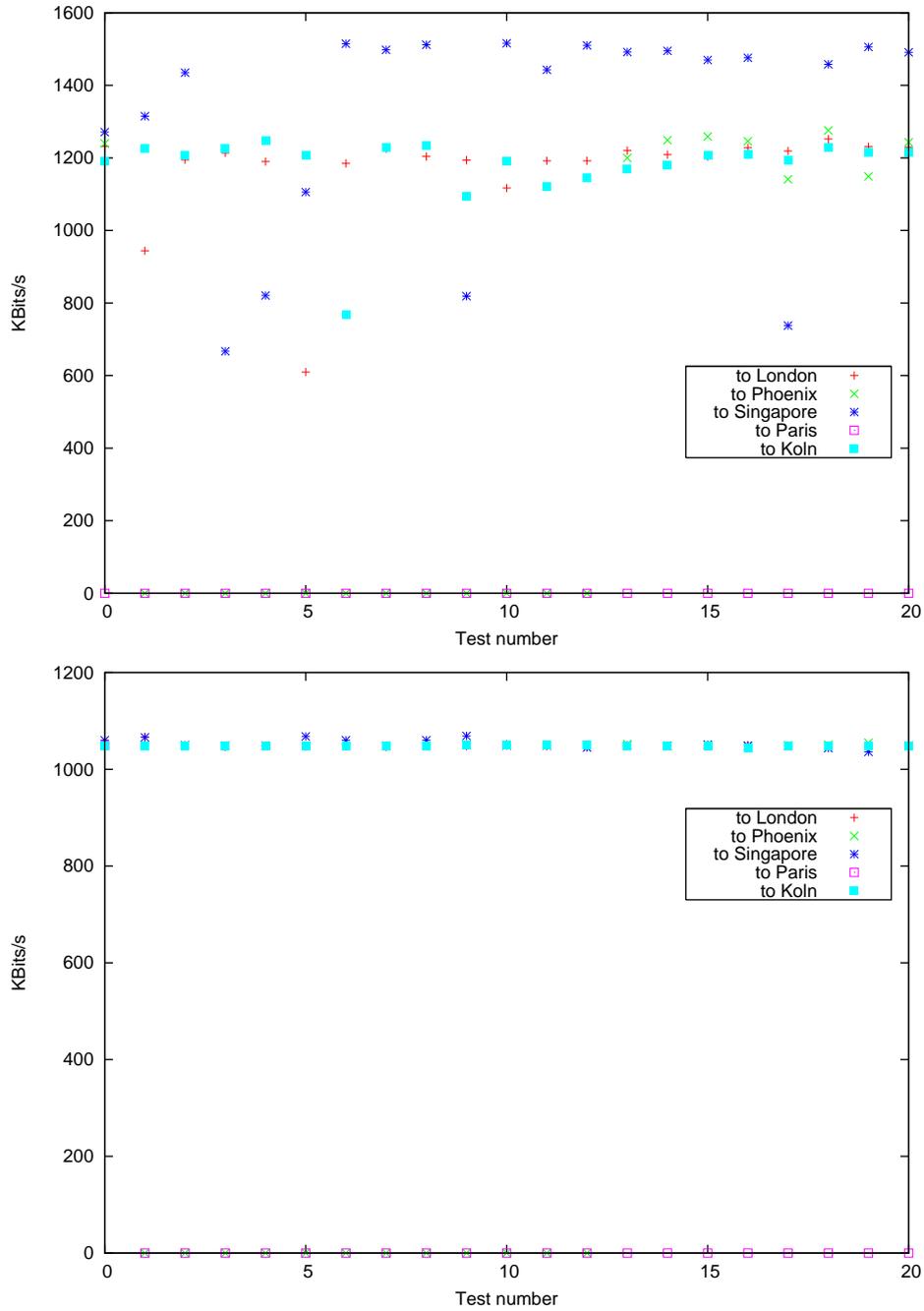


Figure A.2: TCP (top) and UDP (bottom) bandwidth test results for Walla Walla.

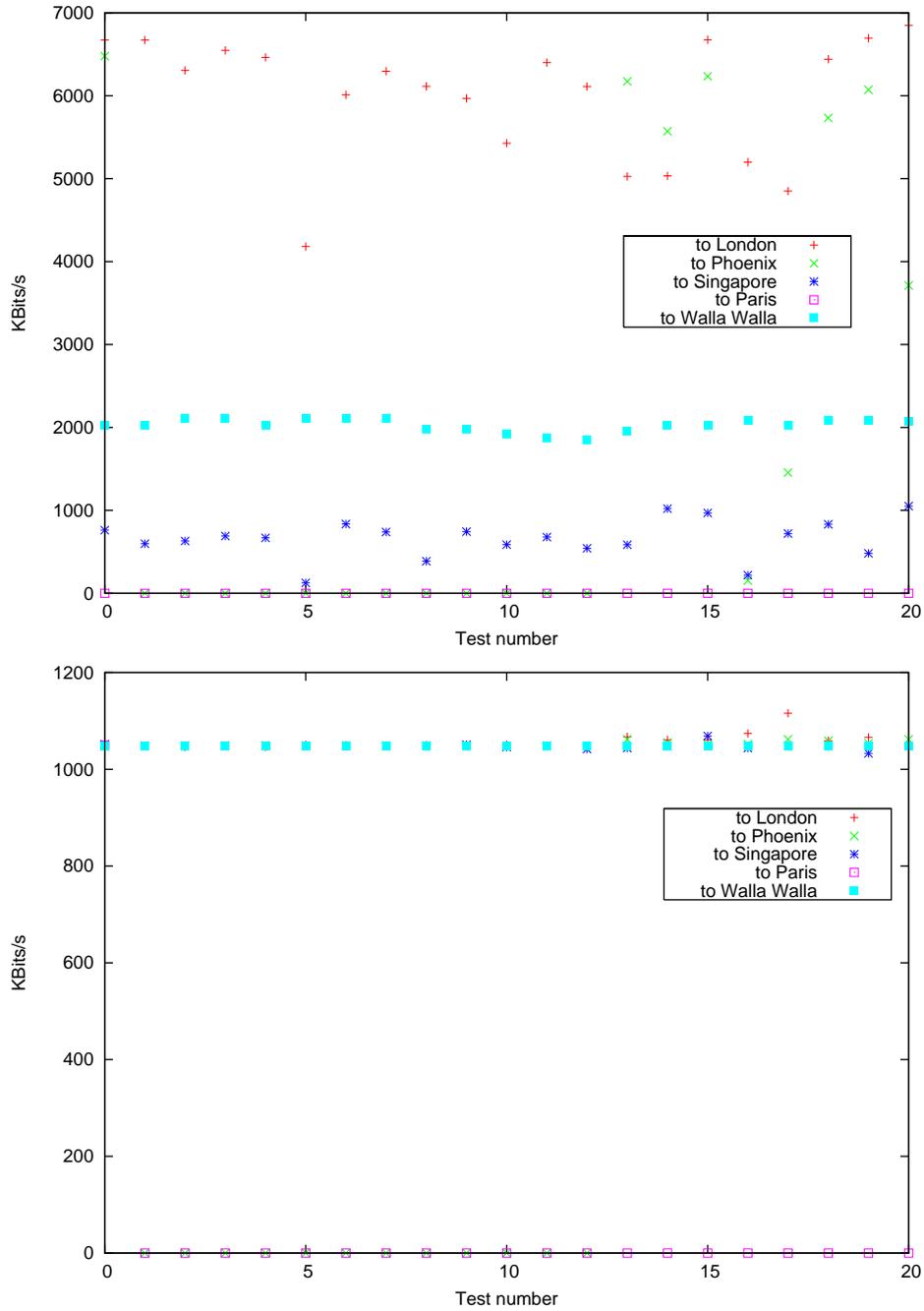


Figure A.3: TCP (top) and UDP (bottom) bandwidth test results for Köln.

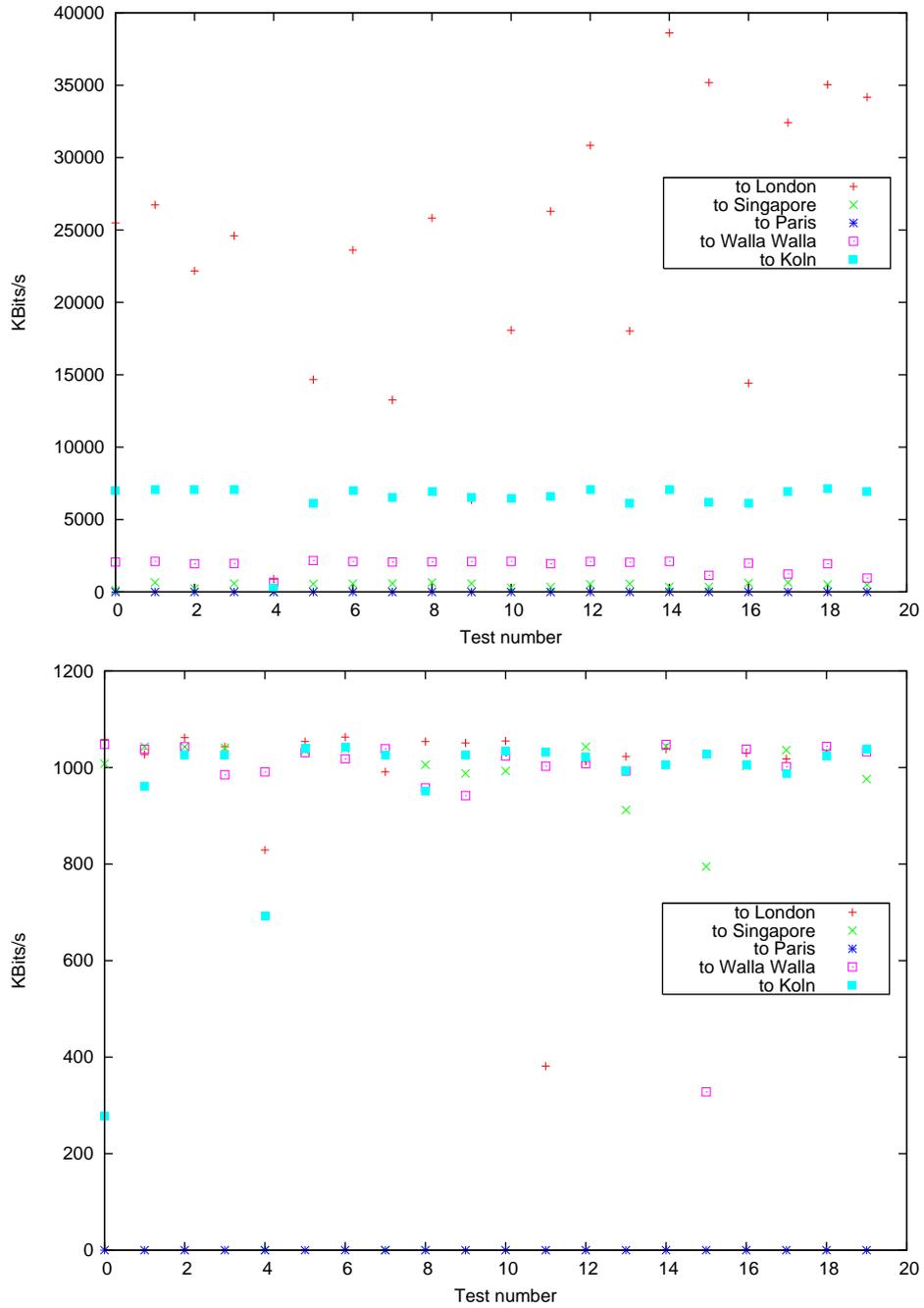


Figure A.4: TCP (top) and UDP (bottom) bandwidth test results for Phoenix.

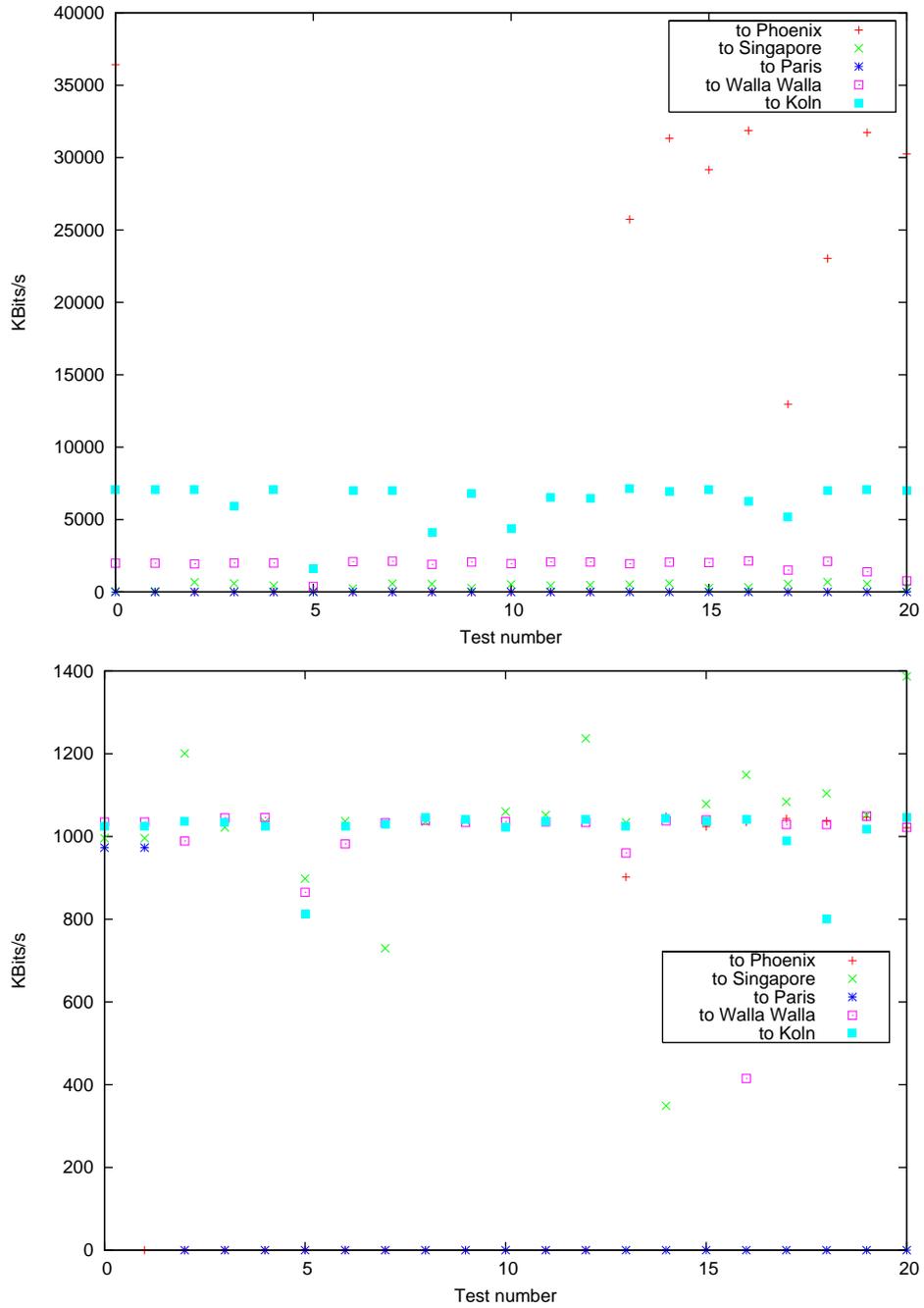


Figure A.5: TCP (top) and UDP (bottom) bandwidth test results for London.

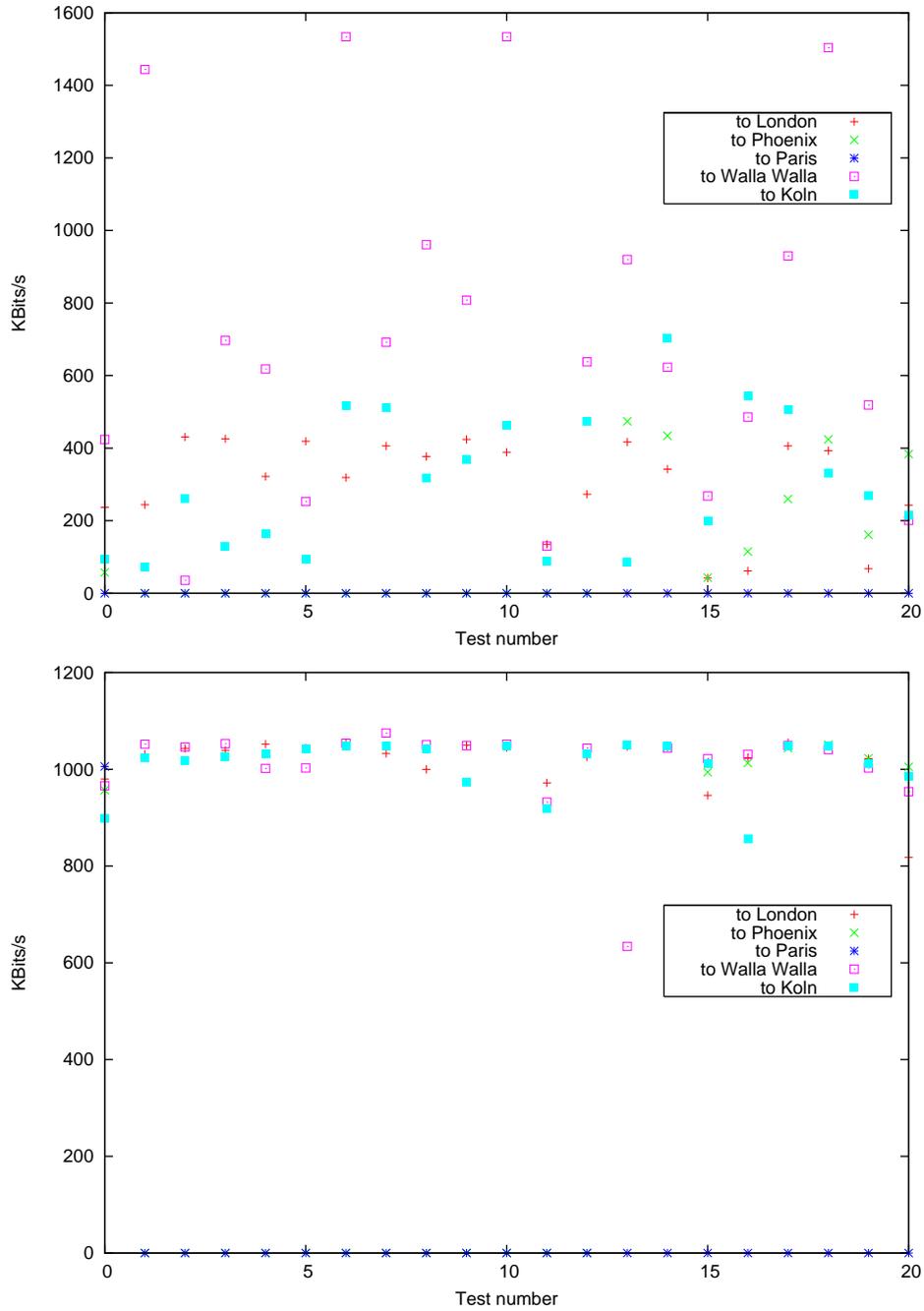


Figure A.6: TCP (top) and UDP (bottom) bandwidth test results for Singapore.



## Appendix B

# Lumberjack Test Results

In this appendix we present the test results of test set 1, which is described in Chapter 4. The tests in test set 1 are summarized once more in Table B.1. Although the table contains 15 different test types, they are all performed in three test runs in which we run five test types each. This means that we dynamically change the SSR value (from 1.1, 1.5, 2.0, 2.5 to 3.0) in each test run to accommodate all test types. All graphs in this appendix represent an execution of a complete test run, which means that the horizontal axis holds all of the possible SSR values. The total duration of a test run is 300 seconds, which means that every SSR interval lasts 60 seconds. We have defined four performance metrics, which are the frame rate, the frame loss, the upload rate and the download rate. Each test run is executed ten times, and all of these executions are shown in a single figure for each performance metric. Hence, the results of every test run is presented in four figures (one for each metric), which each contain 10 graphs (one for each execution). Test set 1 is performed with both unlimited and limited upload settings for the client nodes in the network. The unlimited results are shown in Section B.1, and the limited results are shown in Section B.2.

The test runs are executed on 10-node networks, which means that every figure in this appendix should contain 10 data points for each timestamp. We have chosen to keep the figures organized by only showing the minimum, mean and maximum values of the ten nodes for each of the 300 timestamps (60 seconds for each of the five SSR settings). Table B.2 shows that the minimum node value is represented by the red data points, the mean node value by the green data points, and the maximum node value by the blue data points.

### B.1 Lumberjack Unlimited Client Upload Results

This section contains the twelve (three test runs multiplied by four performance metrics) figures belonging to the three test runs described by Table B.1. The upload rate of the client nodes is not limited in these tests.

Test Run	Test Type	Video Rate (Kbps)	Audio Rate (Kbps)	# of Peers	SSR	$N$
1	1	704	96	10	1.1	2
	2	704	96	10	1.5	2
	3	704	96	10	2.0	2
	4	704	96	10	2.5	2
	5	704	96	10	3.0	2
2	6	704	96	10	1.1	3
	7	704	96	10	1.5	3
	8	704	96	10	2.0	3
	9	704	96	10	2.5	3
	10	704	96	10	3.0	3
3	11	704	96	10	1.1	4
	12	704	96	10	1.5	4
	13	704	96	10	2.0	4
	14	704	96	10	2.5	4
	15	704	96	10	3.0	4

Table B.1: Tests of test set 1.

	Maximum data value across all nodes
	Mean data value across all nodes
	Minimum data value across all nodes

Table B.2: Legend of the data point colors in this appendix.

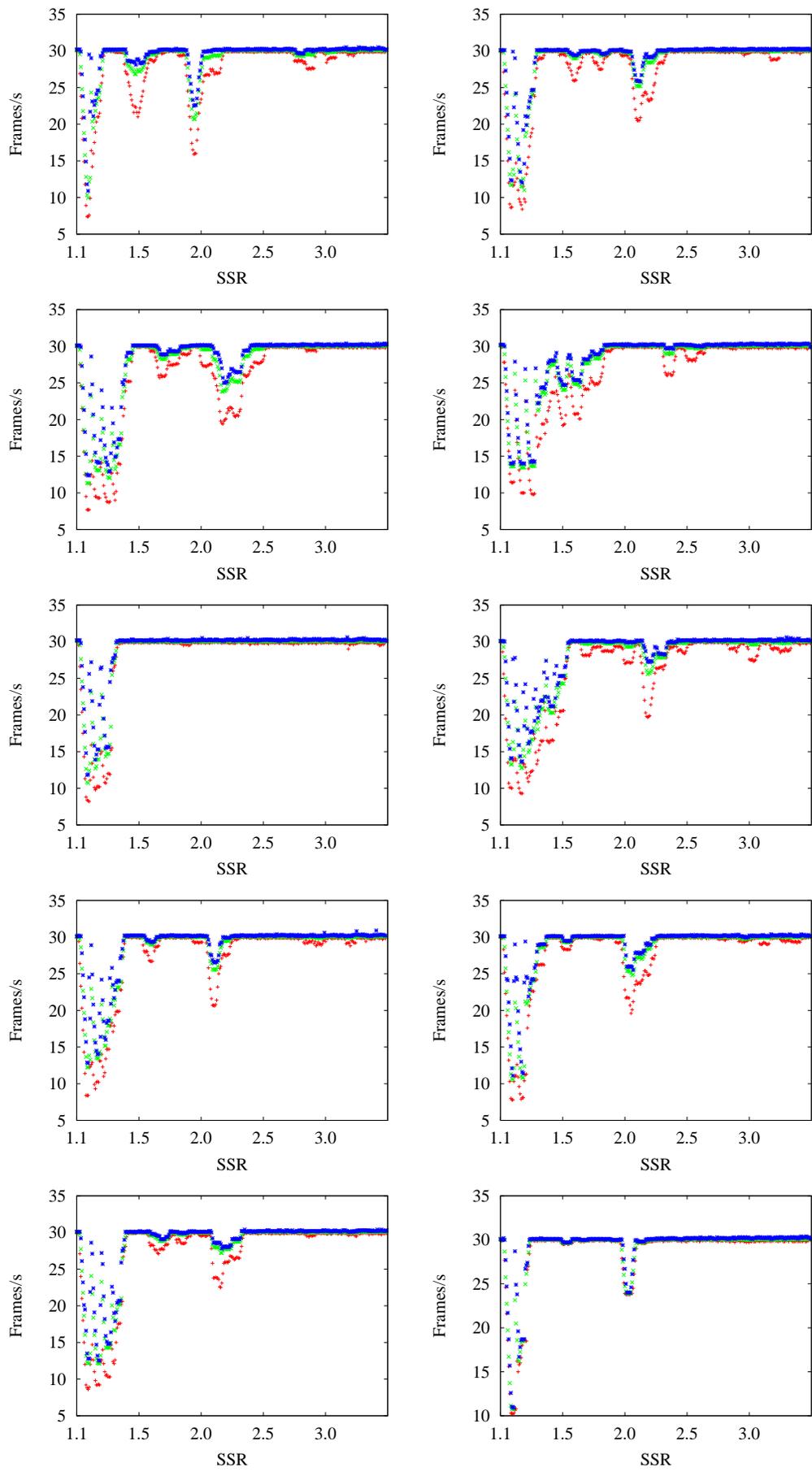


Figure B.1: Minimum, mean and maximum *frame rate* for ten executions of test run 1 at 800 Kbps with two neighbours and no upload limit. (Test Run: 1,  $SSR=1.1-3.0$ ,  $N=2$ ,  $UploadLimit=unlimited$ )

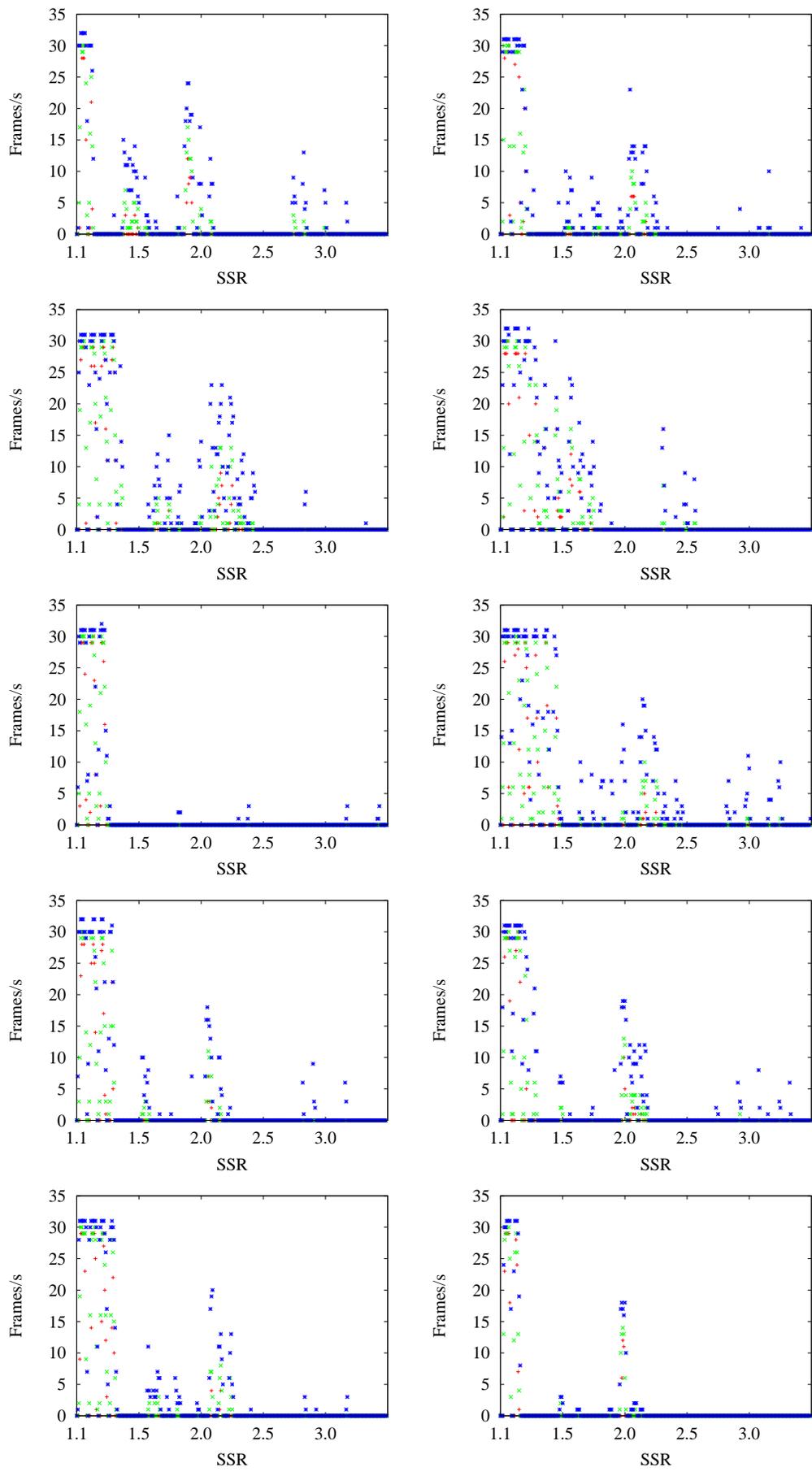


Figure B.2: Minimum, mean and maximum *frame loss* for ten executions of test run 1 at 800 Kbps with two neighbours and no upload limit. (Test Run: 1,  $SSR=1.1-3.0$ ,  $N=2$ ,  $UploadLimit=unlimited$ )

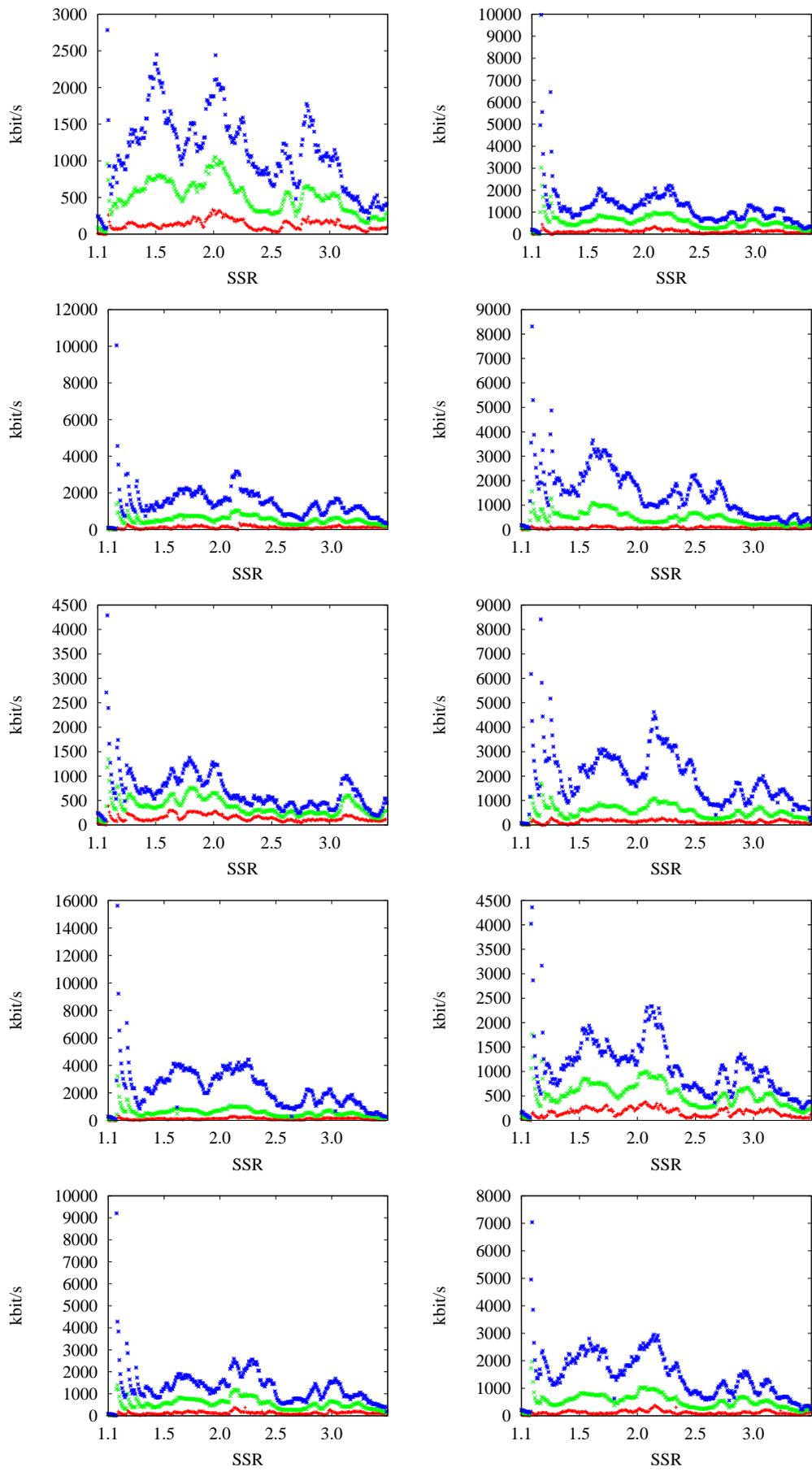


Figure B.3: Minimum, mean and maximum *upload rate* for ten executions of test run 1 at 800 Kbps with two neighbours and no upload limit. (Test Run: 1,  $SSR=1.1-3.0$ ,  $N=2$ ,  $UploadLimit=unlimited$ )

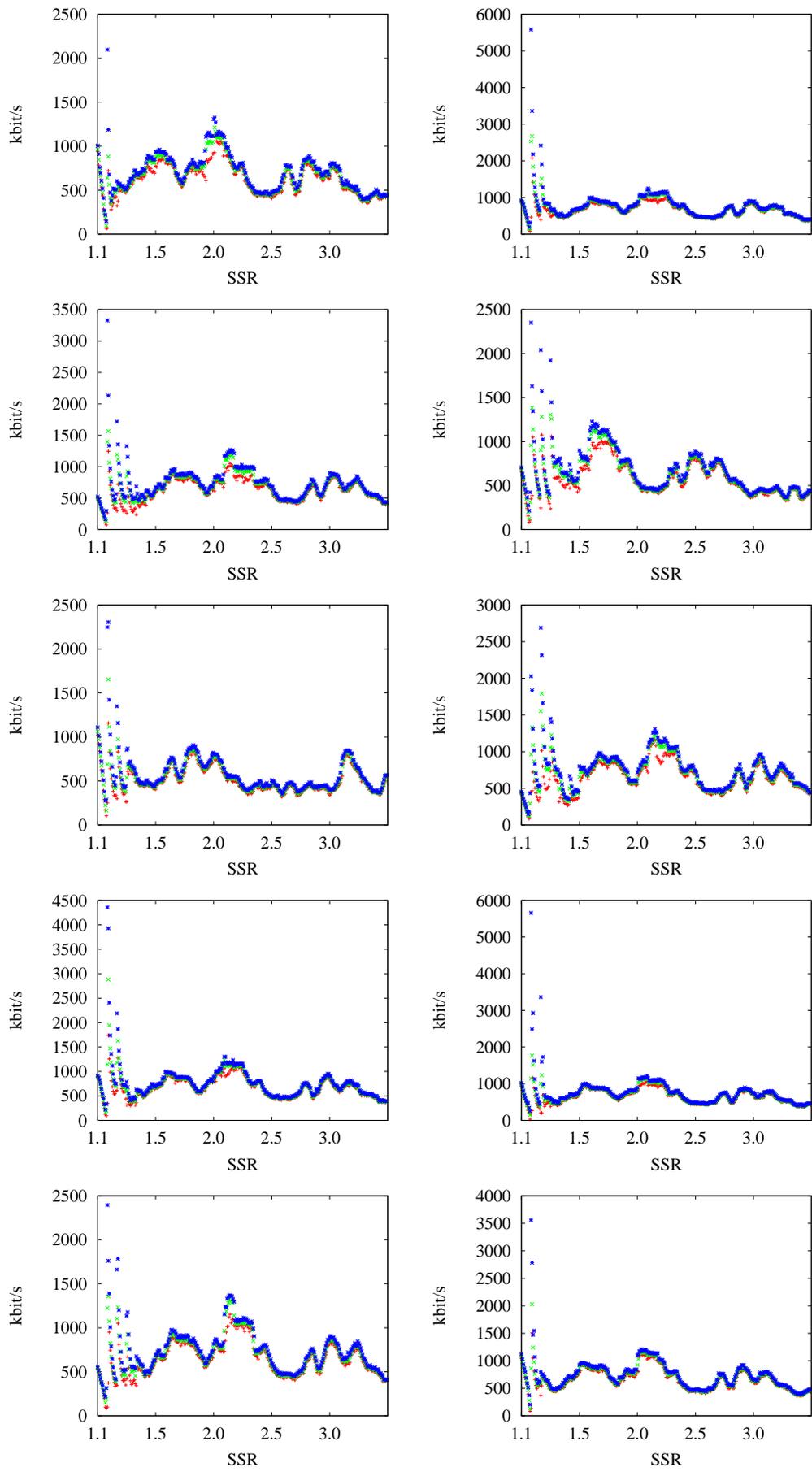


Figure B.4: Minimum, mean and maximum *download rate* for ten executions of test run 1 at 800 Kbps with two neighbours and no upload limit. (Test Run: 1,  $SSR=1.1-3.0$ ,  $N=2$ ,  $UploadLimit=unlimited$ )

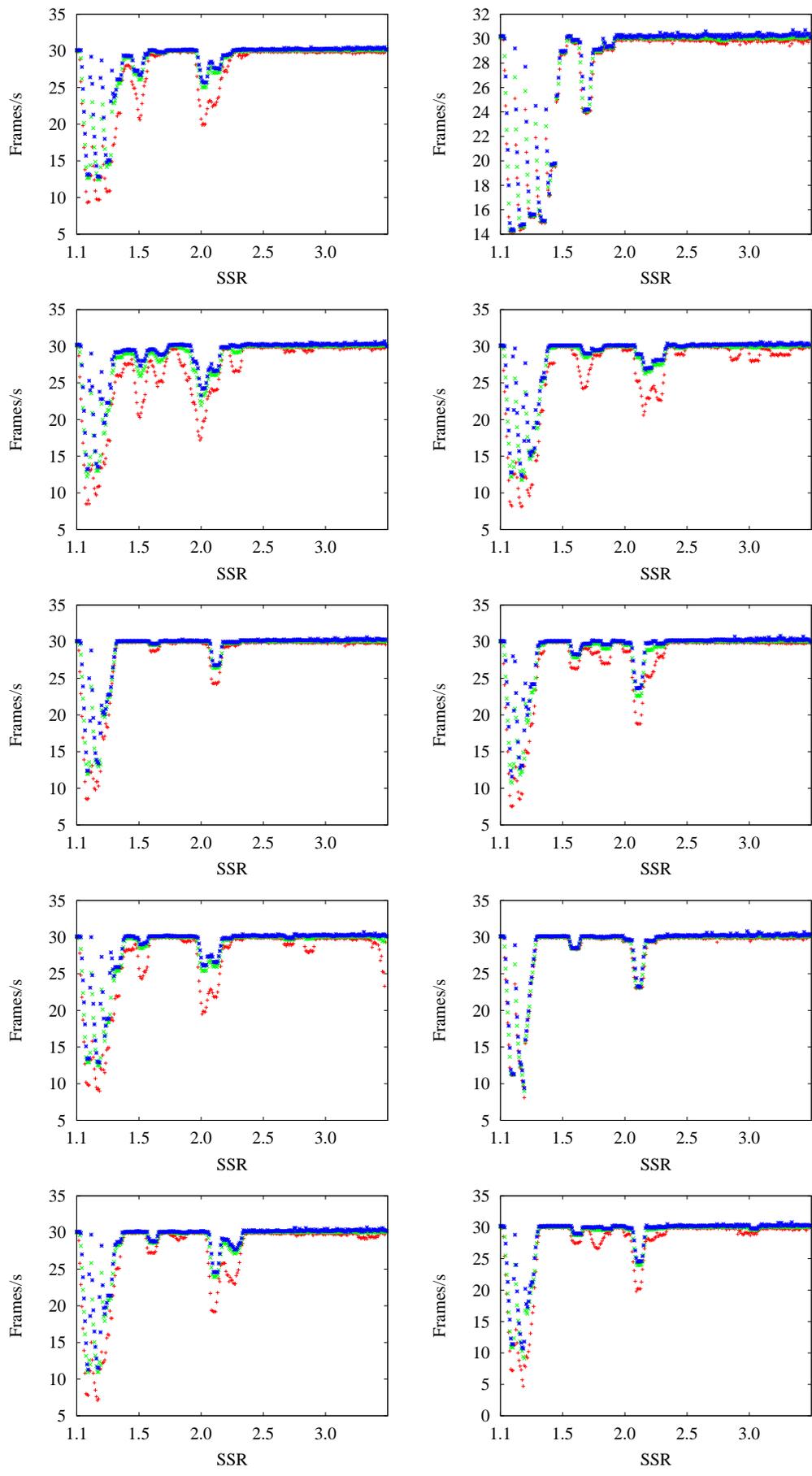


Figure B.5: Minimum, mean and maximum *frame rate* for ten executions of test run 2 at 800 Kbps with three neighbours and no upload limit. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=unlimited$ )

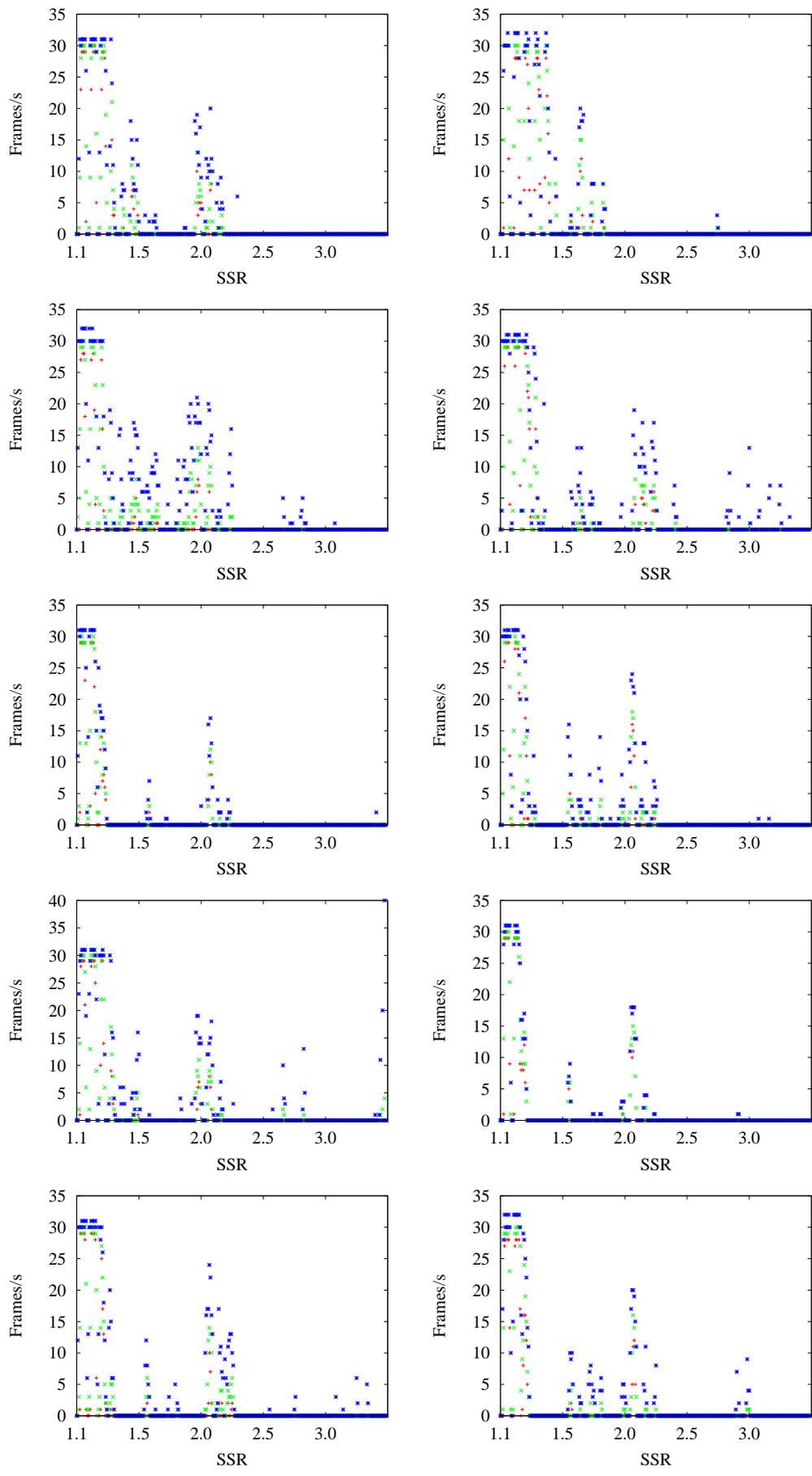


Figure B.6: Minimum, mean and maximum *frame loss* for ten executions of test run 2 at 800 Kbps with three neighbours and no upload limit. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=unlimited$ )

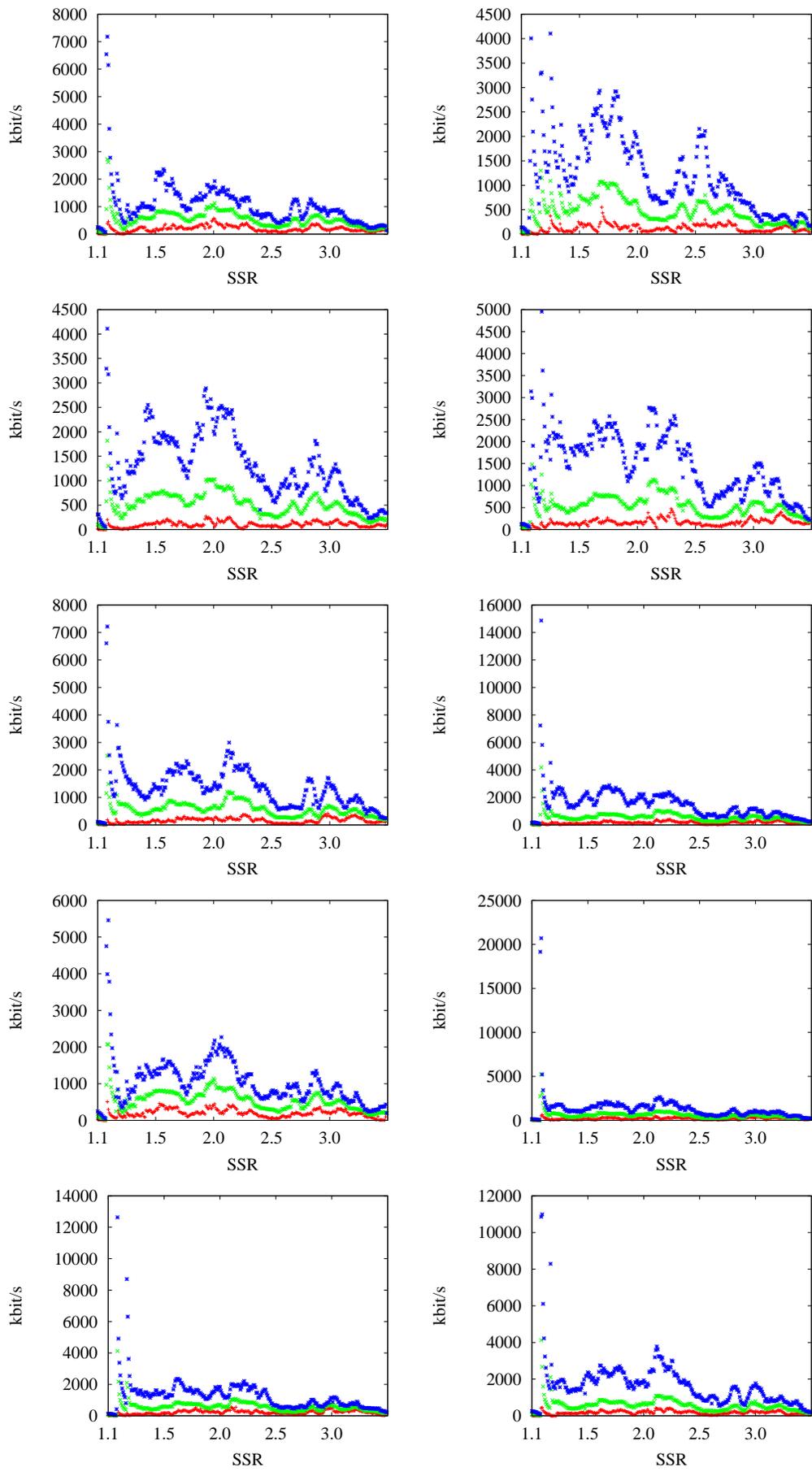


Figure B.7: Minimum, mean and maximum *upload rate* for ten executions of test run 2 at 800 Kbps with three neighbours and no upload limit. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=unlimited$ )

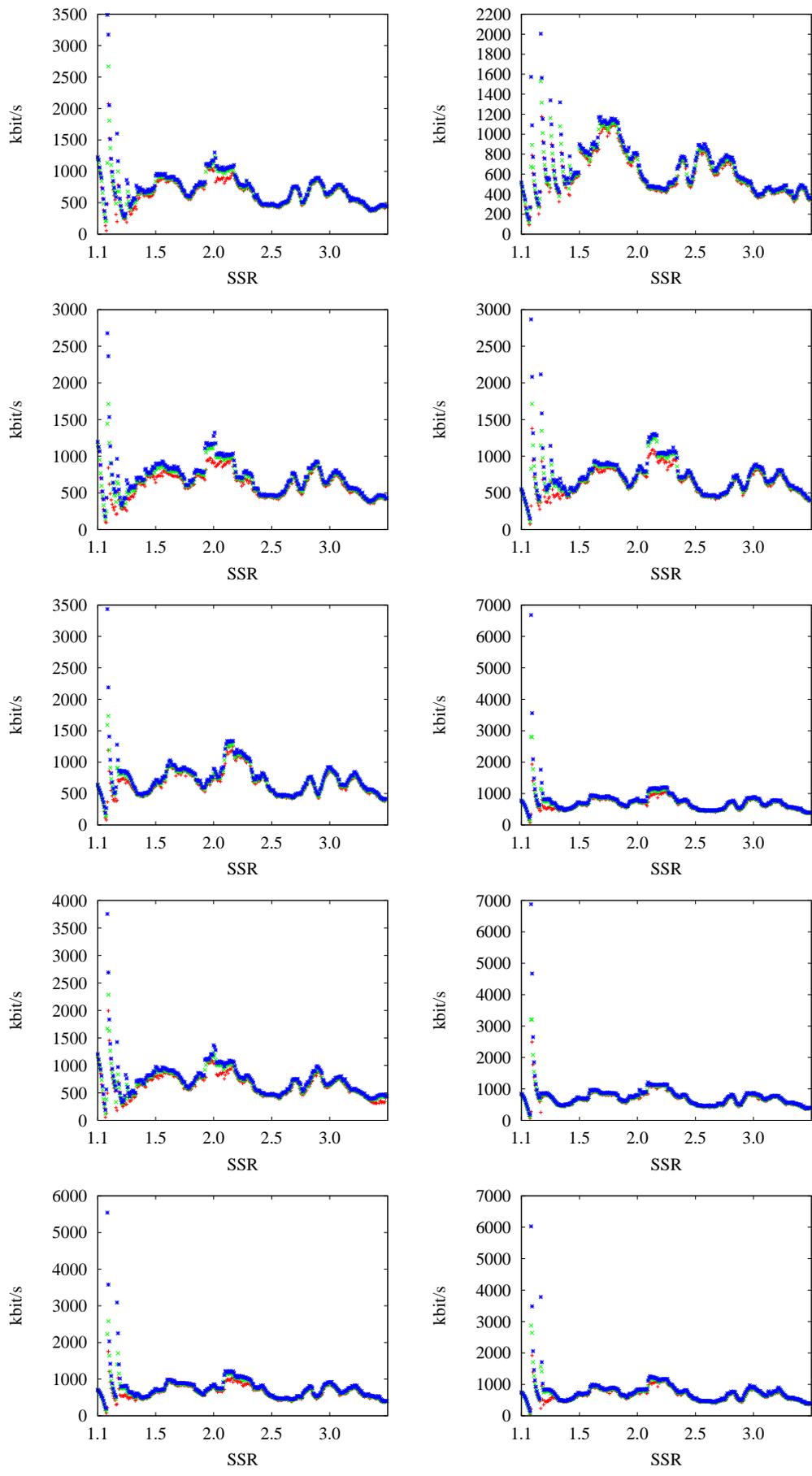


Figure B.8: Minimum, mean and maximum *download rate* for ten executions of test run 2 at 800 Kbps with three neighbours and no upload limit. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=unlimited$ )

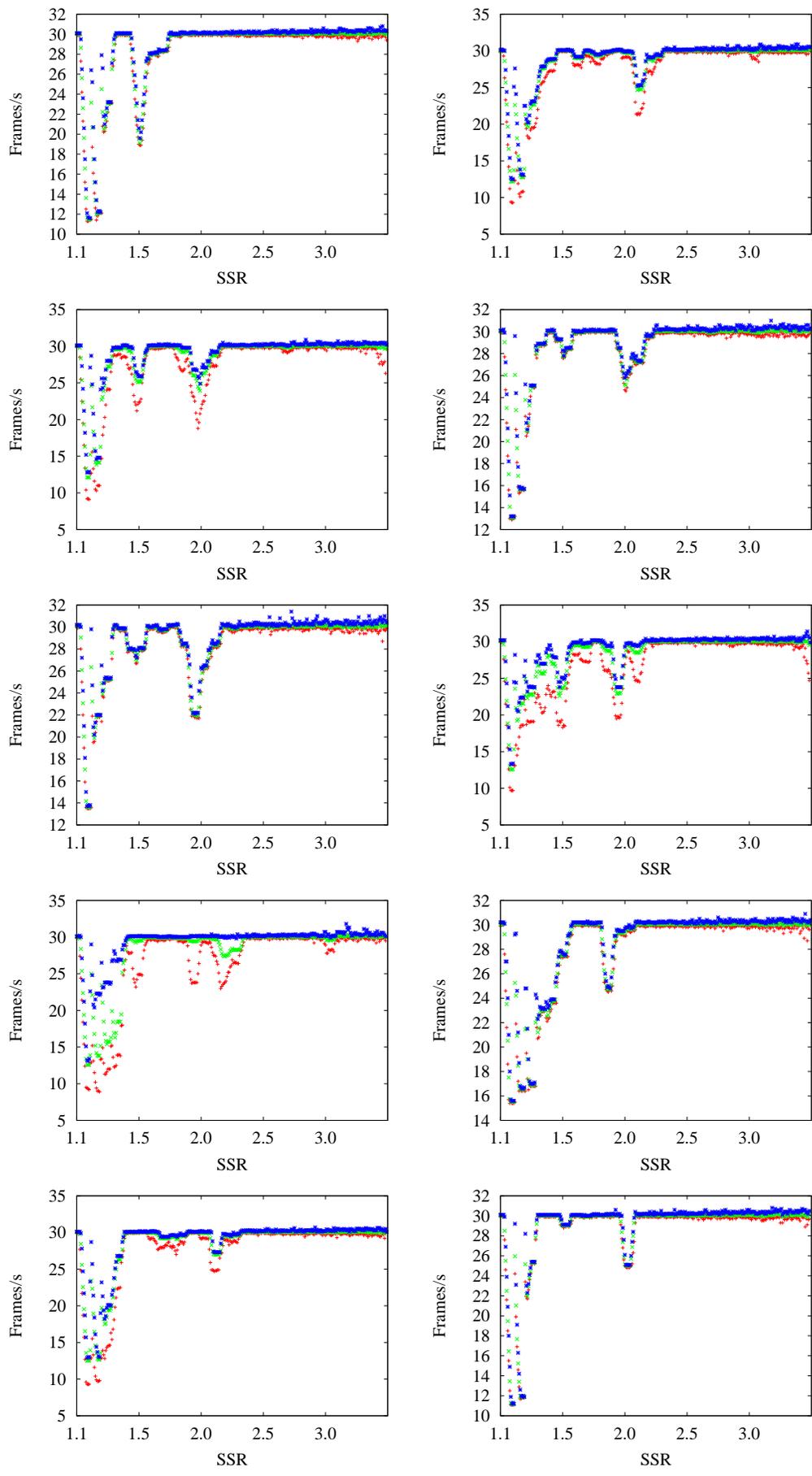


Figure B.9: Minimum, mean and maximum *frame rate* for ten executions of test run 3 at 800 Kbps with four neighbours and no upload limit. (Test Run: 3,  $SSR=1.1-3.0$ ,  $N=4$ ,  $UploadLimit=unlimited$ )

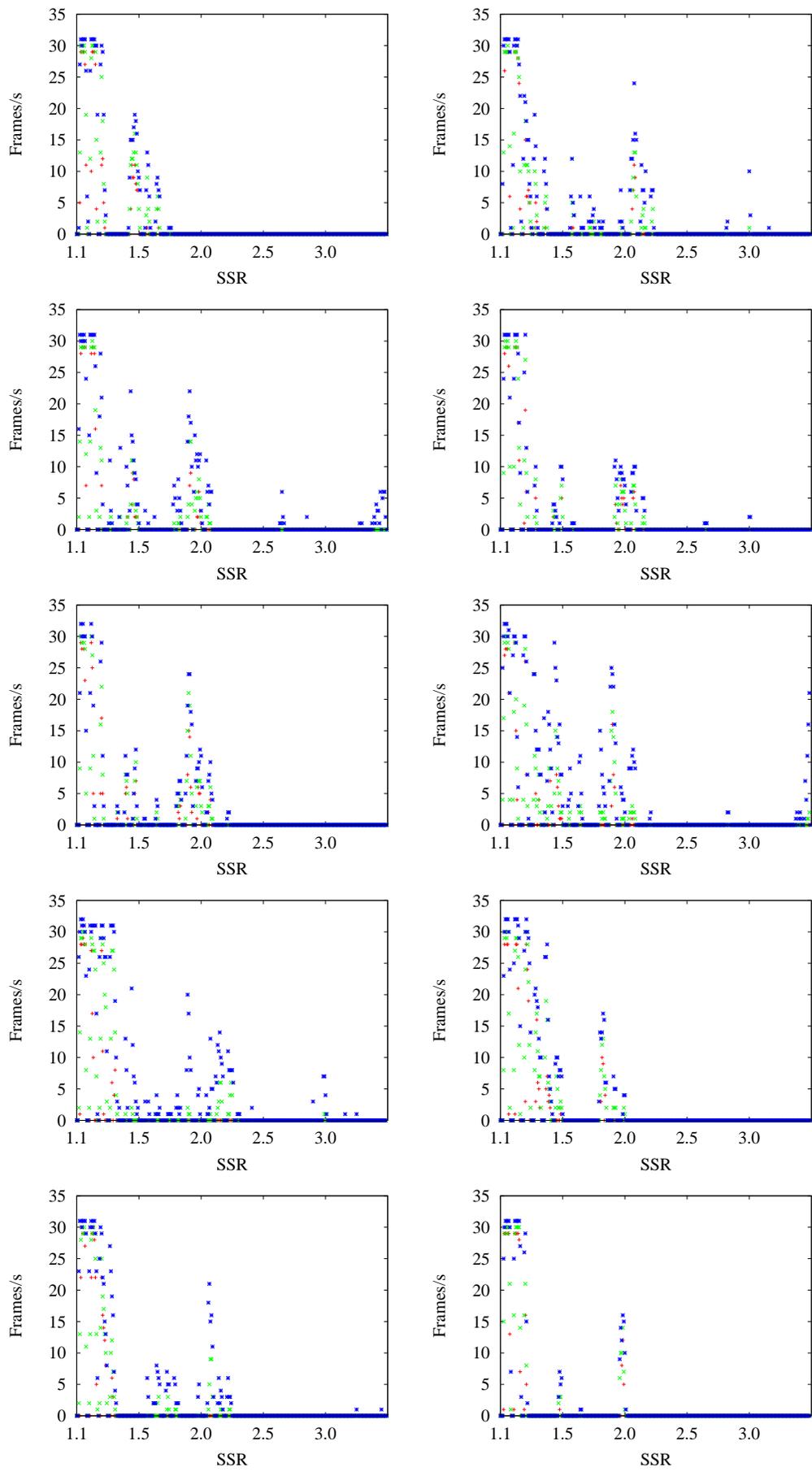


Figure B.10: Minimum, mean and maximum *frame loss* for ten executions of test run 3 at 800 Kbps with four neighbours and no upload limit. (Test Run: 3,  $SSR=1.1-3.0$ ,  $N=4$ ,  $UploadLimit=unlimited$ )

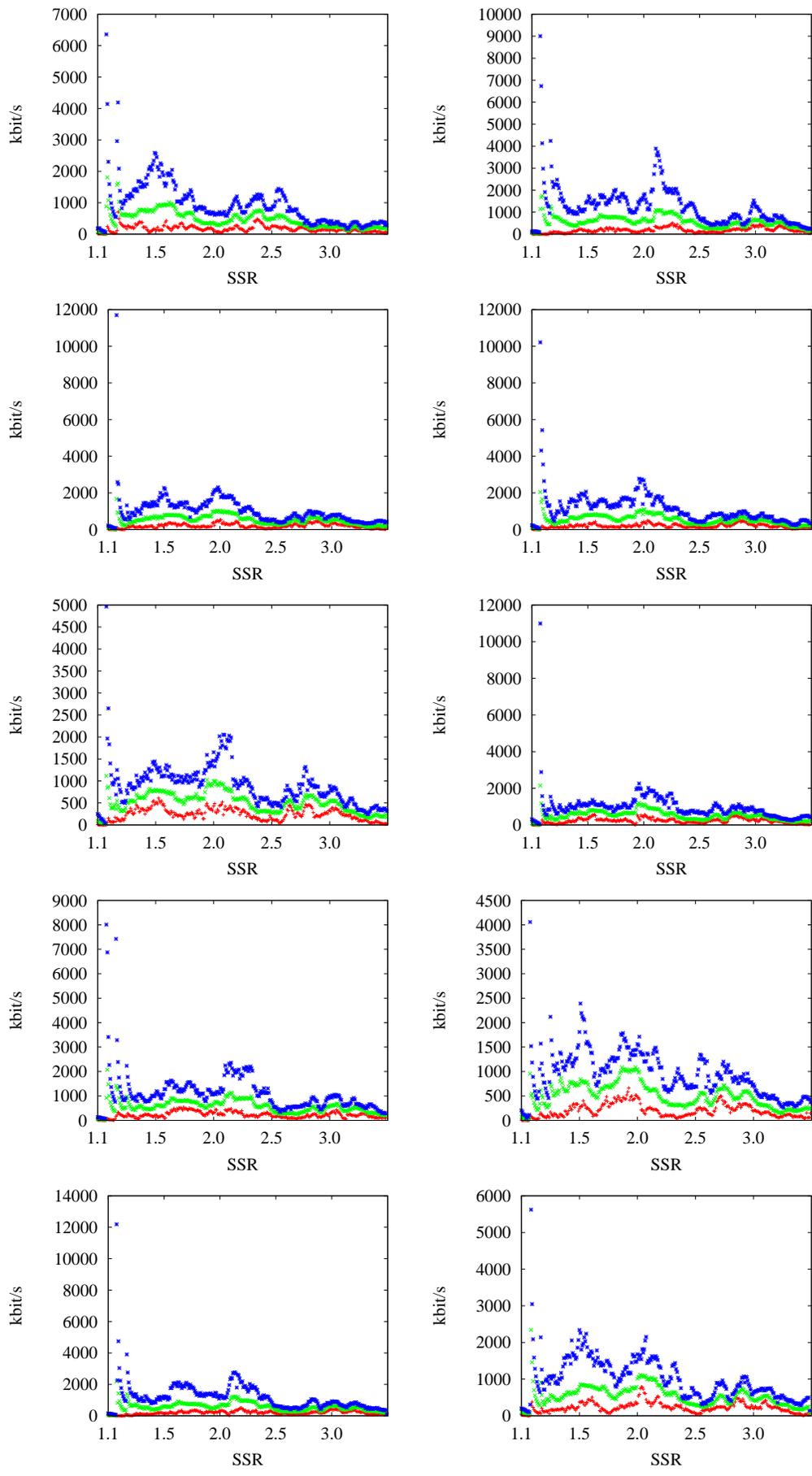


Figure B.11: Minimum, mean and maximum *upload rate* for ten executions of test run 3 at 800 Kbps with four neighbours and no upload limit. (Test Run: 3,  $SSR=1.1-3.0$ ,  $N=4$ ,  $UploadLimit=unlimited$ )

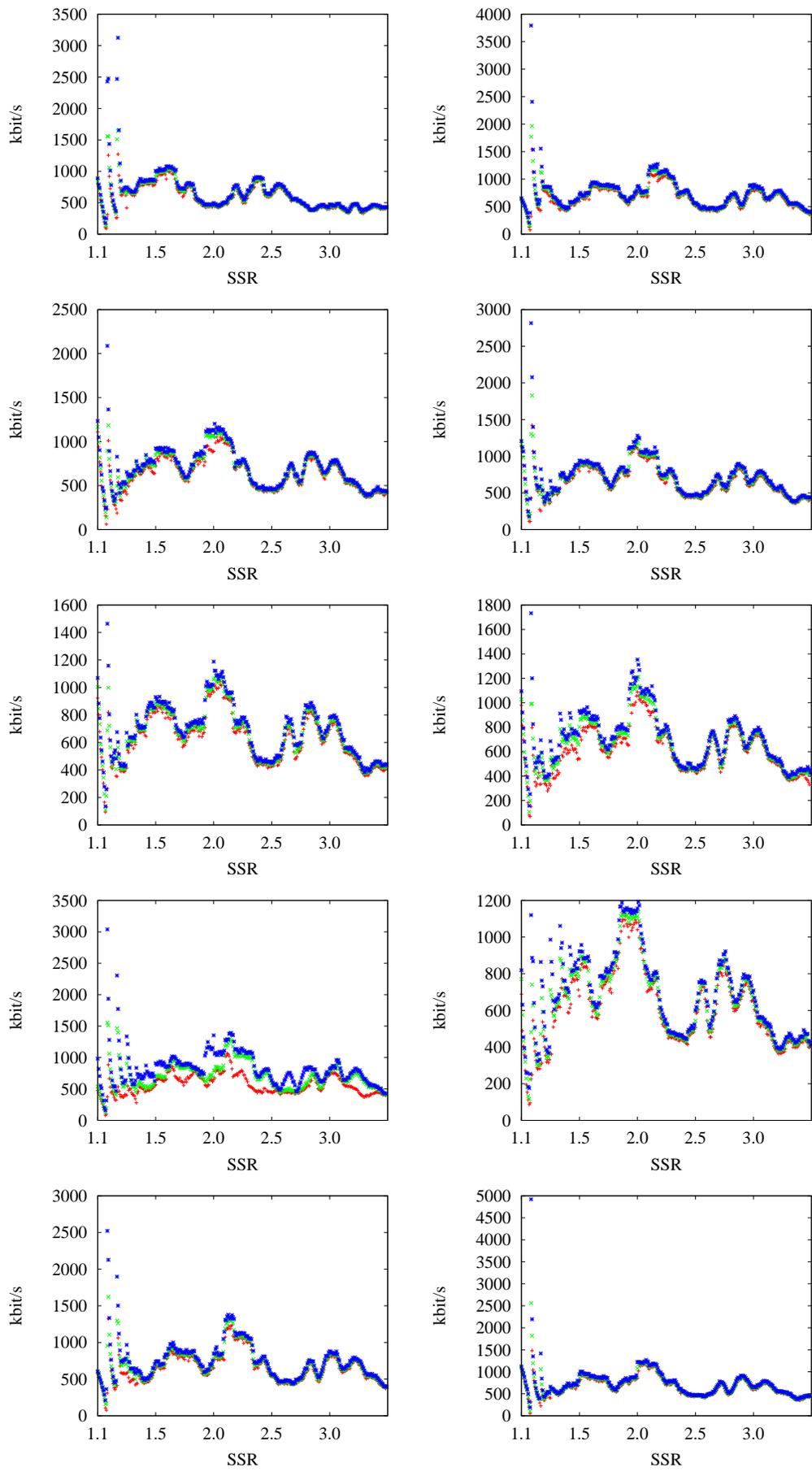


Figure B.12: Minimum, mean and maximum *download rate* for ten executions of test run 3 at 800 Kbps with four neighbours and no upload limit. (Test Run: 3,  $SSR=1.1-3.0$ ,  $N=4$ ,  $UploadLimit=unlimited$ )

## **B.2 Lumberjack Limited Client Upload Results**

This section contains the twelve (three test runs multiplied by four performance metrics) figures belonging to the three test runs described by Table B.1. The upload rate of the client nodes is limited to 1 Mbit/s in these tests, which is a representative upload limit for home users in the Netherlands.

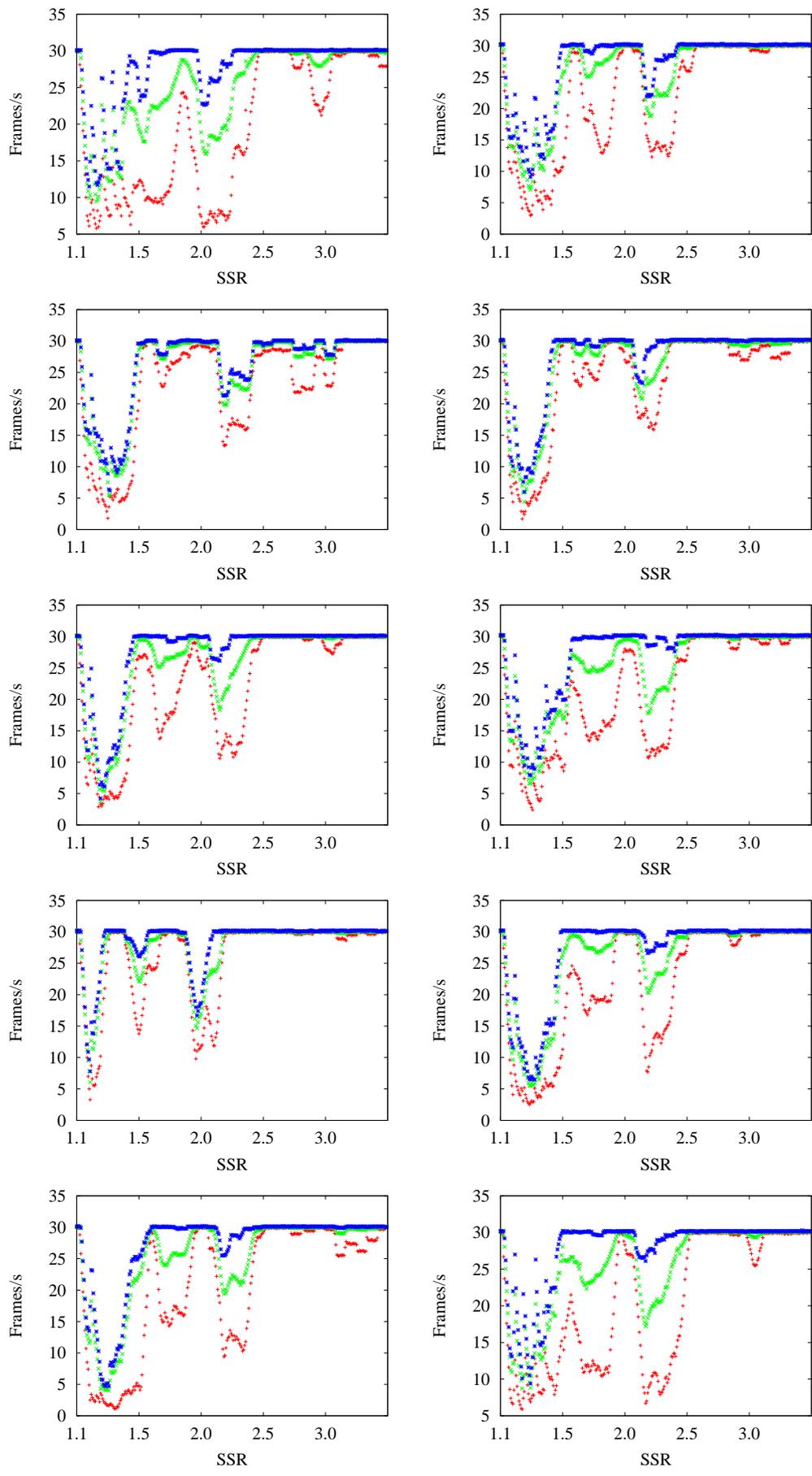


Figure B.13: Minimum, mean and maximum *frame rate* for ten executions of test run 1 at 800 Kbps with two neighbours and an upload limit of 1 Mbit/s. (Test Run: 1,  $SSR=1.1-3.0$ ,  $N=2$ ,  $UploadLimit=1$  Mbit/s)

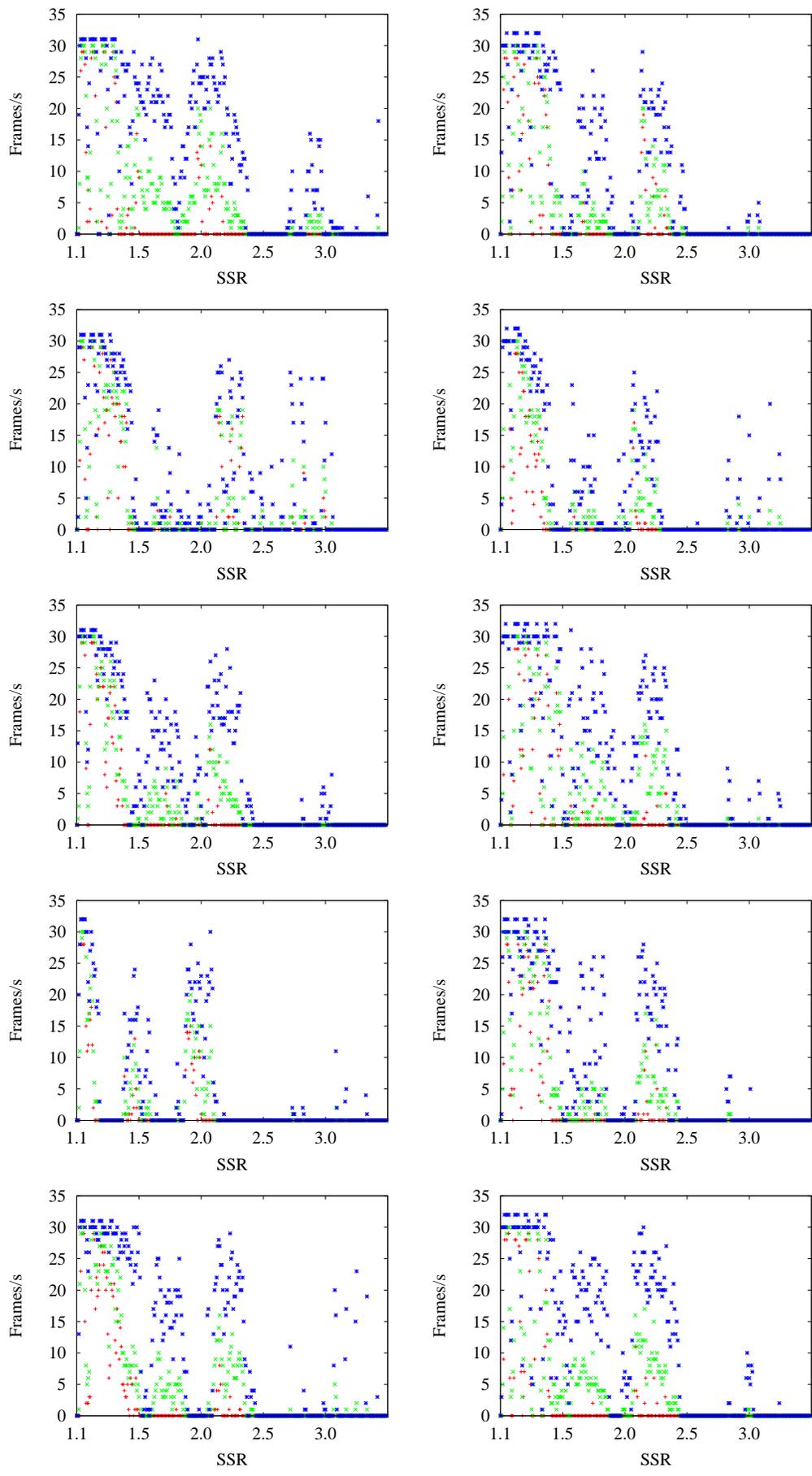


Figure B.14: Minimum, mean and maximum *frame loss* for ten executions of test run 1 at 800 Kbps with two neighbours and an upload limit of 1 Mbit/s. (Test Run: 1,  $SSR=1.1-3.0$ ,  $N=2$ ,  $UploadLimit=1$  Mbit/s)

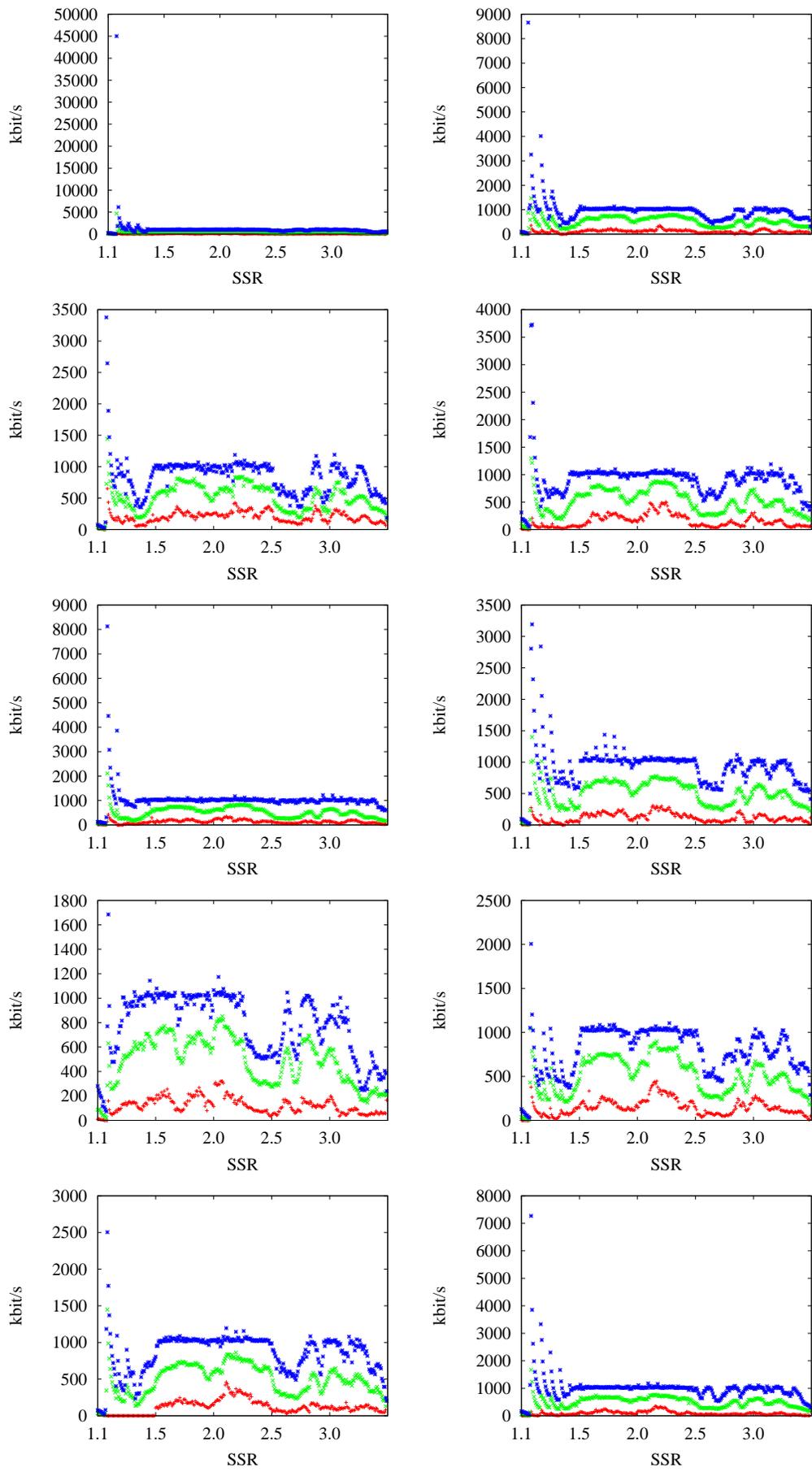


Figure B.15: Minimum, mean and maximum *upload rate* for ten executions of test run 1 at 800 Kbps with two neighbours and an upload limit of 1 Mbit/s. (Test Run: 1,  $SSR=1.1-3.0$ ,  $N=2$ ,  $UploadLimit=1$  Mbit/s)

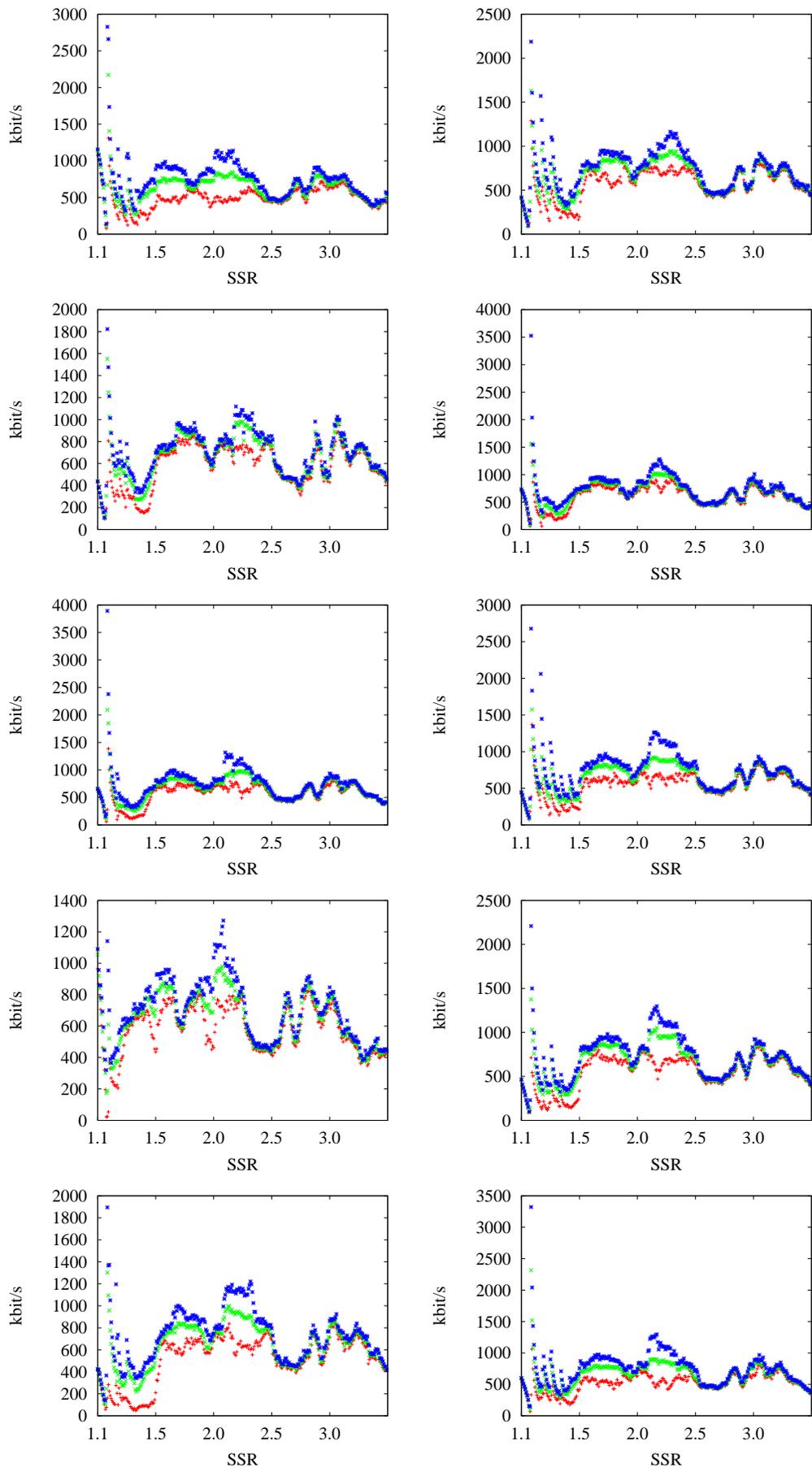


Figure B.16: Minimum, mean and maximum *download rate* for ten executions of test run 1 at 800 Kbps with two neighbours and an upload limit of 1 Mbit/s. (Test Run: 1,  $SSR=1.1-3.0$ ,  $N=2$ ,  $UploadLimit=1$  Mbit/s)

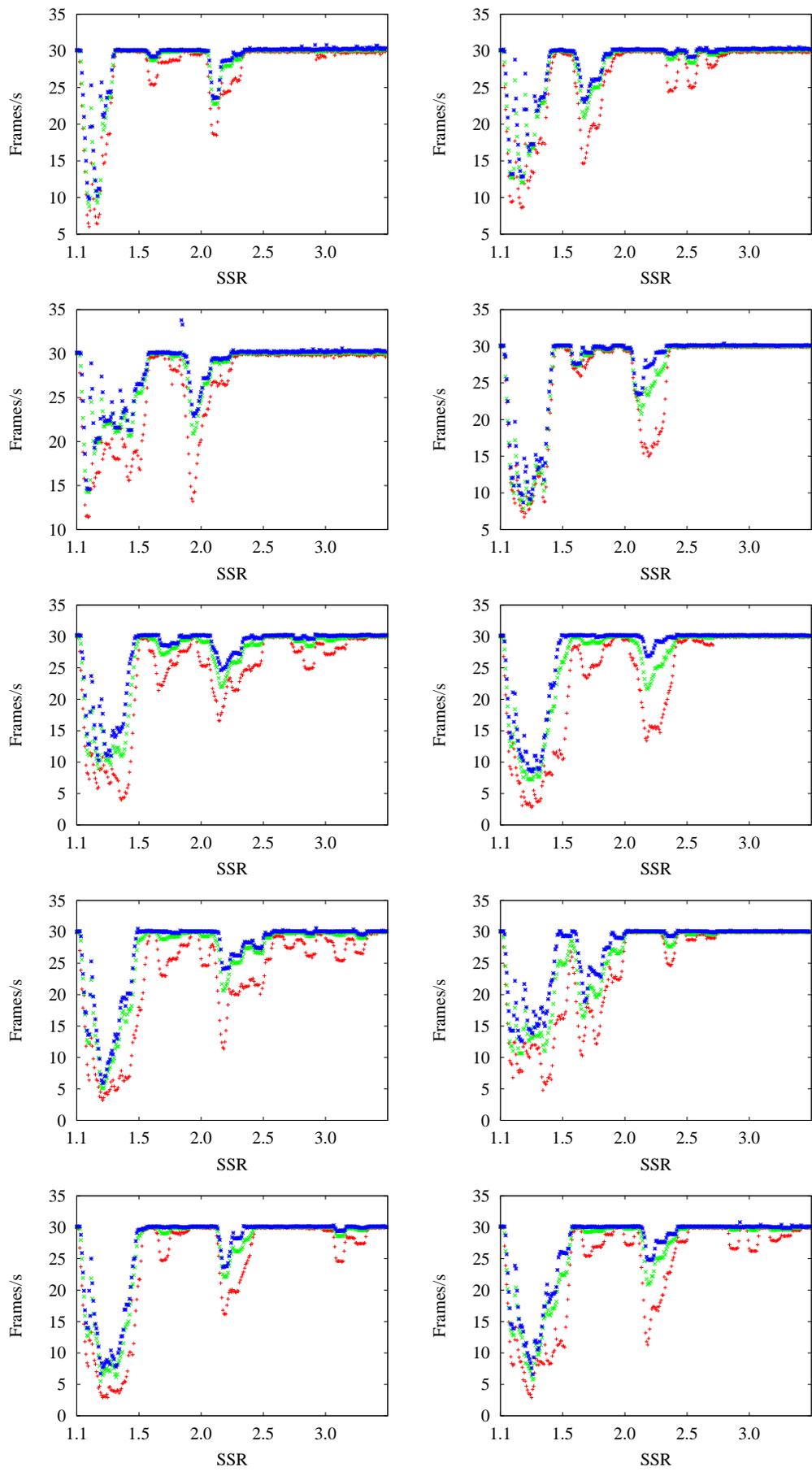


Figure B.17: Minimum, mean and maximum *frame rate* for ten executions of test run 2 at 800 Kbps with three neighbours and an upload limit of 1 Mbit/s. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=1$  Mbit/s)

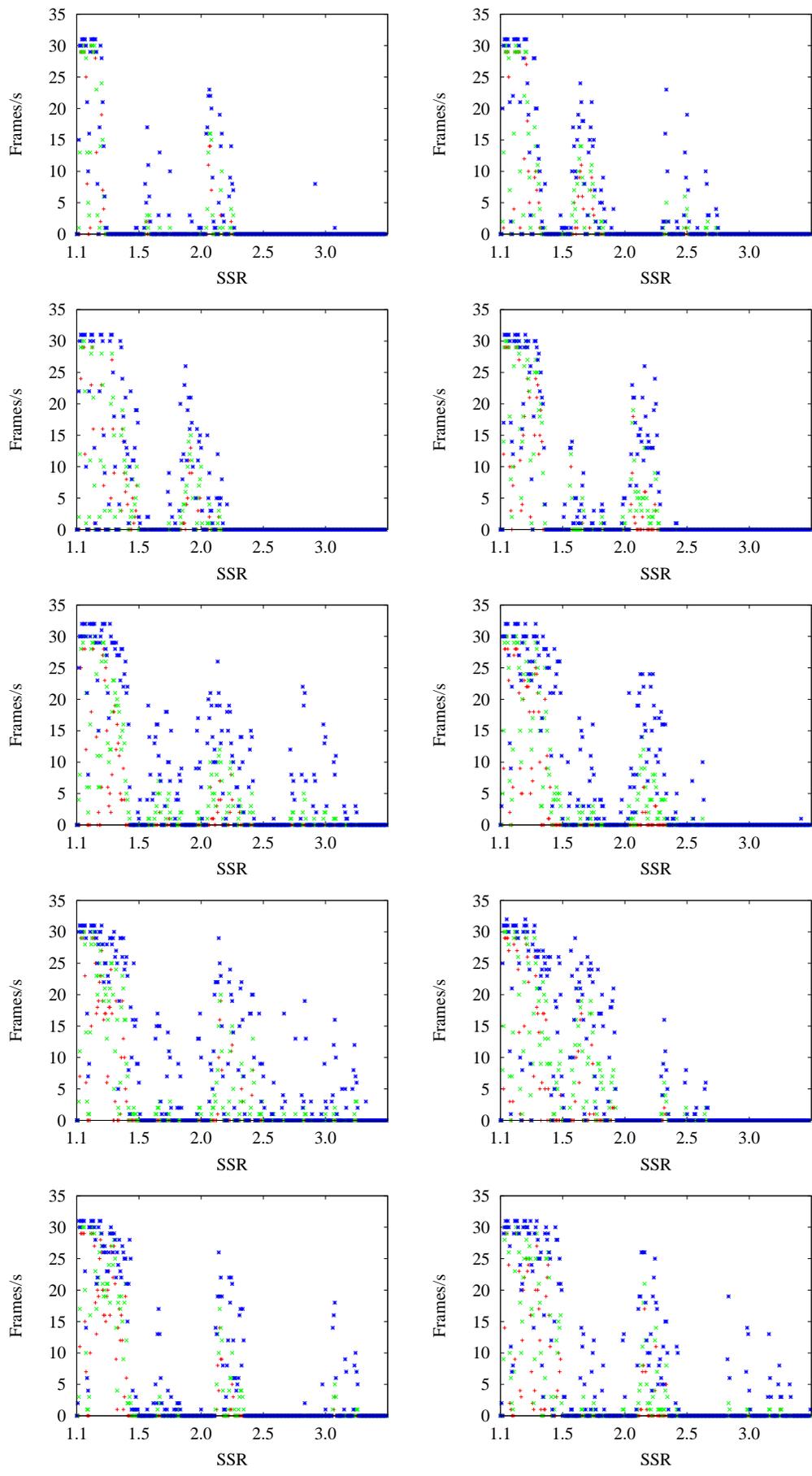


Figure B.18: Minimum, mean and maximum *frame loss* for ten executions of test run 2 at 800 Kbps with three neighbours and an upload limit of 1 Mbit/s. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=1$  Mbit/s)

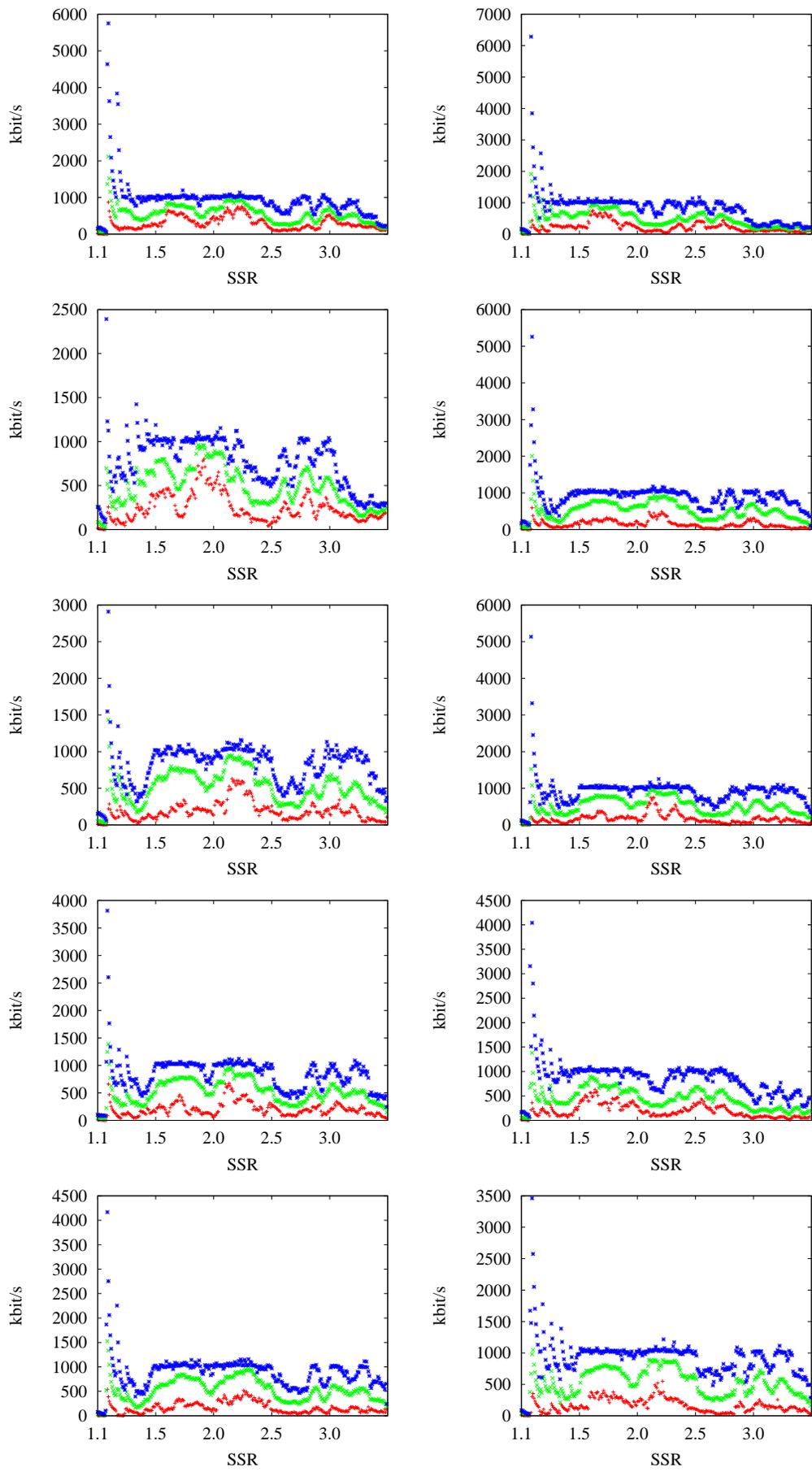


Figure B.19: Minimum, mean and maximum *upload rate* for ten executions of test run 2 at 800 Kbps with three neighbours and an upload limit of 1 Mbit/s. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=1$  Mbit/s)

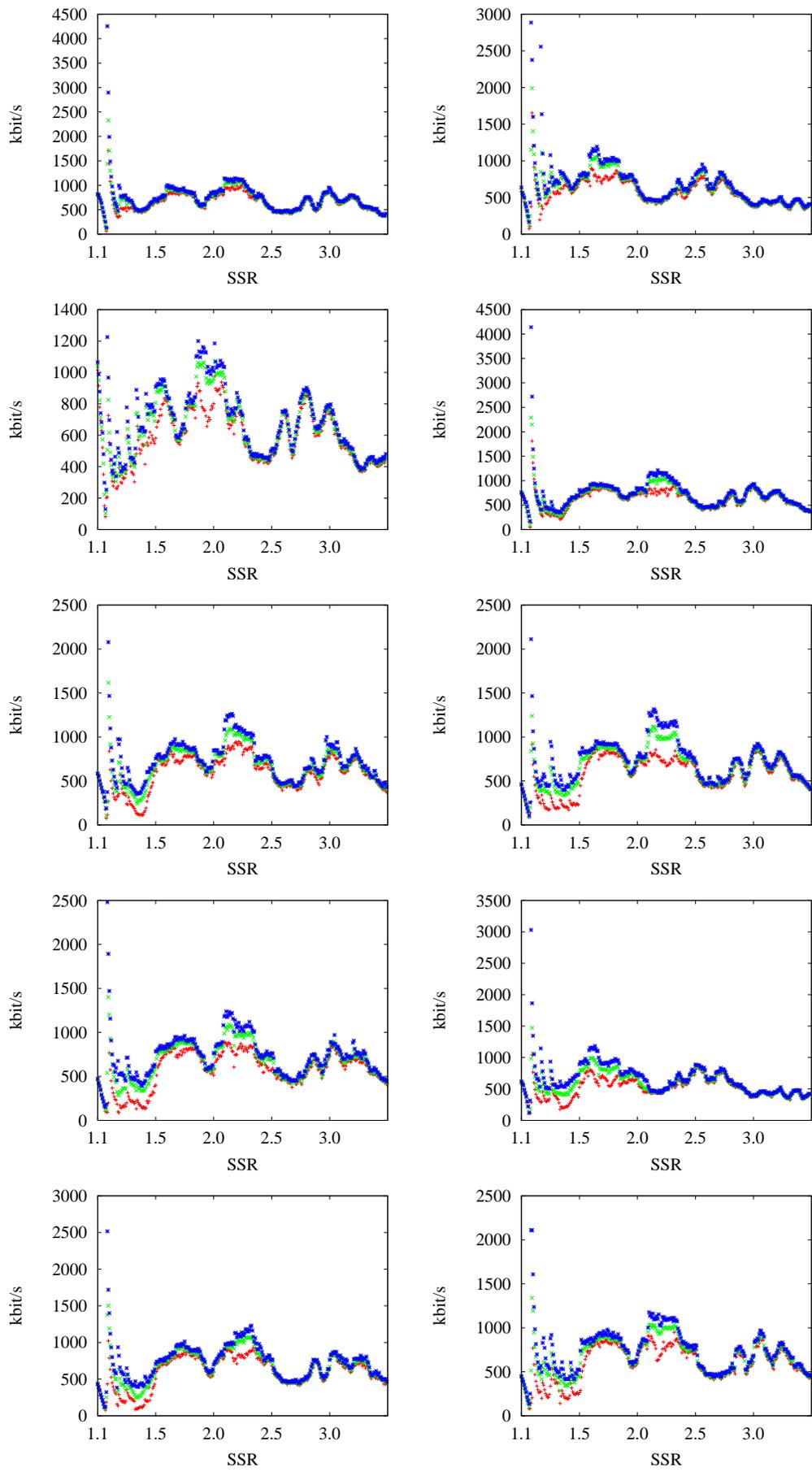


Figure B.20: Minimum, mean and maximum *download rate* for ten executions of test run 2 at 800 Kbps with three neighbours and an upload limit of 1 Mbit/s. (Test Run: 2,  $SSR=1.1-3.0$ ,  $N=3$ ,  $UploadLimit=1$  Mbit/s)

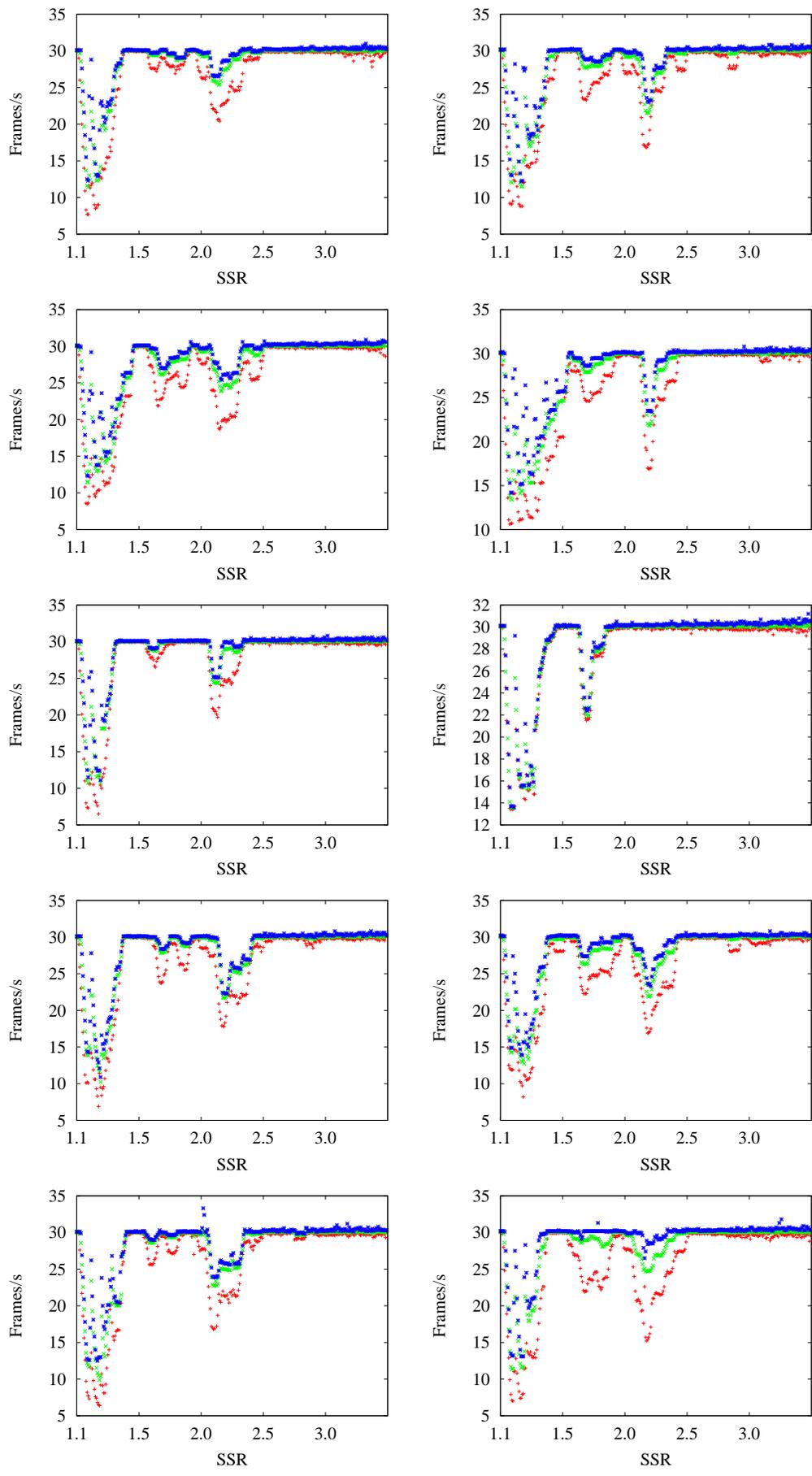


Figure B.21: Minimum, mean and maximum *frame rate* for ten executions of test run 3 at 800 Kbps with four neighbours and an upload limit of 1 Mbit/s. (Test Run: 3,  $SSR=1.1-3.0$ ,  $N=4$ ,  $UploadLimit=1$  Mbit/s)

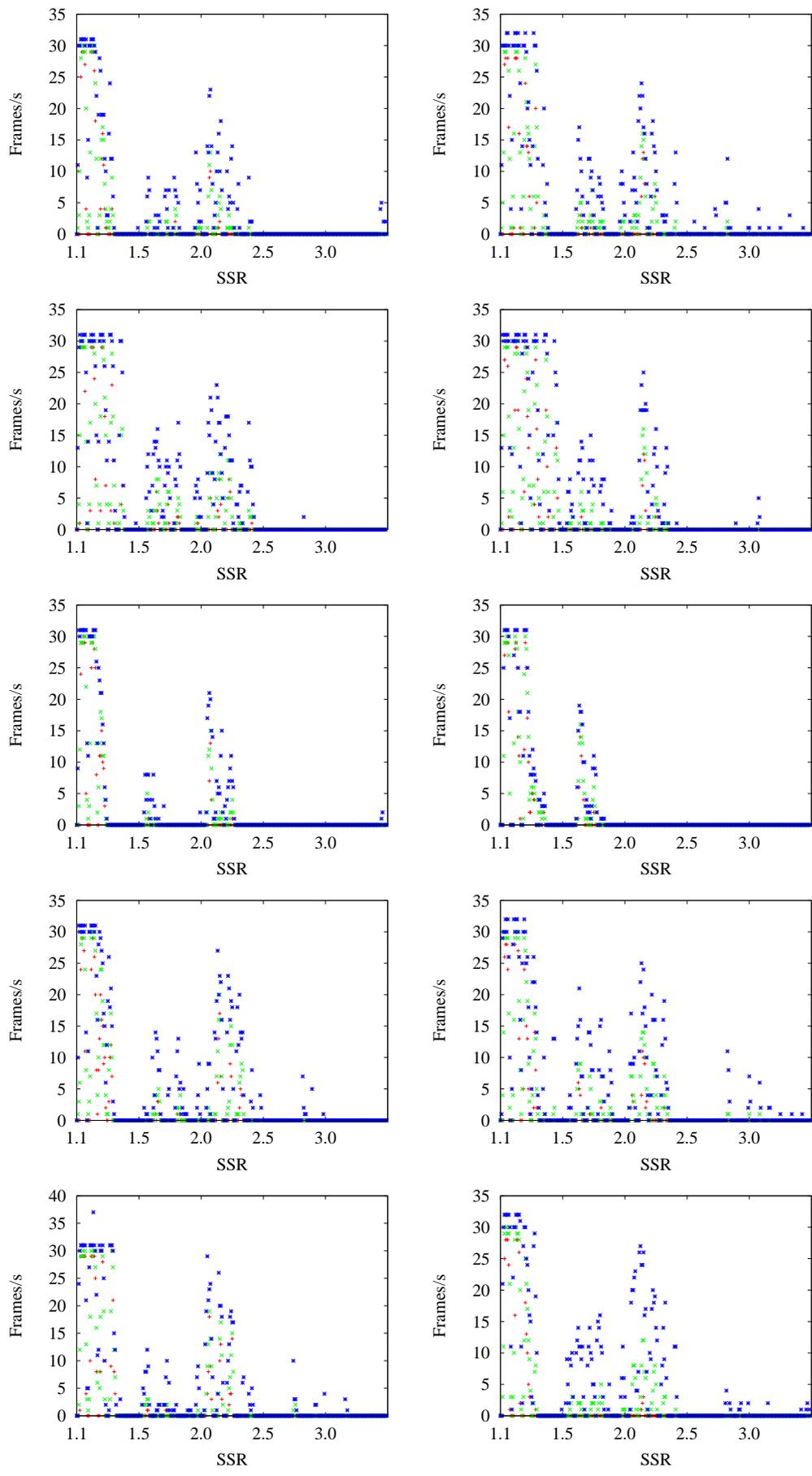


Figure B.22: Minimum, mean and maximum *frame loss* for ten executions of test run 3 at 800 Kbps with four neighbours and an upload limit of 1 Mbit/s. (Test Run: 3,  $SSR=1.1-3.0$ ,  $N=4$ ,  $UploadLimit=1$  Mbit/s)

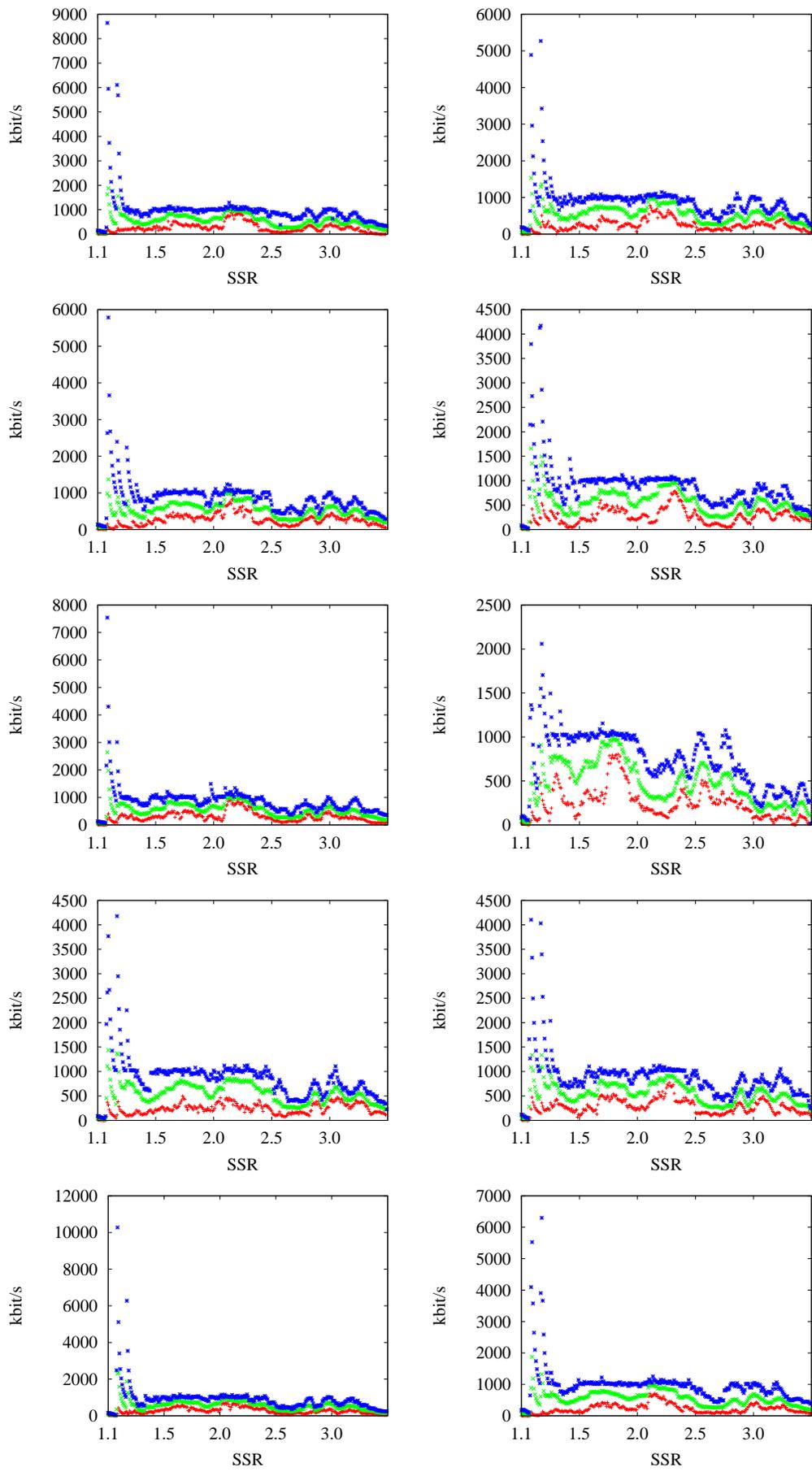


Figure B.23: Minimum, mean and maximum *upload rate* for ten executions of test run 3 at 800 Kbps with four neighbours and an upload limit of 1 Mbit/s. (Test Run: 3,  $SSR=1.1-3.0$ ,  $N=4$ ,  $UploadLimit=1$  Mbit/s)

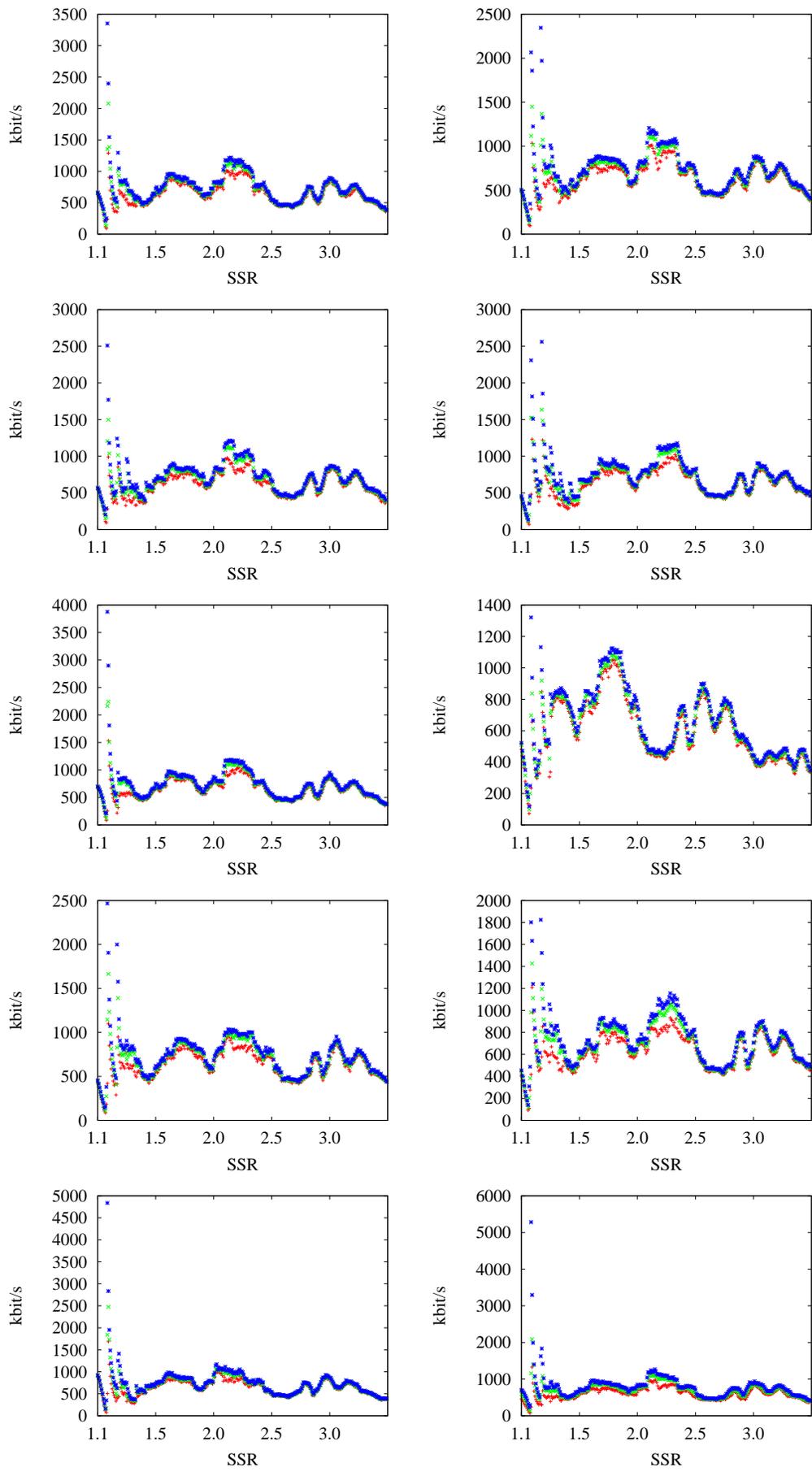


Figure B.24: Minimum, mean and maximum *download rate* for ten executions of test run 3 at 800 Kbps with four neighbours and an upload limit of 1 Mbit/s. (Test Run: 3,  $SSR=1.1-3.0$ ,  $N=4$ ,  $UploadLimit=1$  Mbit/s)