TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

The Performance of
Wireless MPEG-2 Video Streaming
by
Jan Ouwens

Date of defense	19th of August 2005
Supervisors	dr. P.D.V. van der Stok, Philips dr.ir. I. Radovanović, TU/e

Abstract

To determine the behaviour of MPEG-2 video streams over wireless IEEE 802.11b networks, and to see whether using TCP is a viable alternative to using UDP as transport protocol for such streams, we measured the performance of streams over wireless networks using both UDP and TCP, varying several parameters pertaining to the state of the network and the stream. We present how we have implemented IFD, an algorithm designed to optimize the quality of an MPEG-2 video stream and previously only available on top of UDP, to be used on top of TCP, and we have incorporated it into our measurements. We found that, in the described network, TCP's performance is indeed better than UDP's, and that the retry value and the transmission rate of the senders wireless card, and the packet size of individual packets have a profound influence on the performance, regardless of the transport protocol used.

Acknowledgments

I would like to thank the following people, without whom things would have been very different: Peter and Igor, for giving answers and asking questions; Sergei, Jeffrey, and everybody else at the OASIS cluster, for their help and support in technical matters; my parents, for their help and support in non-technical matters; Cornee, for lunch and coffee breaks; Marc, Alina, Martijn, Richard, and Remi, for occasionally breaking the silence; Maarten, Nick and Ralf, for providing me with wallpaper music; Francisco and Paul, for Chinese whispers; and the two cups that gave their lives inside the microwave in the pursuit of knowledge.

Also, I would like to thank my grandmother, who taught me the BASICs. Without her, I wouldn't be where I am today.

Contents

0 Introduction

This Master's thesis discusses the research that I have done as a member of the OASIS cluster within Philips Research Eindhoven, in order to attain my Master's degree in Computer Science at Eindhoven University of Technology (TU/e).

The OASIS cluster within Philips Research, Eindhoven, led by Peter van der Stok, is active in the field of streaming media, and, at the time of my involvement, specifically streaming video over wireless networks. To this end, they used the UDP and RTP protocols, which are specifically designed for real-time media streaming. However, they did not have much experience with TCP as transport protocol. Since other parties were using TCP, and also HTTP, in the field of streaming media with success, they decided to do more research into this as well. The task of doing this research was given to me. As my Master's project, this would give the OASIS cluster more insight into the advantages and disadvantages of TCP, and it would provide me with the opportunity to work in a corporate setting and learn about networking on both a theoretical and a practical level.

The requirements for my project were to find out what can be done to use TCP and HTTP for video streaming (see Section 1.6), and find out how the IFD algorithm can be used to work with TCP, without altering TCP itself (see Chapter 2). Finally, an analysis of the behaviour of streaming data, MPEG-2 video specifically, over wireless links, using both UDP and TCP, was desired (see Chapter 3).

Earlier work by Xylomenos and Polyzos [20] handles the performance of both UDP and TCP in the context of the wireless LAN, but does not consider the specifics of MPEG-2 streaming. Work performed within Philips Research, which does apply to media streaming, indicates that TCP is, mildly put, impractical for video streaming [18, 9]. However, some attempts have been made to modify TCP in such a way that it becomes more 'streaming-friendly' [18]. Moreover, Sergei Kozlov has developed an algorithm to increase the quality of MPEG-2 streams over UDP [10]. In this thesis, we present a way to use this algorithm so that it can be used on TCP, without modifications to TCP itself. The measurements in this thesis will specifically focus on the influence of MPEG-2 video and wireless LANs on the behaviour of streams using UDP and TCP as transport protocol.

1 Context

1.1 Background

Streaming of multimedia is popular on the internet. A lot of streaming applications can be found. Examples include internet radio and streaming of tv shows from www.uitzendinggemist.nl. The challenge is to get the best possible audio/video quality within the network's limitations. The available bandwidth and the amount of data loss are important factors in determining these limitations. To complicate matters further, neither bandwidth nor the amount of data loss are necessarily constants. Other traffic on the network may cause fluctuations in the amount of bandwidth that is available for the stream. External interference may cause an increase in data loss on wireless networks.

There are several different ways to improve the quality of a multimedia stream. On the one hand, the bandwidth and reliability of the network can be improved. On the other hand, better ways to compress the data can be devised. A third area of research to this end, and subject of this thesis, is the protocol sets that are used to stream the data.

It is possible to use streaming protocols that are oblivious to the type of data they stream. However, better results may be achieved by exploiting the characteristics of the data. In this thesis, we are interested in streaming MPEG-2 video. We will therefore look at the properties of MPEG-2 video. Note however that there may be a price to pay. Using this approach limits us to the use of MPEG-2 video only; if we want to stream another type of video, in the best case, nothing will happen. In the worst case, the protocol may misinterpret the data, which will lead to unpredictable results.

The intended environment is the home. A home network may be based on the Internet Protocol Stack, on Bluetooth, or something else. It consists of a number of devices, such as digital televisions, DVD players and PCs. Within this home network, video may be streamed from any sender to any receiver. Typical senders of video streams are DVD player and PCs. Typical receivers are TVs and PCs.

1.2 Streaming

Streaming is a special case of downloading. In *downloading*, we want to transfer data from a remote location to a local device. It is not important how this data is transferred; the important thing is that at some point in time, the data on the local device is exactly the same as the data on the remote device. When the data has been downloaded, we can start using it.

In *streaming*, we want to use the data while it is still in transit from the remote location to the local device, in real-time. The advantage of this, is that we don't have to save all the data locally; we can throw away the data we have already used. In order to be able to do so, it is necessary for the parts of which the data consists, to be transmitted in the exact same sequence in which they appear in the data. In the case of video, it would be impractical if

frame # 86 arrived long before, or long after frame # 85. This implies that the data must have a linear character: it must be possible for the receiver to process a chunk of data in a stream without having prior knowledge of the data that follows. Data that is not linear cannot be streamed. In Microsoft's AVI video format, for example, it is possible to store all the video followed by all the audio of a movie. This encoding requires random file access in order to be displayed, making it unsuitable for streaming.

We also identify a special case of streaming: *live streaming*. In live streaming, in addition to the data arriving in sequence, we want it to arrive in real-time as well. If a movie has a framerate of 25 frames per second, we want 25 frames to arrive every second, preferably in intervals of 40 ms. This is necessary to properly display, for example, a soccer match. It is impossible to temporarily suspend broadcast of the match, just because the connection to one of the receivers is bad and it needs time to catch up.

1.3 Networking

To stream video, one can devise a private protocol, or one can use a protocol that already exists. In this thesis, we will follow the earlier work of the OASIS cluster and focus on the protocols of the Internet Protocol Stack (more commonly known as the TCP/IP Stack, after its two most important protocols [12]). The protocols in the Internet Protocol Stack are divided into four layers: the network interface layer, the internetwork layer, the transport layer and the application layer.

1.3.1 Network interface and internetwork layers

The protocols in the network interface layer, also called the link layer, provide an interface to the network hardware. We will look only at the IEEE 802.11 network interface layer protocol [6], and more specifically, the IEEE 802.11b variant. This protocol is used for wireless LANs. To ensure compatibility with wired LANs, IEEE 802.11 has the same interface as the Ethernet protocol, which is one of the most popular wired internetwork layer protocols.

The internetwork layer protocols, of which the Internet Protocol (IP) [12] is the most widely used, provide an interface to the network interface layer, eliminating the need for higher level protocols to worry about the specifics of the network hardware.

IEEE 802.11

The IEEE 802.11 protocol allows two devices to send data packets to each other without these two devices being physically connected through a wire, using radiowaves instead. While it is possible to connect two wireless devices directly, usually an access point is used as a go-between. An access point is a device that extends the range in which devices can be connected to each other and that can provide security on a wireless network. It can also serve as a bridge between wireless and wired LANs.

The IEEE 802.11 protocol can transmit packets at a rate of either 1 Mbit/s or 2 Mbit/s. In this thesis, we will look at an extension of this protocol, called IEEE 802.11b [7], which can also transmit packets at 5.5 and 11 Mbit/s. There is another popular extension, IEEE 802.11g, which can transmit at up to 54 Mbit/s, which we do not use.

A wireless connection is not 100 % reliable. Without a special protocol to take care of this, a data packet that is transmitted is not guaranteed to arrive at the intended receiver. This

can have several causes. For instance, the signal may be too faint to be received properly because the distance between the two devices is too large. To handle this, it is possible, up to a limit, to increase the strength of the transmission, or the sensitivity of the receiver. Also, an access point can be put between them, to relay the transmission. Another problem is that the signal can be corrupted by outside interference. This interference can be caused by other wireless devices, or even by an active microwave. To handle this, the protocol employs two techniques: first, it tries to avoid collision with interfering transmitters. Secondly, using an acknowledgment scheme, it retransmits corrupted or lost packets.

The protocol roughly works as follows. A device that wants to transmit a data packet waits until no other devices are transmitting data. It then waits a small, random additional amount of time to avoid collision with other devices that are also waiting for the air to be clear of other signals. If, by that time, the air is not clear (i.e., another device has begun transmitting), it starts waiting anew. If the air is clear, it transmits the data packet and starts waiting for an acknowledgment from the receiver. If the acknowledgment does not arrive within a certain amount of time, it re-sends the data. This repeats until the acknowledgment is received and arrival of the packet has been confirmed, or until the packet has been sent a certain, maximal number of times, at which point the packet is declared lost. We will refer to this maximal number of times that one packet can be transmitted as the *retry value*. This retry value can be set by the user. The default value depends on the manufacturer of the device, but is usually either 8 or 16. Each (re)transmission costs time, so if the retry value is too high, this may have an adverse effect on the timing of the stream.

Another mechanism that a wireless device can use to increase reliability, is to decrease the *transmission rate*. By default, for IEEE 802.11b, packets are sent at 11 Mbit/s. However, when the device detects bad link conditions (for instance, due to interference), it will transmit packets at one of the lower rates (1, 2 or 5.5 Mbit/s). The device can decide per packet at which rate to transmit. The criteria for this, however, are not specified by IEEE and can be determined by the manufacturer of the device. For some devices, a maximum transmission rate can be set, although not for all.

IP

The Internet Protocol (IP) assigns to each host in the network a unique number, the *IP address*. IP uses the IP addresses to identify these hosts.

The unit of transmission of IP is the IP datagram, which contains a header and the data that is sent by the higher level protocol. The maximum length of an IP datagram is 65535 bytes, including the header. If the underlying network does not support datagrams of this size, IP will handle the fragmentation of the datagram. IP takes a datagram and moves it from host to host, until the datagram reaches its destination as defined by the IP address. IP does not provide reliable transport: higher-level protocols are responsible for reliability, if they need it.

IP supports multicasting. A certain number of IP addresses is reserved for multicasting, and each of these addresses represents a group of zero or more hosts. Datagrams that are sent to a multicast address are forwarded to the members of the group. In streaming, this is useful if several hosts want to receive the same stream.

1.3.2 Transport layer

The transport layer protocols are responsible for end-to-end transport of data; they have no knowledge of the network's topology and rely on IP for delivery. They take an amount of data, which we call a packet if we use UDP or a segment if we use TCP, and transmit it.

The most important of the transmission layer protocols are the User Datagram Protocol (UDP) [13] and the Transmission Control Protocol (TCP) [14]. Both UDP and TCP allow multiple streams of data to be sent simultaneously; they use *ports* to distinguish between the streams. Depending on the number that identifies a port, the data is sent to the corresponding application layer protocol for further processing.

UDP

The User Datagram Protocol (UDP) is a simple protocol. It simply sends a certain amount of data from a source to a destination. It neither gives any guarantees about the order in which packets arrive, nor about whether they arrive at all. Also, UDP does not take care of splitting and reassembling packets if they are too large to fit in an IP datagram. On the other hand, UDP involves very little overhead, with respect to both time and space. UDP is therefore most useful for applications that need to send large quantities of data quickly and that can recover from data loss; in other words, that value speed over reliability.

TCP

The Transmission Control Protocol (TCP) is the most widely used transport layer protocol. It differs from UDP in the fact that it guarantees arrival, in order, of all data it transmits. This comes at the cost of more overhead and lower transmission speed.

TCP divides the bytes in the send buffer into numbered segments of equal size. It uses a sliding window to transmit these segments. This means that a window is defined over the segments. TCP transmits all segments that lie inside the window. When the receiver has received a segment intact, it sends an acknowledgment for this segment back to the sender. When the sender has received acknowledgments for the first n segments in the window, the window slides n places to the right, allowing the next n segments in the send buffer to be transmitted. If the sender does not receive an acknowledgment for a segment within a certain time, it retransmits the segment until it does receive an acknowledgment. This way it knows for certain that the segment has been received. The receiver then rearranges the segments into their original order, and eliminates any duplicates.

TCP is therefore a reliable protocol. This reliability comes at the cost of higher bandwidth usage, due to datagram retransmissions and acknowledgement transmission. Also, TCP needs to establish a connection before being able to send data, which causes a delay of twice the latency of this connection, before the data can be sent. Further delays may occur if packet loss is common on the network, as the protocol will wait for packets to arrive at their destination before sliding the buffer window.

The specification of TCP also provides for a number of congestion control algorithms. These algorithms adapt the send rate to the load of the network: if more bandwidth is available, the send rate is increased, and if network congestion occurs, the send rate is decreased. This guarantees a near-optimal transmission speed for TCP packets. See [17] for more information on specific congestion control algorithms.

Finally, TCP is not particularly suited to multicast, as every receiver needs to acknowledge each packet. If one receiver does not acknowledge a packet, the buffer window may not slide and all receivers are forced to wait. Thus, TCP is most useful when reliability is more important than speed, and multicasting is not required.

1.3.3 Application layer

Upon the transport layer protocols, the application layer protocols are built. The application layer protocols that are relevant to the subject of this paper, are the Real-Time Transport Protocol (RTP) [16, 15] and the Hypertext Transfer Protocol (HTTP) [3].

RTP

The Real-Time Transport Protocol (RTP) is specially designed for multimedia streaming. Typically, it uses UDP to transfer data, although it could also be built upon other transport layer protocols. The RTP specification does not guarantee reliability, so the application that uses RTP must provide a reliability mechanism itself, or accept the possibility of packet loss. In multimedia streaming, it is usually not fatal if a packet (and therefore a frame) is lost.

The protocol provides a timestamp for every packet. This is used for synchronisation, for example of the audio and video parts of a stream. Each packet header also contains a payload type to identify the type of media the packet contains. Furthermore, there are header fields to identify the source of a stream and possible contributing sources. These are primarily meant for video conferencing and are beyond the scope of this paper.

Receivers of RTP packets can use the Real-Time Control Protocol (RTCP) to send quality feedback to the sender, including packet counts. The sender can use this information to improve the quality of the stream in a manner comparable to TCP's congestion control algorithms.

HTTP

The Hypertext Transfer Protocol (HTTP) is widely known as the protocol of the world wide web. It is connection-based: a client opens a connection with a server. It can then send requests, and the server responds to these requests. When the connection is closed, no more requests can be sent by the client, and no more responses can be sent by the server.

Usually, HTTP uses TCP to transport data, as packet loss could result in incomplete web pages. However, with some effort to overcome data loss, especially when establishing connections, it should be able to use UDP as well.

Headers are used to describe both the HTTP packet (e.g., whether it's a request or a response) and its contents, if present. They are supplied in plain-text inside an HTTP packet, in the form key : value, where headers may appear in any order. The HTTP specification defines many headers that may or may not be used. New headers may be defined and used as well (although then the protocol might not be recognised by other HTTP implementations, such as browsers or web servers, anymore). This approach does introduce some overhead when compared with protocols with fixed headers, such as RTP. Since HTTP's headers are in plain-text, human-readable form, they include the name of the header (for fixed headers, it is not necessary to include a header's name, since it is identified by its position within the header), and no effort is made to compress them. However, this does not need be problematic when the number of headers is small or the payload is large.

The specification of HTTP 1.1 provides two types of transfer encoding: a default one, which sends a payload in one large HTTP packet (which may consist of more than one TCP packet), and 'chunked', which splits the payload into a sequence of separate HTTP packets that are then sent consecutively. HTTP 1.1 requires that the length of the payload be included as a header field inside an HTTP packet. However, when the transfer encoding is set to chunked, only the length of a chunk needs to be included, not the length of the entire stream. This makes live streaming easier, since the length of a live stream is not known beforehand. Nevertheless, the Shoutcast internet radio protocol uses HTTP without using a chunked encoding.

1.4 MPEG

MPEG is one of the most widely used sets of standards for audio and video compression. The MPEG standards that are relevant in this context are called MPEG-1, MPEG-2 and MPEG-4. MPEG-1 is primarily used for Video CD and CD-ROM applications. MPEG-2 is an extension of MPEG-1 to allow for better image quality (at the price of larger bandwidth). DVD movies are encoded in MPEG-2. MPEG-4 is an extension in the other direction: it is optimised for streaming over low bandwidths. As this thesis focuses on MPEG-2, we will use the terms MPEG and MPEG-2 interchangeably.

1.4.1 MPEG video format

The video part of an MPEG file is divided into *Groups Of Pictures (GOPs)*. Each GOP contains a number of picture frames. There are three types of picture frames: intra (I), predicted (P) and bi-directional (B). An *I-frame* is coded as a stand-alone picture frame. A *P-frame* is encoded relative to the previous I- or P-frame. It depends on this frame for both encoding and decoding, but it is smaller in size than an I-frame. A *B-frame* is encoded dependent on the previous I- or P-frame and the next I- or P-frame, so that they are smaller than P-frames. A typical GOP may look as follows [18]:

$$\mathtt{I}_1\ \mathtt{B}_2\ \mathtt{B}_3\ \mathtt{P}_4\ \mathtt{B}_5\ \mathtt{B}_6\ \mathtt{P}_7\ \mathtt{B}_8\ \mathtt{B}_9\ \mathtt{P}_{10}\ \mathtt{B}_{11}\ \mathtt{B}_{12}\ \mathtt{I}_{13}$$

Where $\mathtt{I}_{13}$ belongs to the next GOP. Since a B-frame needs the next P-frame in order to be decoded, this means that one I- or P-frame and two B-frames need to be buffered before the other P-frame arrives which is needed to decode the B-frames. Therefore, GOPs are usually transmitted in a different order [18]:

$$\mathtt{I}_1\ \mathtt{P}_4\ \mathtt{B}_2\ \mathtt{B}_3\ \mathtt{P}_7\ \mathtt{B}_5\ \mathtt{B}_6\ \mathtt{P}_{10}\ \mathtt{B}_8\ \mathtt{B}_9\ \mathtt{I}_{13}\ \mathtt{B}_{11}\ \mathtt{B}_{12}$$

This way, the decoder only needs to buffer two frames. Note that, in this order, $\mathtt{I}_{13}$, $\mathtt{B}_{11}$ and $\mathtt{B}_{12}$ all belong to the next GOP. In the remainder of this thesis, they will be treated as such.

If for any reason bandwidth drops such that it is no longer possible to transmit all frames in time for decoding, and a scheduler that is aware of the MPEG encoding scheme is present, this information can be used to make a decision on what packets should be dropped. Dropping a B-frame merely results in one less frame being displayed, freezing the video for a brief moment. Dropping a P-frame, however, can lead to distorted picture, as all succeeding frames in the GOP (in transport order), as well as the two B-frames at the start of the next GOP, depend on this frame. Dropping an I-frame renders an entire GOP unusable. So, if frames need to be dropped, dropping B-frames over P-frames is preferable. If the bandwidth is really low, it is better to drop a P-frame than an I-frame.

1.4.2 Payload format for MPEG

As we have seen, an MPEG stream contains a considerable amount of internal dependencies. A dropped frame may render other, uncorrupted data useless. Therefore, the RFC 2250 protocol exists for transmitting MPEG video through RTP in such a way that certain error recovery strategies can be applied in case data is lost [5]. This protocol splits up an MPEG file according to the following rules:

- An MPEG Video-specific header, specified by the standard, must be at the top of the payload.
- The MPEG Video_Sequence_Header, if present, must be at the top of the payload, directly following the MPEG Video-specific header.
- The MPEG GOP_Header, if present, must be at the top of the payload, but after a Video_Sequence_Header.
- The MPEG Picture_Header, if present, must be at the top of the payload, but after a GOP_Header.
- Any sequence of headers must be completely contained within one packet.

This encapsulation of the MPEG headers makes it easier for the decoder to detect and recover from data loss. The loss of a GOP_Header or a Picture_Header can be detected by matching a counter to the temporal reference field in the MPEG Video-specific header. The standard also describes ways to reconstruct these headers, if desired. Lost payload data can not be reconstructed and is dealt with by discarding the entire frame that it belongs to.

This payload format was designed to be used with RTP, but with HTTP it might prove useful as well, possibly in a slightly modified form.

1.5 IFD

The *I-Frame Delay (IFD)* algorithm [10] was developed to improve the quality of an MPEG stream over a link with a low or varying bandwidth. Depending on the transport protocol used, two things may happen when bandwidth drops to a level below that which is required for the stream to arrive completely and in real time.

If the transport protocol is UDP (with RTP on top), data will be lost. When the send buffer overflows, new data packets will be discarded. An MPEG frame is typically composed of 20 to 60 UDP packets. If one packet is lost, most decoders discard the entire frame. This will lead to seemingly random frame drops in the stream, which may have very visible consequences (in the form of artefacts). In the worst case, a loss of a single data packet belonging to an I-frame causes an entire GOP to be corrupted.

If the transport protocol is TCP, the stream will lag. When the send buffer overflows, TCP will block the sending process until the buffer is ready to accept new data. If the available bandwidth on the link is sufficiently low, this causes a noticeable slowdown of the stream.

To solve these issues, the IFD algorithm is placed on top of the transport protocol. It monitors the network condition, and if the bandwidth is too low it starts to drop frames selectively.

```
void buffer_enqueue()
{
    while (true)
    {
        while (C is empty)
            do_nothing();
        if (W is empty)
            store(C in W);
        else
        {
            if (C holds an I−frame)
                overwrite(W with C);
            else if (C holds a B−frame)
                discard(C);
            // C holds a P−frame
            else if (W holds an I− or P−frame)
                discard(C);
            else
                overwrite(W with C);
        }
    }
}
```

Figure 1.1: IFD algorithm in pseudo-C [10].

IFD uses three buffers for this: the current (C), the wait (W), and the send (S) buffer. Each of these buffers is capable of holding exactly one MPEG frame.

Incoming MPEG data is parsed and divided into frames. Frames are assembled in C. When the frame in C is complete, IFD tries to place it in W. If W already contains a frame, IFD compares the type of both frames. It keeps the most important one and discards the other, so that only B-frames (and occasionally a P-frame) may be dropped. In Figure 1.1 we see how this works in pseudo-C. Simultaneously, the frame present in S is being transmitted by the operating system. When S is empty, the frame in W is moved to S. If both are empty, the operating system waits until a new frame is available.

An optimization of this algorithm introduces a flag that signals if an I- or P-frame was lost in the current GOP. If so, all subsequent frames in the GOP will be discarded, as they cannot be displayed properly anymore. As a consequence, the first two B-frames of the next GOP must be discarded as well, as they partially depend on the last P-frame of the current GOP, as we have seen in Section 1.4.

A pseudo-C implementation for this optimization can be seen in Figures 1.2 and 1.3. Note that this optimized version of IFD was *not* used in the measurements described in Chapter 3.

```
void buffer_enqueue()
{
    while (true)
    {
        while (C is empty)
            do_nothing();

        // Reset flag, so we do not drop
        // more than strictly necessary.
        discard_rest_of_GOP = false;

        // Determine if the GOP was disturbed.
        if (C holds an I−frame)
        {
            // An I−frame signals the start of a new GOP.
            // The current, new GOP is not disturbed,
            // but the previous one might be.
            if (GOP_disturbed)
                prev_GOP_disturbed = true;
            GOP_disturbed = false;
        }
        else if (C holds a P−frame)
        {
            // If the GOP was already disturbed,
            // discard the rest of the GOP.
            if (GOP_disturbed)
                discard_rest_of_GOP = true;
            // When we reach the first P−frame, it is no longer
            // important whether the previous GOP was disturbed,
            // because the P−frame from the previous GOP is no
            // longer needed to decode frames from the current GOP.
            prev_GOP_disturbed = false;
        }
        else if (C holds a B−frame)
        {
            // If the current GOP was disturbed, discard the rest.
            // If the previous GOP was disturbed, discard the rest
            // (until the first P−frame is encountered).
            if (GOP_disturbed || prev_GOP_disturbed)
                discard_rest_of_GOP = true;
        }

        ⋮
```

Figure 1.2: Optimized IFD algorithm in pseudo-C. Continued in Figure 1.3.

```
        ⋮

        // If the GOP was disturbed, then
        // discard C, regardless of its type.
        if (discard_rest_of_GOP)
            discard(C);

        // If the GOP was not disturbed,
        // perform IFD normally.
        else if (W is empty)
            store(C in W);
        else
        {
            if (C holds an I−frame)
                overwrite(W with C);
            else if (C holds a B−frame)
                discard(C);
            // C holds a P−frame
            else if (W holds an I− or P−frame)
            {
                discard(C);
                // We just discarded a P−frame,
                // so the current GOP is now disturbed.
                GOP_disturbed = true;
            }
            else
                overwrite(W with C);
        }
    }
}
```

Figure 1.3: Optimized IFD algorithm in pseudo-C. Continued from Figure 1.2.

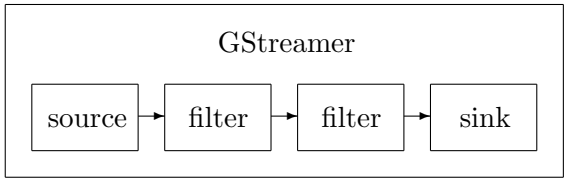


Figure 1.4: A GStreamer pipeline.

1.6 GStreamer

We set up the streams with a tool called GStreamer [4]. GStreamer is an open source application for Linux that is specifically designed to set up media streams dynamically. It allows *elements* to be strung together into a pipeline through which the data streams [19]. These elements each have one specific task, such as reading from a file, decoding MPEG video, displaying images on the screen.

There are three sorts of elements: source, sink and filter. Source elements produce data and sink elements consume them. Filter elements receive data from a source or another filter, process the data and pass it on to the next element. A Gstreamer pipeline must always start with a source and end with a sink. There can be zero or more filters between them. Figure 1.4 illustrates this.

Elements with similar tasks are grouped together in *plugins*. For instance, the elements tcpsink, which transmits data over TCP, and tcpsrc, which receives data from TCP, form the tcp plugin. GStreamer can easily be extended through addition of custom plugins [1]. The concept of plugins exists for the benefit of development and installation of elements and has no influence on the execution of GStreamer pipelines.

Data travels between elements in the form of packets which are, confusingly, called *buffers*. Filter elements often apply some modification to these buffers, if their task is simple. Sometimes, though, they use the incoming buffers to construct new buffers. They send the new buffer to the next element in the pipeline, and release the old one.

Buffers consist of actual data, and of meta-data. The most important part of this meta-data is the timestamp. Not all buffers have timestamps, but buffers containing time-sensitive data (such as MPEG frames) usually do. Normally, GStreamer's scheduler will allow buffers to pass through the pipeline as quickly as possible. However, some elements, usually sinks, inspect a buffer's timestamp and compare it to GStreamer's internal clock, to see whether a buffer is early or late. In the former case, the element will wait until the buffer is no longer late before sending it out, thereby slowing down the stream. In such a case, GStreamer's scheduler will allow other elements to work in the mean time.

If a buffer is late, the stream is said to be *out of sync*. When a buffer is more than two seconds late, GStreamer also produces a warning on the screen. This often happens with non-live streams when there is insufficient available bandwidth on the link: the delay increases steadily, until at some point, a buffer is more than two seconds late and a warning message is produced. All subsequent buffers that are more than two seconds late each produce a warning, causing the screen to fill up quickly. (See Section A.6.4 for instructions on how to remove this warning.)

The version of GStreamer used in this project, was 0.6.5. This was not the most recent

Element	Plugin	Purpose
Sinks:		
filesrc	Core	Reads data from a file on a local filesystem.
udpsrc	udp	Receives UDP packets from the network.
tcpsrc	tcp	Receives TCP packets from the network.
httpsrc	httpsrc	Streams data from a remote file through HTTP.
gnomevfssrc	gnomevfs	Streams data from a local file or from a remote file through HTTP.
Filters:		
rfc2250enc	mpegstream	Encodes MPEG data according to the RFC 2250 protocol [5]. Modified by Ralph Meijer [11].
mpegstat	mpegstream	Displays statistics on MPEG data. Written by Jan Ouwens.
rtpmpegenc	rtp	Packetizes RFC 2250 encoded MPEG data to RTP packets. Written by Ralph Meijer [11].
rtpmpegparse	rtp	De-packetizes RTP packets and subsequently decodes RFC 2250 encoded MPEG data. Written by Ralph Meijer [11].
mpeg2dec	mpeg2dec	Decodes MPEG data to image frames.
Sinks:		
filesink	Core	Writes data to a file on a local filesystem.
fakesink	Core	Disposes of data without using it.
xvideosink	xvideosink	Displays image frames on the screen.
udpsink	udp	Transmits data over the network using UDP.
tcpsink	tcp	Transmits data over the network using TCP.
ifdsink	ifd	Applies the IFD protocol and transmits data over the network using TCP. Written by Jan Ouwens; see Chapter 2.

Table 1.1: GStreamer elements.

version, but the most widely used within the OASIS cluster and therefore the most practical to use. Table 1.1 contains a list of the most important GStreamer elements that were used, the plugins that they reside in, and their purpose. For more information on installing, using, and modifying GStreamer, see Appendix A.

The easiest way to display a local movie, is to use the following pipeline:

filesrc – mpeg2dec – xvideosink

The filesrc element reads an MPEG-2 file from a local filesystem and feeds it into the GStreamer pipeline. The mpeg2dec element decodes the stream, and the xvideosink element displays the decoded frames on the screen.

Setting up a video stream to be transmitted over UDP is more complicated. Two pipelines are necessary; the first is the pipeline on the sending machine, the second on the receiver.

filesrc – rfc2250enc – rtpmpegenc – udpsink
udpsrc – rtpmpegparse – mpeg2dec – xvideosink

The filesrc reads a file. rfc2250enc parses the MPEG stream and divides the frames into chunks that will fit inside an RTP packet, according to the RFC 2250 protocol. rtpmpegenc adds the RTP headers, and finally, udpsink adds UDP headers and transmits the packets. On the receiver, the udpsrc element receives the packets. rtpmpegparse reverses the work of both rtpmpegenc and rfc2250enc. The mpeg2dec and xvideosink elements decode the MPEG stream produced by rtpmpegparse and display it on the screen.

For TCP, the pipelines are easier. Again, the first is the sender, and the second the receiver:

filesrc – tcpsink
tcpsrc – mpeg2dec – xvideosink

No rfc2250enc or rtpmpegenc are necessary, because the former's purpose is to provide a means of recovering from the loss of individual packets, which is not an issue for TCP, and the latter's purpose is to introduce order to the sequence of packets, which TCP already does.

For HTTP, no GStreamer pipeline is necessary on the sending machine, as this is handled by a webserver. For the receiver, the following pipeline can be used:

httpsrc – mpeg2dec – xvideosink

To learn how to actually build these pipelines in GStreamer, the reader is referred to Section A.3.

Note that two plugins are listed in Table 1.1 for use with HTTP: httpsrc and gnomevfssrc. The former is designed specifically for HTTP. It is based on GNOME's libghttp library, which is no longer being actively developed. It is being replaced by the GNOME Virtual Filesystem, which in turn is the basis of the gnomevfssrc plugin. The main advantage, and at the same time the main disadvantage, of gnomevfssrc, is the fact that it supports many different filesystems, both local, such as ext3 and FAT, and remote, such as HTTP and FTP. While this makes the plugin very flexible, it also makes it quite bulky with features that take up space and will probably never be used. However, as the libghttp is an open source project, there is no need for the original developers if bugs turn up.

2 IFD plugin

2.1 Purpose

A working implementation of the IFD algorithm, on top of UDP/RTP, has already been made by Sergei Kozlov, in the form of a queueing discipline inside the transport layer of the Linux kernel [10]. While this implementation works quite well, it has a number of disadvantages:

1. It depends on the sending application to tag UDP packets. The stream arrives to the algorithm in the form of individual UDP packets, instead of complete MPEG frames. The algorithm therefore needs this tagging to identify the MPEG frame type (I, P or B). This means a modification of the sending application is necessary, as this tagging of packets is non-standard.
2. It does not work for TCP based streams, for two reasons. Firstly, as mentioned above, it depends on the tagging of packets. Since TCP is handled by the operating system as a continuous stream of data, and not as a sequence of individual packets, there are no packets to tag. Hence, there is no way to detect the type of MPEG frame being transmitted. Secondly, being located inside the transport layer, it is incapable of dropping TCP segments, as TCP would simply retransmit them.
3. When based on UDP, an instance of the algorithm must be present on every node of the stream's path through the network, or at least on the nodes with a low outgoing bandwidth, since emission speed on each node is determined by the available bandwidth between the current and the next node. As a consequence, if the stream is transmitted wirelessly through an access point, IFD should run inside the access point, and not (only) on the sending machine.

Placing the algorithm in the application layer overcomes disadvantage 2. By parsing the MPEG stream and buffering it according to the rules of IFD, we overcome disadvantage 1 as well. While it is also possible to parse the MPEG stream from within the Linux kernel, this is neither pretty nor practical, since the network layer is not the place to be parsing MPEG data. In the application layer, we may be able to reuse the application's MPEG parsing routines.

With IFD in the application layer and TCP as transport protocol, disadvantage 3 can also be overcome, since for TCP, the emission speed is determined by the minimum of the available bandwidths between the nodes along the stream's route across the network. Therefore, the emission speed applies to the entire route, and not just to the first hop. If this transmission speed is not high enough to support the stream's bitrate, IFD can be applied.

IFD will probably not yield better performance in itself when used over TCP instead of over UDP. The main advantage lies in the fact that, as said above, only the sender needs to

type	in	out	comment
Chain	*push*	*push*	Preferred for most elements.
Loop	*pull*	*push*	Will block on a pull if the previous element is unable to provide new data immediately.
Get	-	*pull*	Used for source elements.
Decoupled	*push*	*pull*	Implemented as a combination of *chain* and *get*.

Table 2.1: Input/output behaviour of GStreamer element types [19].

have a running instance of IFD, while for UDP, IFD needs to be present on each node along the stream's path.

Since GStreamer is easily extensible through its plugin system, and since it is already used heavily within the OASIS cluster, we want to make an IFD plugin that can be used within GStreamer. It must support TCP as transport protocol. UDP is not necessary at this point, as IFD is already available for UDP.

The plugin should be able to perform the following tasks:

1. **Parse incoming MPEG data**
   Data must be separated into frames, and the frames must be identified as I, P or B.
2. **Enforce timing**
   For a movie of 25 frames per second, a frame must be emitted no more often than every $\frac{1}{25}$ s (40 ms).
3. **Perform IFD**
   If the TCP connection cannot handle 25 frames per second, a frame must be dropped according to the rules of the IFD algorithm.

2.2 Design and implementation

In the previous section, we have decided that our new IFD-over-TCP implementation should be made in the form of a GStreamer element. We have seen in Section 1.5 that IFD consists of two simultaneously operating parts: the part that parses the incoming MPEG stream, performs the actual IFD algorithm and places frames in the wait buffer W, and the part that takes frames from W and transmits them over the network. This seems to translate nicely into two elements: an IFD filter element, and a network sink. For this to work, we need the network sink to be able to pull MPEG frames from the IFD element. The behaviour of existing GStreamer network sinks, however, is such that data is pushed into them. We will have to rewrite the relevant sink elements (tcpsink and, possibly, udpsink).

Table 2.1 lists the combinations of input and output behaviour that GStreamer allows. It is possible that data is pulled out of an element, instead of pushed out, using a *get* element or a *decoupled* element. Since our element is not a source element, we need the decoupled type for our IFD element.

The first task the element needs to perform, is parsing the incoming MPEG stream. The mpegstream plugin already contains a function library for this purpose that we can use. However, this library requires a *loop*-based element. This excludes the use of the decoupled type. Adapting the library to be compatible with a push-input behaviour would mean practically rewriting the parser. Since rewriting the MPEG parser seemed more complex than changing from pull-out to push-out, we decided to switch to a loop-based element and use the existing MPEG parser.

In order to decide at what times to push out an MPEG frame, it is necessary to know the load of the network. Determining the load of the network is relatively easy for TCP: we just need to look at the available room of the TCP send buffer when a new frame is available. If there is enough room to hold the frame, it can be sent. If there is not, a frame needs to be discarded.

Unfortunately, it seems there is no portable way to read the state of the send buffer directly from the application layer in Linux. Behaviour of system calls may vary between platforms, and even between Linux kernel versions[1]. And there are other issues. What if the tcpsink writes to a different network interface than the one our IFD element polls? What if a udpsink is used instead of a tcpsink?

To avoid these issues, we decided to merge the IFD filter element and the network sink into a single IFD sink element, which we call ifdsink.

This way, we can determine the network load by simply writing to the buffer and seeing what happens. A call to Linux's write operation returns the number of bytes actually written to the socket's buffer. If this number is less than the amount of bytes we tried to write, we know the buffer is full and that we have to resend the remaining bytes.

Waiting for the buffer to become empty so the application can send the next frame costs time, and it's a pity to spend it idling. It can better be used to parse MPEG data. The framerate (usually 25 frames per second) can be another source of waiting. In order to reduce the amount of busy waiting in the system, we introduce two threads: one to parse the MPEG data, and one to transmit frames via TCP. In the existing implementation for UDP, this is not an issue, as it writes directly into the kernel's send buffer. Where we use a thread to read from this buffer and transmit, the existing implementation simply uses the mechanism that already exists in the Linux kernel, and needs only to focus on parsing the MPEG data and deciding which frames to decide, and when.

An additional advantage of threading is that there is no need to determine the load of the network actively. It is possible to make the write call blocking, so that a call to it will not return until the entire frame has been written. While the write call is blocked, another thread can parse MPEG data.

Figure 2.1 shows the structure of the proposed IFD element, which is basically a simple producer/consumer system. There are two threads, each with their own buffer. The first thread is the MPEG Parser. It fills its buffer C with an MPEG frame. As soon as the frame is complete, it reads the time. When the frame's timestamp corresponds to the current time, it offers the frame in C to the shared buffer W according to the rules of IFD. After that, it starts parsing a new frame.

[1] At the time of writing, the current version of the man page for the ioctl system call says: 'CONFORMING TO: No single standard', which says enough.

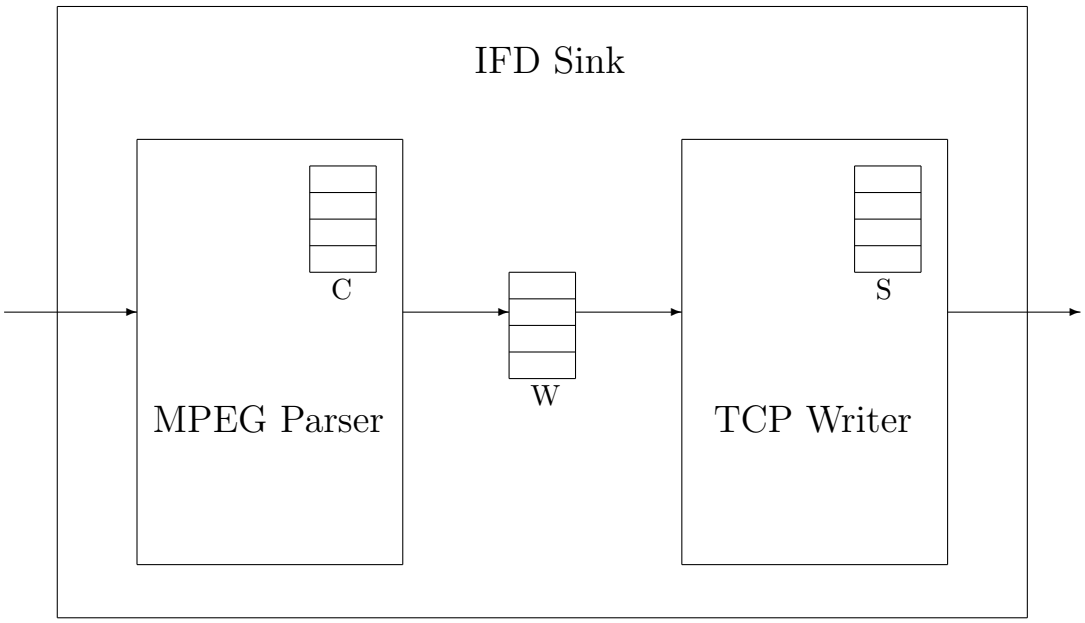


Figure 2.1: Structure of the GStreamer ifdsink element.

The other thread is the TCP Writer. It waits until W contains a frame, and copies it to its own buffer S. It then starts to write S to TCP. When S is completely written to TCP, it waits again until a new frame is available from W.

Figure 2.2 shows this in pseudo-C.

2.3 Recommendations for UDP

The discussion in the previous section focused mainly on implementing IFD with TCP as transport protocol. However, most considerations also apply to UDP. There is, however, one caveat: the socket is assumed to be blocking. On Linux machines, this is true for TCP. For UDP, however, it is not. In case the buffer is full, it simply overflows and data is lost.

There are several ways to make the Linux kernel's UDP buffer blocking. One is to set maximum socket buffer size to a value smaller than the network interface's txqueuelen setting. This can be achieved within a program using the sysctl system call, or from the command prompt by echoing an appropriate value to the file /proc/sys/net/core/wmem_max. The reasoning behind this, is that the socket buffer is a representation of the interface's buffer. If its size is larger than that of the interface buffer, everything added to the socket buffer when the interface buffer is full, will simply be discarded. If its size is less than the interface buffer's, the latter can not overflow.

Note that data may still be lost during transmission through unreliable media. Also, note that disadvantage 3 in Section 2.1 still applies.

```
thread MPEG_Parser()
{
    while (true)
    {
        get(raw_MPEG_data);
        C = parse(raw_MPEG_data);
        sync_clock ();
        mutex_lock();
        // Critical section begins
        IFD(C, W);
        // Critical section ends
        mutex_unlock();
    }
}
```

```
thread TCP_Writer()
{
    while (true)
    {
        await( W is not empty);
        mutex_lock();
        // Critical section begins
        S = W;
        make_empty( W);
        // Critical section ends
        mutex_unlock();
        tcp_write (S);
    }
}
```

Figure 2.2: IFD plugin in pseudo-C.

3 Measurements

3.1 Purpose

There are many factors that affect the quality of a video stream, some more subtly than others. However, for only a relative few of these factors, their influence is well understood. In this chapter, we will look at some of these factors and, through measurements, try to gain a better understanding of their impact.

In accordance with Philips's vision of the connected home [9], we will focus our measurements on a wireless home network environment. This choice eliminates a number of factors, which we will therefore not investigate. We will use the IEEE 802.11 protocol [6], more specifically its extension IEEE 802.11b [7], and we will restrict ourselves to a situation where the stream is the only contender for the available bandwidth, unlike the competitive environment of the internet. As a result, the conclusions we draw in this chapter apply to the home environment, and not to the internet.

Factors that we will investigate include the choice of transport protocol, the presence of interference on the wireless link, the IEEE 802.11 retransmission setting, and MPEG movie bitrate.

3.2 Set-up

The following equipment was used for the measurements:

- A laptop, called 'sender'. This was a Dell Latitude D600 with Pentium M 1.6 GHz, 512 MB RAM, 80 GB HDD and Cisco Aironet 350 wireless card, running Fedora Core 2 with Linux kernel 2.6.10.
- A PC, called 'receiver'. This was a MyCom computer with Pentium 4 2.8 GHz, 512 MB RAM and 80 GB HDD, running Debian GNU/Linux 3.1 (Sarge) with Linux kernel 2.4.25.
- A laptop, called 'sniffer'. This was a Dell Inspiron 4150 with Mobile Pentium 4 1.8 GHz, 512 MB RAM, 30 GB HDD and Cisco Aironet 350 wireless card, running Knoppix 3.8.1 (installed on HDD) with Linux kernel 2.6.11.
- A LinkSys WRT54G access point, operating in IEEE 802.11b mode.
- A Samsung RE-1300 microwave.

A lot of wireless devices are present at the High Tech Campus. These devices can interfere with our measurement setup, causing unpredictable results. To eliminate this problem, we decided to do the measurements at my home, where less interference was present. However, one, and sometimes two, other access points were visible to my network. Even though they were not very active at the times I performed the measurements, I have performed each

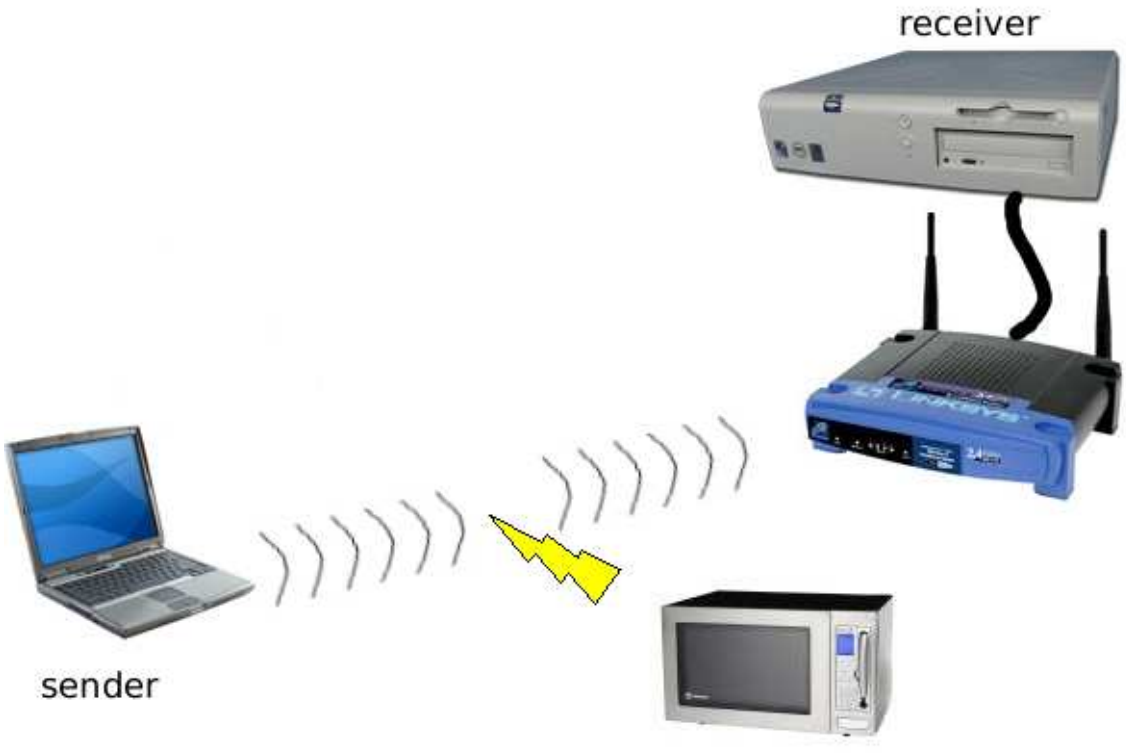


Figure 3.1: Set-up (not on scale).

measurement three times: once in the morning, once in the middle of the day, and once in the afternoon, to minimize the influence of interference from these access points.

The laptop was placed at one end of a room, the PC and access point at the other. The distance between laptop and access point was approximately 4 m. The access point had a wired connection to the receiver PC. The microwave was placed in the line of sight of both the laptop and the access point, about 50 cm distant from the direct path between the laptop and the access point. See also Figure 3.1.

Both computers were equipped with a custom build of GStreamer 0.6.5 [4]. An element called mpegstat was written to extract statistics. This element parses an incoming MPEG stream into GStreamer buffers containing one MPEG frame each. Next, it assigns to these GStreamer buffers timestamps matching the MPEG frames' timestamps. It does not, however, take action based on these timestamps; for instance, it will not pause if it detects that a frame is early. Finally, mpegstat outputs statistics for each complete MPEG frame –it discards incomplete frames–, as well as for each GOP header it encounters. The element deliberatly produces more statistics than strictly needed for the measurements. This is done partly because it facilitated debugging the element, and partly because it may be useful outside the scope of these measurements as well. Note that all statistics are gathered from the MPEG data itself, and from GStreamer's internal data structures. No extra data is transmitted in the stream for the mpegstat element.

For MPEG frames, the following statistics are given:

- The current timestamp, in nanoseconds from the moment the first UDP or TCP packet was received by the receiving instance of GStreamer.
- The timestamp (in transport order) at which the frame in question was expected to be played, in nanoseconds from the first transmitted frame in the stream.
- The difference between these two timestamps. This value is the latency; see Section 3.4.
- The type of the frame in question (I, P or B).
- A temporal reference: the sequence number, in display order, of the frame within the current GOP. This value is used to rearrange the frames, which arrive in transport order, back into display order and as such can be used to detect frame losses.
- An 'RTP timestamp': the timestamp, in display order, that the rtpmpegenc element would have assigned to the frame, based upon its actual transport order timestamp, and its place in the GOP in display order.
- The size of the frame in question, in bytes.

For each encountered GOP header, the following statistics are given:

- A timestamp, from the first frame of the stream, in hh:mm:ss format.
- The 'pictures' value. This number corresponds to the first frame in the GOP, and signifies its frame sequence number since the beginning of the current second. It is reset every second. This value is used by the MPEG decoder for timing, since the length of a GOP is not usually a practical number to calculate with.
- The 'RTP-timestamp' (see above) of the frame that has a tempref of 0 within this GOP.

Of these values, especially the difference between the two timestamps, the frame type, the temporal reference and the size are useful for the analysis of the measurements.

Other tools that were used are Ethereal 0.10 [2] and Iperf 2.0.1-1 [8].

The movie used was the famous penguin movie (see Figure 3.2), cut to a length of 1 minute, at bitrates of 3, 4, 5 and 6 Mbit/s. The movies' internal GOP structure was the same for each movie of the same length; the only difference was the byte size of the individual frames.

3.3 Parameters

Table 3.1 contains a list of the parameters that we varied in the measurements, along with the specific values that we chose for each parameter.

We measured two variants of UDP: blocking and non-blocking. The difference is in the behaviour of the send buffer. The former uses a blocking buffer that makes the sending process wait until the buffer is empty enough to be able to accept new data. Due to this waiting, the stream may be transmitted slower than real-time. Live streaming is therefore

Parameter	**Values**
Protocol	• blocking UDP + RTP • TCP • non-blocking UDP + RTP • TCP + IFD
Retry value	• 0 • 1 • 2 • 3 • 4 • 8 • 16 • 256
MPEG bitrate	• 3 • 4 • 5 • 6
Interference	• no • yes
Time of day	• morning (8h30 - 11h10) • mid-day (11h10 - 14h20) • afternoon (14h20 - 17h00)

Table 3.1: Measurement parameters.

Figure 3.2: Walking on ice.

not guaranteed with blocking UDP. With non-blocking UDP, on the other hand, the send buffer can simply overflow. The data that overflows is lost, but there is no waiting, so the stream is always live. Non-blocking is the default behaviour for UDP on Linux systems.

One purpose of these measurements is to get more insight into TCP as a transport protocol for video streaming. As TCP's behaviour is always blocking, and hence non-live, we chose to include blocking UDP to be able to compare two non-live protocols. Also, we hoped including blocking UDP and comparing it with non-blocking UDP would help us to see how much data is lost in the send buffer, and how much data is actually lost on the air. Finally, we included the TCP-with-IFD algorithm to test the implementation, and also to compare its performance with an often used live streaming protocol: non-blocking UDP.

All measurements have been performed with the same MPEG movie, although we had to use different files for the different bitrates. For each of the bitrate, the movie itself, as well as the internal GOP structure, remained the same. The movie's length was one minute, in all cases.

The interference was generated with a microwave. As the movie was always one minute in duration (or sometimes longer in the case of non-live streams), we decided to let the movie run for 20 seconds without interference, then turn on the microwave for another 20 seconds, and then let the movie finish without interference. This way we hoped to be able to see what happens during the transitions, as well as during the interference itself.

We have divided the day in three parts and performed each of the measurements once for each part of the day, to minimize the influence of the (already quite low) interference from outside our control (more specifically, neighbours' wireless networks).

It was impossible to set the maximum transmission rate for our wireless card, the Cisco Aironet 350, under Linux, hence its absence from the parameter list.

3.4 Quantities and expectations

The quantities we are interested in, are throughput, duration for non-live protocols, loss for lossy protocols, latency, and jitter. For video streaming, it is important to know the values of these quantities for MPEG frames, and not so much for the UDP or TCP packets that contain the frame. Therefore, all measurements in this chapter, except otherwise noted, apply to MPEG frames, and not UDP or TCP packets. This has a profound impact on the results of the measurements. For example, the loss of one UDP packet causes the loss of a complete MPEG-2 frame, which consists of several UDP packets. Therefore, loss will always be higher for MPEG frames than it will be for UDP packets.

We want to measure throughput, because this is the deciding factor in the size of the stream that can be transmitted. The more throughput, the higher the movie's bitrate can be, and the higher the quality of the video. We expect TCP to have the lowest throughput, due to its retransmission mechanism and its congestion control algorithms, by which UDP is not hindered. For the throughput graphs, we will use a sampletime of 1 s, so that, when the movie is streamed in real-time, we have 25 frames in a sample.

For TCP, if the throughput goes down, the movie will need more time to be transmitted. The movie will then run slower, which is undesired. However, for blocking UDP, a lower throughput does not necessarily mean a longer movie duration, as two mechanisms are at work. The send buffer may block, as with TCP, in which case the stream's duration will increase. Also, packets may be lost on the air. This causes a lower throughput, but has no effect on the duration. In both cases, however, the duration will never be less than 60 s, because even if the stream could be transmitted faster, GStreamer will slow down the stream to real-time. We expect that the duration will increase with the movie's bitrate, as higher bitrate means bigger files and more transmission time. For TCP, lower retry values will also cause delays, due to more TCP retransmissions, and the effect of TCP's congestion control algorithms. For the live protocols, non-blocking UDP and TCP with IFD, the duration of the stream will always be 60 s.

TCP is lossless, but the other protocols are not. Since the loss of a B-frame has only a minimal effect on the perceived quality of a video stream, while the loss of a single I-frame can cause annoying artifacts in the image for almost half a second, it is important to know exactly what type of data is lost. For both UDP variants, we expect a relatively high amount of loss of I-frames, and slightly less P- and B-frames, respectively, as I-frames are larger and hence more likely to be lost than P-frames, and P-frames are more likely to be lost than B-frames. When we apply IFD, however, we expect almost no I-frames to be lost, only few P-frames, and relatively many B-frames.

We also look at consecutive loss of UDP packets. The loss of one packet causes the loss of an entire frame, since the decoder cannot decode an incomplete MPEG frame. Thus, ten consecutively lost packets are preferable to ten individual losses throughout the stream. In the former case, all losses may occur within only one frame, causing only one frame to be lost, while in the latter case, ten MPEG frames may be lost.

Latency is an indication of a frame's lateness. If, for a frame i, S_i is its MPEG timestamp in milliseconds, i.e. the time it is expected to have arrived, and R_i is its arrival time in

milliseconds, then the frame's latency L_i is calculated as follows:

$$L_i = R_i - S_i$$

It can tell us, for instance, how much later frames will be when interference is present, and if the stream can recover again when the interference ends.

Also, there is a conjecture that the length of an MPEG frame has an influence on its latency: that it will take noticably longer for a large frame to arrive, causing some jitter between larger and smaller frames.

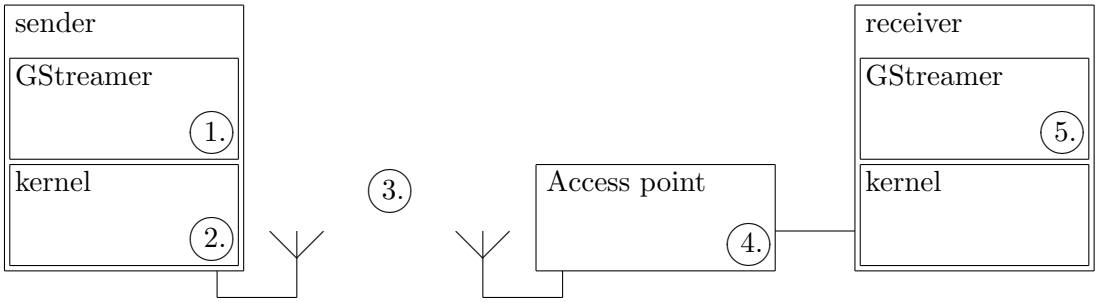
Finally, the jitter is the time that elapses between the arrival of two consecutive frames. We calculate the jitter J continuously as each packet i arrives, using the RTP jitter algorithm described in [16], as follows:

$$J = J + \frac{|L_{i-1} - L_i| - J}{16}$$

where the factor 16 is used for noise reduction [16].

In the MPEG movie we use, there are 25 frames per second. That means one frame every 40 ms. So if the jitter remains below these 40 ms, a buffer of one frame is sufficient to be able to show one frame every 40 ms. If the jitter increases, however, more frames need to be buffered in order to display the frames in time. The higher the jitter, the more frames need to be buffered. Therefore, we want the jitter to remain as small as possible.

The following sections describe the actual measurements done. For each set of measurements, the set-up is described for as far as it differs from the general set-up described in Section 3.2. At the least, the exact GStreamer pipelines on the sender and the receiver, and a short description of where data loss may occur are given. Figure 3.3 describes the possible locations at which data can be lost.



#	Location	Probability
1.	Sender's GStreamer pipeline	Only possible if data is dropped here deliberately.
2.	Sender's socket buffer	Any data overflowing the buffer is lost.
3.	Air	Possible, especially when interference is present.
4.	Access point's socket buffer	Very unlikely, as the wired connection between AP and receiver is much faster than the wireless connection between sender and AP. We have seen no evidence of this actually occurring in any of the measurements.
5.	Receiver's GStreamer pipeline	Only possible if data is dropped here deliberately. When an MPEG frame arrives incomplete due to the loss of one or more of its constituent packets, the entire frame is dropped.

Figure 3.3: Possible data loss locations.

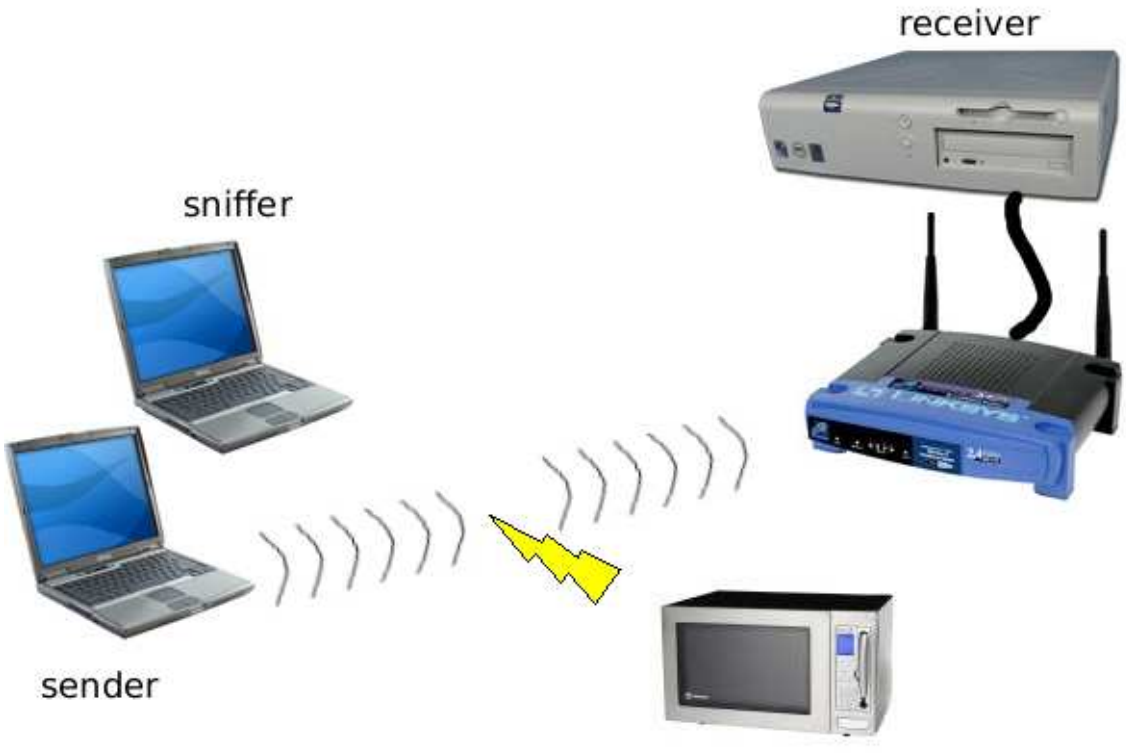


Figure 3.4: Set-up with sniffer (not on scale).

3.5 IEEE 802.11b link layer

Set-up

This set of measurements focuses on the behaviour of UDP and TCP when placed on top of the IEEE 802.11b protocol, regardless of the kind of data. It is important that the stream be continuous, and not bursty like an MPEG stream, where each 40 ms a frame is transmitted. We use GStreamer to set up this stream. To eliminate the burstiness, we use the fakesink element instead of the mpegstat element, and we modify the udpsink element to disregard GStreamer's internal timestamps (See Section A.6.4). For tcpsink, this is not necessary, as it already does not use timestamps.

For this set of measurements, the sniffer was added to the set-up, as depicted in Figure 3.4. This set was performed once using UDP, and once using TCP as transport protocol.

For UDP, the following pipelines were used:

Sender: filesrc – rfc2250enc – rtpmpegenc – udpsink
Receiver: udpsrc – rtpmpegparse – fakesink

The sender's pipeline is a typical pipeline for UDP streaming, as seen in Section 1.6. The receiver has the udpsrc element to receive the data from the network, followed by the rtpmpegparse element which decodes the RTP packets and produces an error message when it

detects the loss of a packet. This message is used to determine loss on the receiver. Finally, the fakesink element discards the data, as it is not needed further.

The udpsink element was modified to disregard timestamps and send as quickly as possible. This way, we have a continuous stream of data, and not the sequence of bursts that we would otherwise have.

The sender's socket buffer was made blocking by setting the value of the /proc/sys/net/core/wmem_max file to the wireless interface's txqueuelen minus one.

For TCP, the following pipelines were used:

Sender: filesrc – tcpsink
Receiver: tcpsrc – fakesink

The sender's pipeline is a typical pipeline for TCP streaming, as seen in Section 1.6. The receiver has the regular MPEG-2 decoder and the screen sink replaced by our mpegstat filter, which generates the data we need for the analysis, followed by the fakesink which simply discards the data.

As an exception to the rule, for these measurements we did not activate the microwave for a short interval, but for the entire time the transmission took place.

Results

The first thing of note is the fact that the sniffer, however close it was placed to the sender, did not pick up a large number of frames. The exact number is hard to estimate, because the sniffer picks up retransmissions, which the receiver does not. If we look only at the unique data frames (and not the retransmissions), we note that, on average, only about 65 % of the frames is picked up by the sniffer. Compare the right-most columns of Tables 3.4 and 3.5, and Tables 3.2 and 3.3.

However, despite this discouraging observation, still some useful information can be obtained from this data.

Table 3.2 shows for each retry value the number of observed transmissions, both in the absence and in the presence of interference, when using UDP. Dashes indicate that a combination of retry value and number of transmissions is not applicable. For instance, if the retry value is 2, a packet is discarded if it does not arrive within 2 transmissions. Therefore, 3 transmissions or more cannot occur. Note that a retry value of 1 means a packet will be transmitted at most one time, without retry. The first retry value at which retries may occur is 2, where packets are transmitted at most two times (one transmission, and one retry).

Each column shows the actual number of packets observed, and a percentage relative to the total unique number of frames picked up by the sniffer. The two right-most columns show the total number of frames; the first not including retransmissions, the second including retransmissions. We see that a vast majority of packets is transmitted only once, even when interference is present. However, in reality these numbers must be lower, due to the fact that the sniffer did not pick up a large amount of packets.

re-	micro	1		2		3		4		5		unique	total
try	wave	#	%	#	%	#	%	#	%	#	%	pkts	pkts
1	**no**	18766	100.0	–	–	–	–	–	–	–	–	18766	18766
	yes	23571	100.0	–	–	–	–	–	–	–	–	23571	23571
2	**no**	16407	99.8	30	0.2	–	–	–	–	–	–	16437	16467
	yes	23060	93.0	1745	7.0	–	–	–	–	–	–	24805	26550
3	**no**	16537	99.7	48	0.3	0	0.0	–	–	–	–	16585	16633
	yes	18584	92.9	1204	6.0	208	1.0	–	–	–	–	19996	21616
4	**no**	15506	99.8	25	0.2	0	0.0	0	0.0	–	–	15531	15556
	yes	17689	91.9	1327	6.9	214	1.1	13	0.1	–	–	19243	21037
5	**no**	15979	99.7	43	0.3	0	0.0	0	0.0	0	0.0	16022	16065
	yes	18622	91.5	1366	6.7	326	1.6	30	0.1	16	0.1	20360	22532

Table 3.2: Number of transmissions per packet on 802.11b level in %, using UDP.

re-	micro	Lost		Received		total
try	wave	#	%	#	%	pkts
1	**no**	36	0.1	29349	99.9	29385
	yes	374	1.3	29011	98.7	29385
2	**no**	0	0.0	29385	100.0	29385
	yes	195	0.7	29190	99.3	29385
3	**no**	0	0.0	29385	100.0	29385
	yes	18	0.1	29367	99.9	29385
4	**no**	0	0.0	29385	100.0	29385
	yes	4	0.0	29381	100.0	29385
5	**no**	0	0.0	29385	100.0	29385
	yes	3	0.0	29382	100.0	29385

Table 3.3: Number of losses on UDP level.

Table 3.3 reflects the same data stream as Table 3.2, but as observed by the receiver instead of the sniffer. It shows the number of packets that were actually reported lost by the receiver, again in absolute numbers, and in percentages. We observe that the amount loss is zero, or very low, even with the microwave on. For example, when no interference is present, at a retry value of 2, no packets were lost. With interference, the number of lost packets decreases with increasing retry value. For 18 packets, a retry value of 3 was too low when interference was present, even though, according to Table 3.2, at least 208 frames had been transmitted three times by the IEEE 802.11b protocol. So, for 190 frames, the three transmissions were enough.

Table 3.4 shows the same parameters, but using a TCP stream instead of a UDP stream. Again, a majority requires only 1 transmission, although this majority is smaller than for UDP.

Table 3.5 shows the amount of retransmissions on the TCP level, in absolute numbers, and in numbers relative to the total amount of unique packets. We see that very few TCP retransmissions have taken place.

Table 3.6, finally, shows the time the stream needed in order to be transmitted completely, for both protocols and for each retry value, with and without interference. There does not appear to be a correspondence between the duration and the amount of lost packets, as indicated by Table 3.3 and the first column of Table 3.5. We do see that, with interference, a stream has a longer duration than without interference. Also, the duration at retry value 1 is significantly longer than the duration at higher retry values.

Conclusions

1. The data gathered by the sniffer was quite unreliable due to its incompleteness. It is not entirely clear why the sniffer performed so poorly. Fortunately, additional data gathered at the receiver allowed us to gain some insight into the working of the protocol. Nevertheless, more accurate measurements are needed for conclusive insight.
2. While a higher retry value does indeed correspond to a lower probability of packet loss, the influence is much smaller than previously assumed. For retry values from 3 upwards, the amount of loss is more or less stable, and even almost negligible when no interference is present. The wireless card's default retry value of 16 seems rather cautious.
3. The retry value does appear to have a bearing on the duration of the stream, as does the presence of interference. It appears that the wireless card transmits packets at a lower transmission rate when link conditions are bad. Separate measurements have confirmed this. However, the IEEE 802.11b specification [6] leaves many details to the interpretation of the manufacturer of the wireless device. Devices other than the one used for these measurements (Cisco Aironet 350) may exhibit different behaviour.

In the remainder of this chapter, we will look mainly, though not exclusively, at retry values from 1 to 4, as higher retry values did not show any significant difference.

re-	micro	1		2		3		4		5		unique	total
try	wave	#	%	#	%	#	%	#	%	#	%	pkts	pkts
1	**no**	14947	100.0	–	–	–	–	–	–	–	–	14947	14947
	yes	15426	100.0	–	–	–	–	–	–	–	–	15426	15426
2	**no**	5743	97.3	160	2.7	–	–	–	–	–	–	5903	6063
	yes	10417	89.2	1260	10.8	–	–	–	–	–	–	11677	12937
3	**no**	5152	97.7	120	2.3	1	0.0	–	–	–	–	5273	5395
	yes	7139	88.7	825	10.2	89	1.1	–	–	–	–	8053	9056
4	**no**	5236	97.7	122	2.3	1	0.0	0	0.0	–	–	5359	5483
	yes	6437	87.0	882	11.9	74	1.0	2	0.0	–	–	7395	8431
5	**no**	4909	97.6	122	2.4	0	0.0	0	0.0	0	0.0	5031	5153
	yes	6234	83.5	1128	15.1	98	1.3	5	0.1	0	0.0	7465	8804

Table 3.4: Number of transmissions per packet on 802.11b level in %, using TCP.

re-	micro	1		2		3		4		5		unique	total
try	wave	#	%	#	%	#	%	#	%	#	%	pkts	pkts
1	**no**	15284	98.9	163	1.1	3	0.0	0	0.0	0	0.0	15450	15619
	yes	15257	98.8	187	1.2	3	0.0	1	0.0	0	0.0	15448	15644
2	**no**	15496	100.0	0	0.0	0	0.0	0	0.0	0	0.0	15496	15496
	yes	15424	99.7	42	0.3	2	0.0	0	0.0	0	0.0	15468	15514
3	**no**	15501	100.0	0	0.0	0	0.0	0	0.0	0	0.0	15501	15501
	yes	15489	100.0	5	0.0	0	0.0	0	0.0	0	0.0	15494	15499
4	**no**	15503	100.0	0	0.0	0	0.0	0	0.0	0	0.0	15503	15503
	yes	15501	100.0	0	0.0	0	0.0	0	0.0	0	0.0	15501	15501
5	**no**	15501	100.0	0	0.0	0	0.0	0	0.0	0	0.0	15501	15501
	yes	15501	100.0	0	0.0	0	0.0	0	0.0	0	0.0	15501	15501

Table 3.5: Number of transmissions per packet on TCP level in %.

	UDP		TCP	
retry value	**without microwave**	**with microwave**	**without microwave**	**with microwave**
1	87.4	243.9	250.2	561.6
2	59.0	155.3	52.5	128.7
3	66.8	75.9	54.2	91.3
4	57.1	92.1	57.8	64.8
5	58.2	82.9	48.4	60.9

Table 3.6: Stream duration in seconds

3.6 Blocking UDP

Set-up

We used the following GStreamer pipelines:

Sender: filesrc – rfc2250enc – rtpmpegenc – udpsink
Receiver: udpsrc – rtpmpegparse – mpegstat – fakesink

The sender's pipeline is a typical pipeline for UDP streaming, as seen in Section 1.6. The receiver has the regular MPEG-2 decoder and the screen sink replaced by our mpegstat filter, which generates the data we need for the analysis, followed by the fakesink which simply discards the data.

The sender's wireless interface's txqueuelen was set to 1000; the value of the /proc/sys/net/core/wmem_max file was set to 999. This ensures a blocking but non-lossy socket buffer. We have done this so no data loss will occur there. Packets may be lost on the air. Data may also be dropped in the receiving GStreamer pipeline, if a frame is received partally, in which case the entire frame is discarded.

Because the sender's socket buffer may block, delays may occur, and the stream does not arrive in real-time any more. Therefore, a stream using blocking UDP as transport protocol is not a live stream under all circumstances.

Results

Figure 3.5 shows for retry values from 1 to 4 how the throughput progresses with time for a 6 Mbit/s movie without interference. This graph –and all subsequent other graphs, unless noted otherwise– was made using the afternoon measurement point. We see that the upper boundary for the throughput lies around 4 Mbit/s. We note how the throughput for retry value 1 is very erratic. It is significantly lower than the throughputs at higher retry values, but still peaks to 4 Mbit/s. This is unexpected; if we compare with Table 3.3 we see that there are losses for retry value 1, but not quite as much as this graph indicates. The explanation for this, is that the transmission rate also decreases with decreasing retry value. The reduced rate is the cause for the decreased throughput. We do not yet know why exactly the transmission rate drops when the retry value is 1. The decision of when and how to adapt the transmission rate is not defined in the specifications [6, 7], and is made by the manufactureres of wireless devices. Possibly, wireless cards other than the Cisco Aironet 350 we have used, behave differently.

Note also how time progresses past the 60 s that the movie would have taken in real-time. This is due to the blocking in the send buffer, and the fact that the movie's bitrate exceeds the effective throughput of the stream.

In Figure 3.6 we see the same parameters, this time with interference. It is clearly visible how the bandwidth drops during the interval in which the microwave is active, and returns to its former level when the microwave is done. The lower the retry value, the bigger the influence of the microwave's interference.

Figure 3.7 shows the throughput at a constant retry value of 4, for different bitrates. For bitrate 3, the movie finishes after 60 s. It only needs a throughput of 3 Mbit/s, and so has

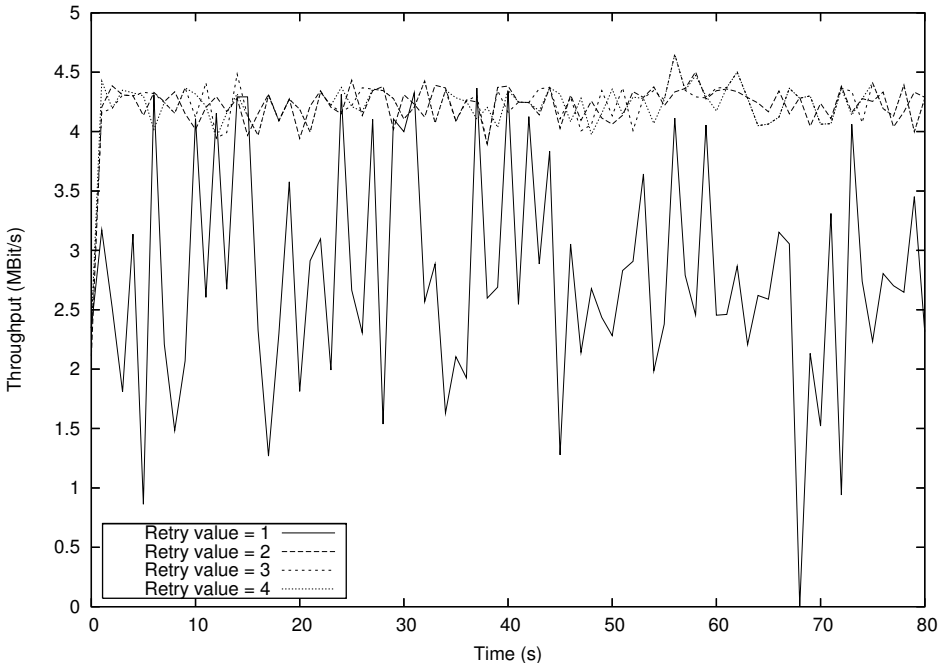


Figure 3.5: Throughput. Protocol = blocking UDP, bitrate = 6 Mbit/s, microwave = no, time = afternoon.

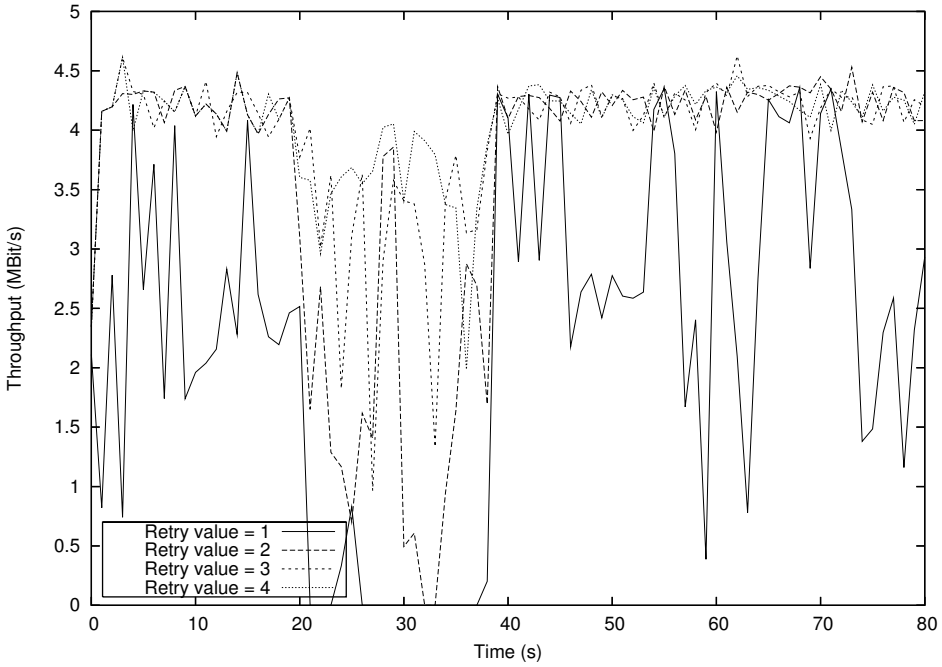


Figure 3.6: Throughput. Protocol = blocking UDP, bitrate = 6 Mbit/s, microwave = yes, time = afternoon.

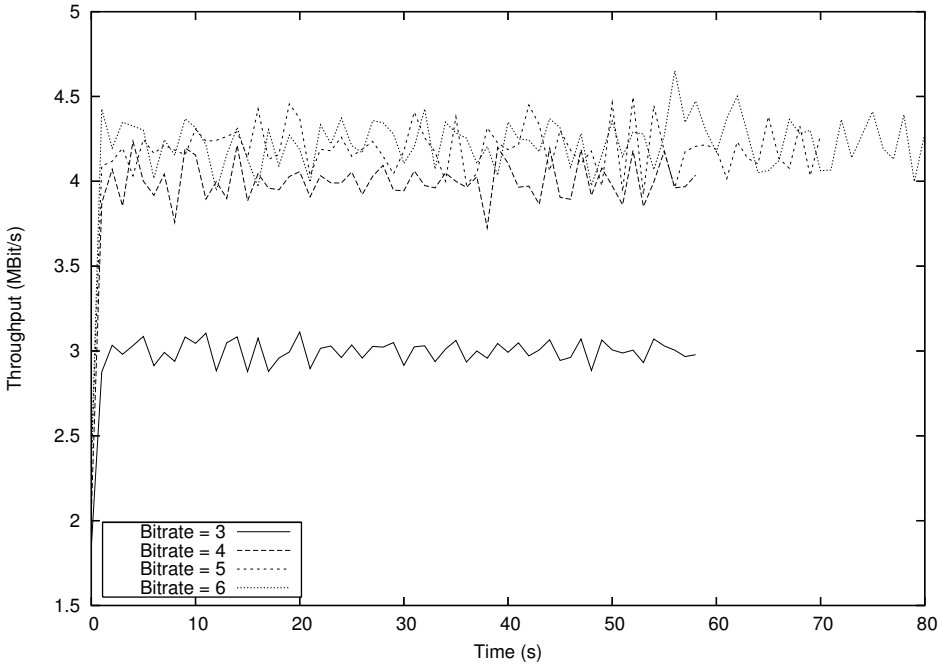


Figure 3.7: Throughput. Protocol = blocking UDP, retry value = 4, microwave = no, time = afternoon.

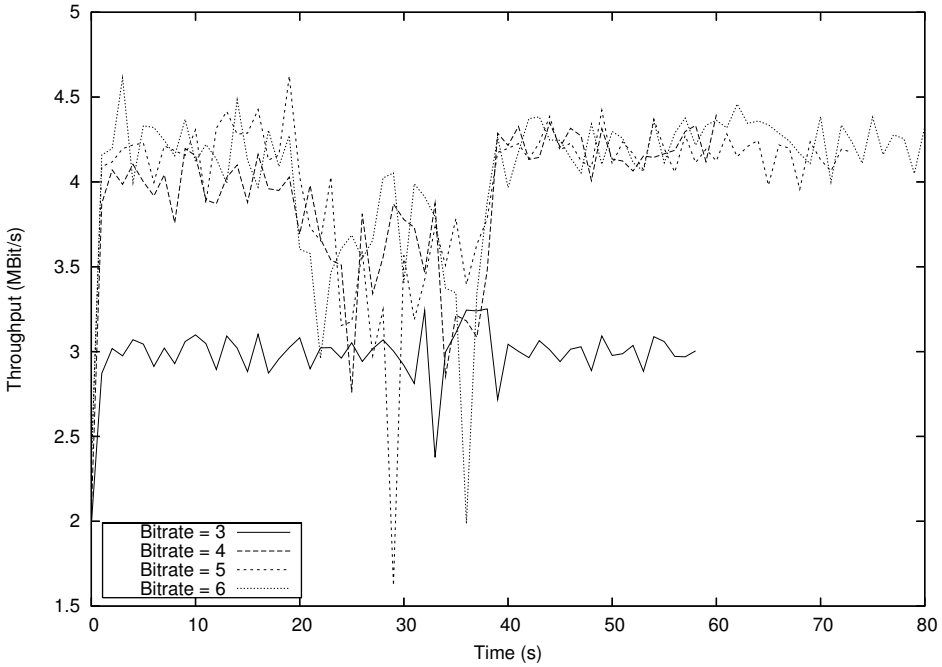


Figure 3.8: Throughput. Protocol = blocking UDP, retry value = 4, microwave = yes, time = afternoon.

some leeway. At bitrate 4, the same thing happens. At bitrates 5 and 6, the stream exceeds the available bandwidth and therefore needs more time to finish.

Figure 3.8 shows what happens when we introduce interference. At 3 Mbit/s, the stream experiences no negative side-effects, apart from a momentary disturbance in throughput. At higher bitrates, the drop in throughput is clearly visible.

The precise durations of the streams can be seen in Table 3.7. The values are the averages of the three measurement points: morning, midday and afternoon. Note how the duration of the stream increases with increasing video bitrate, as higher movie bitrates imply larger files and hence longer transmission times. The duration is also shorter for higher retry values. This is unexpected; we would simply have expected more loss in this case. The explanation for this lies again in a decrease of the transmission rate, which is the cause for the increased duration.

Also, with interference, the duration is higher than without. The retry value 0 apparently is translated to a higher value, probably the default value 16, as a retry value of 0 would otherwise be impossible: at 0 transmissions per packet, a stream would take a very long time to complete. Retry values 8 and 16 offer only slightly better performance than retry value 4. Also, we see that setting the retry value to 1 yields longer durations than turning on the microwave at a higher retry value does.

The amount of loss is shown in Tables 3.8 through 3.11. The tables show for each type of MPEG frame the amount that was lost, and the percentage of that type of frame that was lost. The amounts are the averages of the three measurement points that were performed: morning, midday and afternoon. For example, at bitrate 3 and retry value 1, with interference, an average of 84 B-frames was lost, which equals 7.1 % of all 998 B-frames. We see that the amounts of loss do not differ much between bitrates. Loss occurs at retry values of 1 and 2. For values 3 and higher, loss is negligible.

We see also that, in general, relatively more I-frames are lost than P-frames, and relatively more P-frames than B-frames. We expected this to be the case, because I-frames are larger than P-frames, and P-frames larger than B-frames. Their larger size increases the chance for them to be lost on the way. However, for 6 Mbit/s movies, this does not hold. It is not clear why exactly, but possibly the decreased transmission rate is an influence.

If we compare these tables to Table 3.3, we see that, for retry values of 3 and higher, loss is negligible. For retry values 1 and 2, however, we see that a small amount of UDP loss leads to very large amounts of MPEG-2 frame loss. This is due to the fact that a partial MPEG-2 frame is discarded entirely by the decoder: the data in the UDP packets that belong to a discarded frame is discarded as well.

In Table 3.12, we see, for each permutation of the parameters, how often UDP packets (as opposed to MPEG frames) were lost consecutively. The first column shows the percentage of times when a sequence of 1 UDP packet was lost. The second column shows the percentage of times when a sequence of 2 UDP packets was lost, and so on. The last two columns show the total number of lost packets, and the number of times a sequence of one packet or more was lost, respectively. We see that packets are usually lost individually, and only in some cases in small sequences. Sequences of more than 2 packets are quite rare and only occur when interference is present. Since the loss of a single UDP packet will cause the MPEG decoder

retry value	micro wave	bitrate 3	4	5	6
0	**no**	59.9	60.0	69.9	84.6
	yes	59.9	60.1	72.3	87.3
1	**no**	103.4	133.4	166.8	123.2
	yes	112.1	152.2	189.1	141.4
2	**no**	60.2	63.5	77.7	84.9
	yes	63.5	74.9	86.5	95.3
3	**no**	60.0	60.0	72.2	84.7
	yes	59.9	60.4	73.8	88.8
4	**no**	59.9	60.0	70.7	85.2
	yes	59.9	60.3	73.6	88.2
8	**no**	59.9	60.0	70.0	84.9
	yes	59.9	60.1	72.5	88.1
16	**no**	59.9	60.0	69.9	85.0
	yes	59.9	60.1	72.8	88.2

Table 3.7: Average movie duration in seconds for Blocking UDP

retry	micro	I		P		B		total	
value	wave	#	%	#	%	#	%	#	%
0	**no**	0.3	0.3	0.0	0.0	2.0	0.2	2.3	0.2
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	**no**	14.0	13.9	45.0	11.2	71.0	7.1	130.0	8.7
	yes	13.0	12.9	40.3	10.1	84.0	8.4	137.3	9.2
2	**no**	2.7	2.6	5.7	1.4	13.7	1.4	22.0	1.5
	yes	3.0	3.0	10.3	2.6	23.0	2.3	36.3	2.4
3	**no**	0.0	0.0	0.0	0.0	0.3	0.0	0.3	0.0
	yes	0.3	0.3	0.0	0.0	1.0	0.1	1.3	0.1
4	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.3	0.1	0.0	0.0	0.3	0.0
8	**no**	1.3	1.3	0.0	0.0	18.3	1.8	19.7	1.3
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	total	101		400		998		1499	

Table 3.8: Blocking UDP: average frame losses for 3 Mbit/s movie

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	**no**	17.0	16.8	54.0	13.5	97.0	9.7	168.0	11.2
	yes	15.7	15.5	56.7	14.2	119.7	12.0	192.0	12.8
2	**no**	0.3	0.3	5.0	1.2	10.7	1.1	16.0	1.1
	yes	4.7	4.6	14.0	3.5	21.3	2.1	40.0	2.7
3	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.3	0.3	0.3	0.1	0.7	0.1	1.3	0.1
4	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	total	101		400		998		1499	

Table 3.9: Blocking UDP: average frame losses for 4 Mbit/s movie

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	**no**	14.3	14.2	68.0	17.0	125.0	12.5	207.3	13.8
	yes	18.0	17.8	71.0	17.8	121.0	12.1	210.0	14.0
2	**no**	1.3	1.3	4.7	1.2	13.7	1.4	19.7	1.3
	yes	4.7	4.6	11.3	2.8	18.3	1.8	34.3	2.3
3	**no**	0.0	0.0	0.0	0.0	0.7	0.1	0.7	0.0
	yes	0.0	0.0	0.3	0.1	0.0	0.0	0.3	0.0
4	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.7	0.2	0.0	0.0	0.7	0.0
8	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	total	101		400		998		1499	

Table 3.10: Blocking UDP: average frame losses for 5 Mbit/s movie

to discard the entire frame that the lost packet was part of, we would like losses to occur in groups, so relatively fewer frames are lost. Unfortunately, this is not the case.

Figure 3.9 shows the progress of latency of MPEG-2 video frames through time of the afternoon measurement point for different retry values, a bitrate of 3 Mbit/s, without interference. Due to the internal mechanics of GStreamer, a certain offset is added to the latency. Apparently, some processing takes place before a frame's timestamp is measured. This explains why, at retry values of 2 and higher, the latencies fluctuate around different values, instead of fluctuating around zero. Note that the scale of the graph is logarithmic. We see that, at retry value 1, the latency grows rather quickly, especially in the beginning. Also, as the retry value increases, the fluctuations become smaller.

In Figure 3.10 we see the same thing, with interference. What is most notable, is how the latency at retry value 2 suddenly starts growing when the microwave is activated at the 20 s mark, and does not manage to drop back to its former level when the microwave is finished at the 40 s mark.

Figure 3.11 depicts the latency for retry value 4 without interference, at different bitrates. We see how the latency fluctuates but does not grow for bitrates 3 and 4, which lie below the maximal throughput of 4 Mbit/s, but for movie bitrates of 5 and 6 Mbit/s, which exceed the available 4 Mbit/s throughput, it does grow. In Figure 3.12, we see how the latency suddenly starts growing when the microwave is turned on, and, while decreasing again when the microwave is finished, doesn't manage to return to its former level.

Figures 3.13 and 3.14 show the relation between MPEG frame size and latency with and without interference, for retry values 1 and 4 at a 3 Mbit/s bitrate, and for bitrates of 3 and 6 Mbit/s at retry value 4 respectively. We had expected to see the latency increase with frame size, since transmission of a bigger frame costs more time than transmission of a smaller one. However, no such relation is apparent.

We see in Figures 3.15 to 3.18 that, for low retry values, the jitter may rise to over 100 ms. Also, if the bitrate is high and interference is present, we see in Figure 3.18 that the jitter rises to about 70 ms. However, in most cases, jitter does not exceed 20 ms.

Conclusions

1. Although preliminary measurements with Iperf [8] showed an effective throughput of 6 Mbit/s for retry values of 3 and up, the effective throughput measured with GStreamer in these cases was merely about 4 Mbit/s. We suspect, as this was the only parameter that changed between them, this has to do with the fact that, while Iperf continuously sent UDP packets of maximum size, packet sizes for MPEG streams transmitted by GStreamer varied more or less randomly between 50 bytes and 1100 bytes. The cause for this is the packetization scheme enforced by the RFC 2250 encoder [5].
2. A stream with a bitrate of 3 Mbit/s is more resilient against interference and bad network conditions than a stream of which the bitrate is closer to (or exceeding) the maximal effective throughput.
3. Setting the retry value to 3 or higher guarantees liveness, unless the movie's bitrate exceeds the available bandwidth.

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	**no**	4.7	4.6	22.7	5.7	36.3	3.6	63.7	4.2
	yes	13.0	12.9	56.7	14.2	105.0	10.5	174.7	11.7
2	**no**	0.0	0.0	0.0	0.0	0.3	0.0	0.3	0.0
	yes	1.0	1.0	5.7	1.4	11.7	1.2	18.3	1.2
3	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	1.3	0.3	0.7	0.1	2.0	0.1
4	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.3	0.1	0.0	0.0	0.3	0.0
8	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	total	101		400		998		1499	

Table 3.11: Blocking UDP: average frame losses for 6 Mbit/s movie

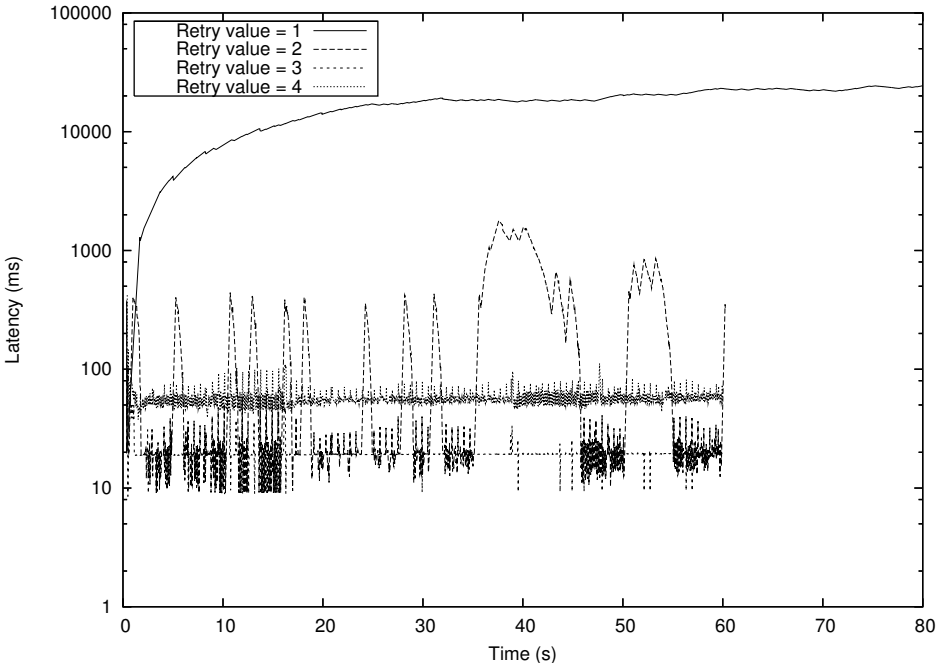


Figure 3.9: Latency. Protocol = blocking UDP, bitrate = 3 Mbit/s, microwave = no, time = afternoon. Logarithmic scale.

retry value	bit rate	micro wave	1	2	3	4	5-10	10+	lost pkts	loss seq's
1	**3**	**no**	98.9	1.1	0.0	0.0	0.0	0.0	454	449
		yes	97.1	2.6	0.0	0.2	0.2	0.0	529	509
	4	**no**	98.6	1.4	0.0	0.0	0.0	0.0	598	590
		yes	98.6	1.2	0.0	0.0	0.1	0.0	751	738
	5	**no**	99.7	0.3	0.0	0.0	0.0	0.0	741	739
		yes	98.1	1.7	0.1	0.0	0.1	0.0	852	832
	6	**no**	99.5	0.5	0.0	0.0	0.0	0.0	200	199
		yes	97.1	1.9	0.8	0.0	0.3	0.0	393	376
2	**3**	**no**	97.0	3.0	0.0	0.0	0.0	0.0	68	66
		yes	98.4	1.6	0.0	0.0	0.0	0.0	128	126
	4	**no**	100.0	0.0	0.0	0.0	0.0	0.0	49	49
		yes	99.3	0.0	0.7	0.0	0.0	0.0	138	136
	5	**no**	100.0	0.0	0.0	0.0	0.0	0.0	61	61
		yes	98.3	1.7	0.0	0.0	0.0	0.0	119	117
	6	**no**	100.0	0.0	0.0	0.0	0.0	0.0	1	1
		yes	98.6	1.4	0.0	0.0	0.0	0.0	70	69
3	**3**	**no**	0.0	100.0	0.0	0.0	0.0	0.0	2	1
		yes	100.0	0.0	0.0	0.0	0.0	0.0	4	4
	4	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		yes	100.0	0.0	0.0	0.0	0.0	0.0	4	4
	5	**no**	100.0	0.0	0.0	0.0	0.0	0.0	2	2
		yes	100.0	0.0	0.0	0.0	0.0	0.0	1	1
	6	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		yes	100.0	0.0	0.0	0.0	0.0	0.0	6	6
4	**3**	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		yes	100.0	0.0	0.0	0.0	0.0	0.0	1	1
	4	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		yes	0.0	0.0	0.0	0.0	0.0	0.0	0	0
	5	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		yes	100.0	0.0	0.0	0.0	0.0	0.0	2	2
	6	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		yes	100.0	0.0	0.0	0.0	0.0	0.0	1	1

Table 3.12: Blocking UDP: Probability (in %) of consecutive packet loss

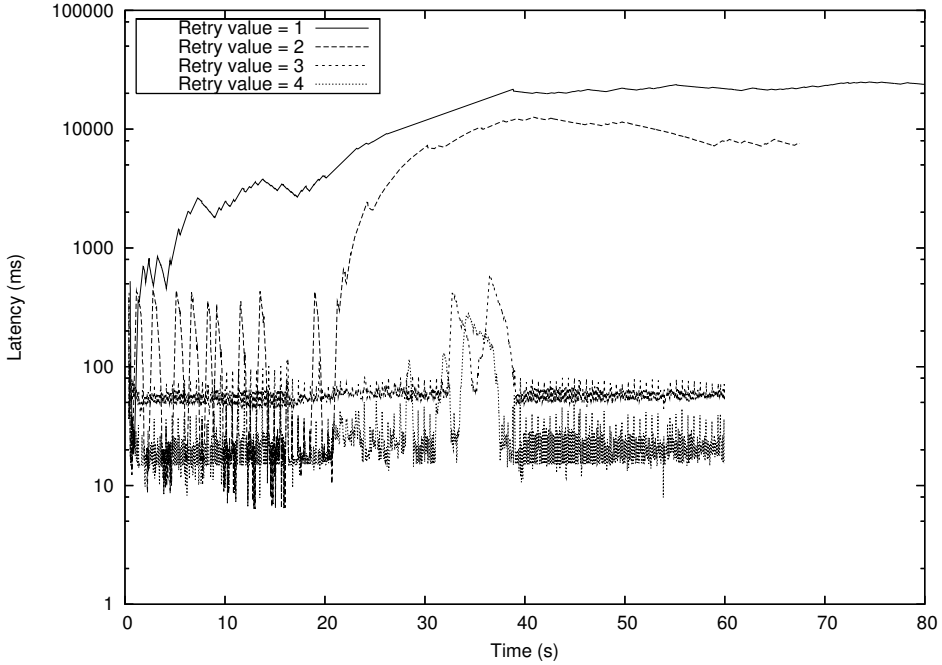


Figure 3.10: Latency. Protocol = blocking UDP, bitrate = 3 Mbit/s, microwave = yes, time = afternoon. Logarithmic scale.

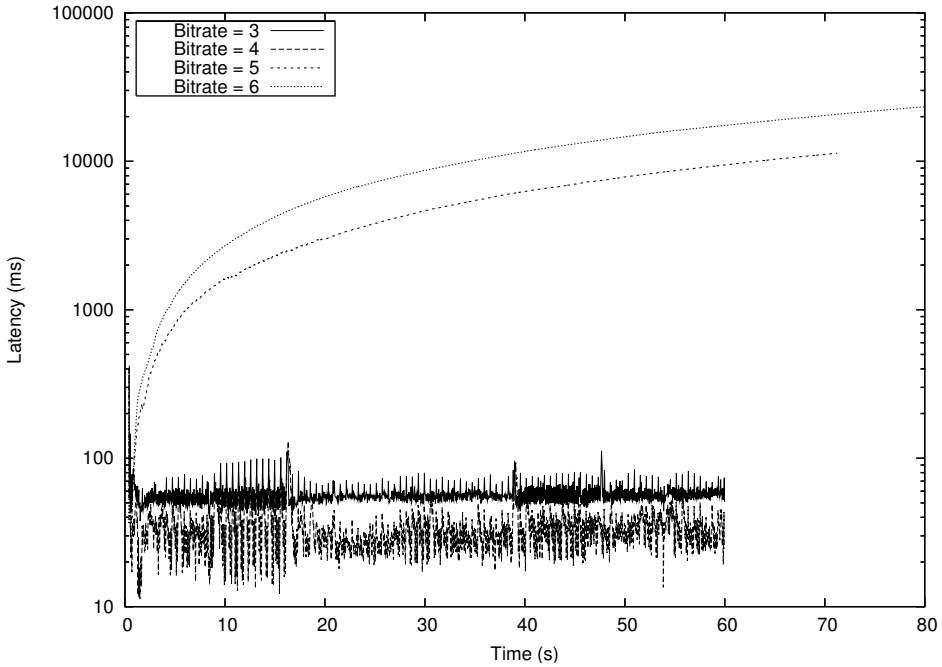


Figure 3.11: Latency. Protocol = blocking UDP, retry value = 4, microwave = no, time = afternoon. Logarithmic scale.

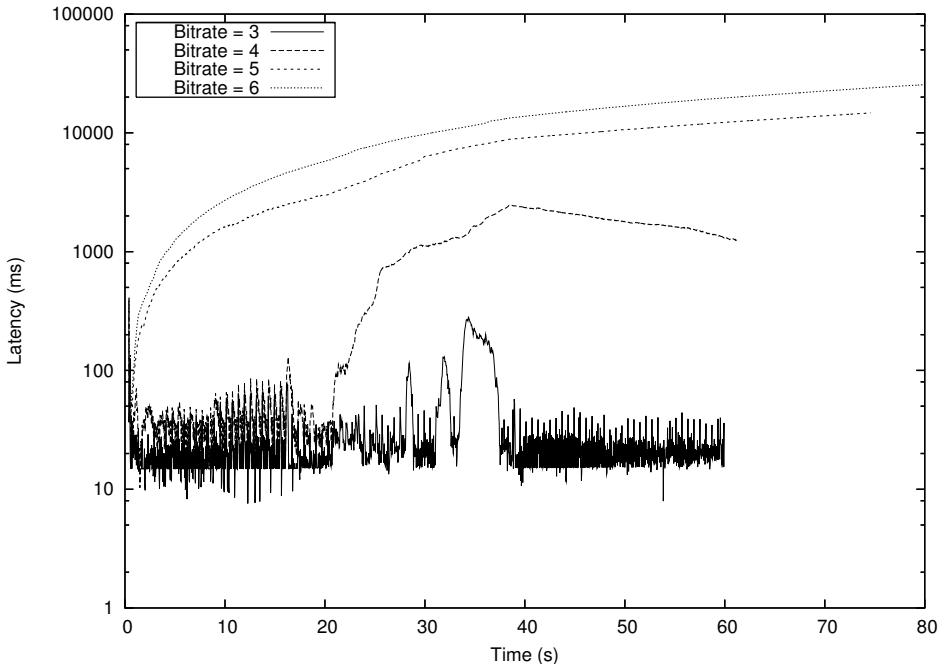


Figure 3.12: Latency. Protocol = blocking UDP, retry value = 4, microwave = yes, time = afternoon. Logarithmic scale.

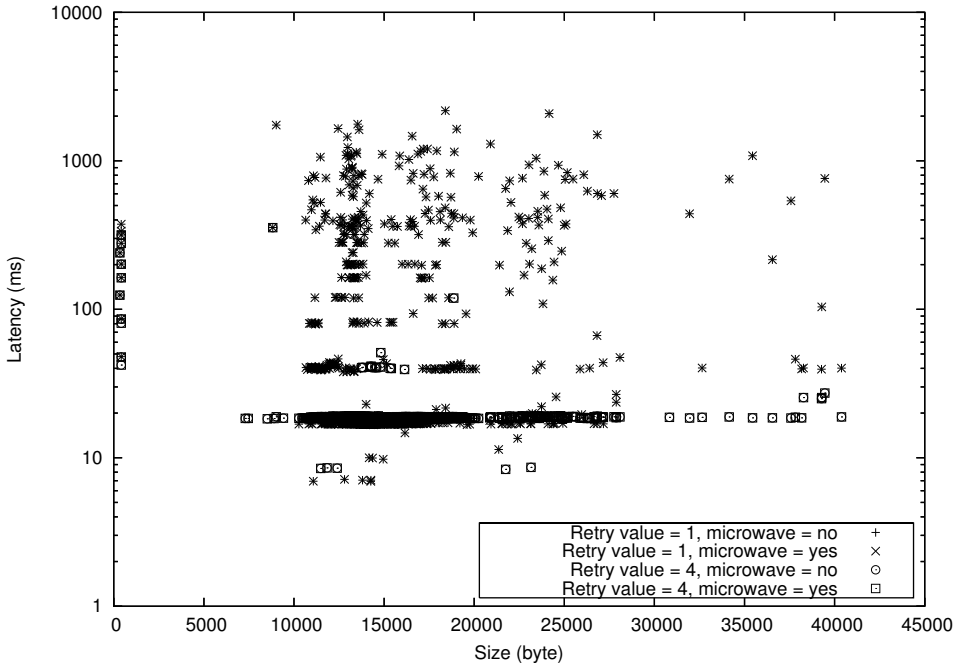


Figure 3.13: Size / latency. Protocol = blocking UDP, bitrate = 3 Mbit/s, time = afternoon.

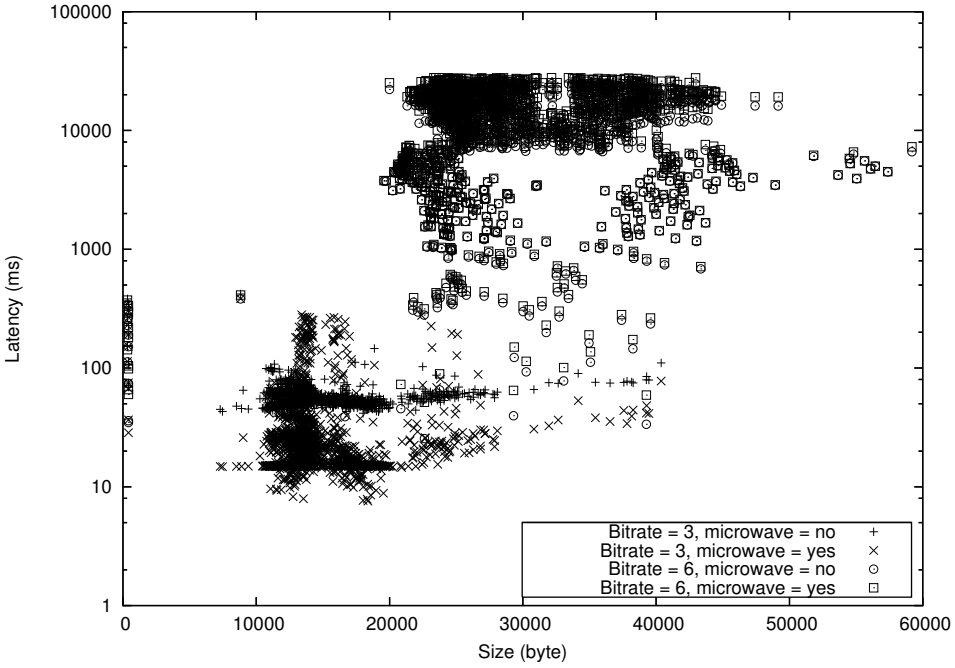


Figure 3.14: Size / latency. Protocol = blocking UDP, retry value = 4, time = afternoon.

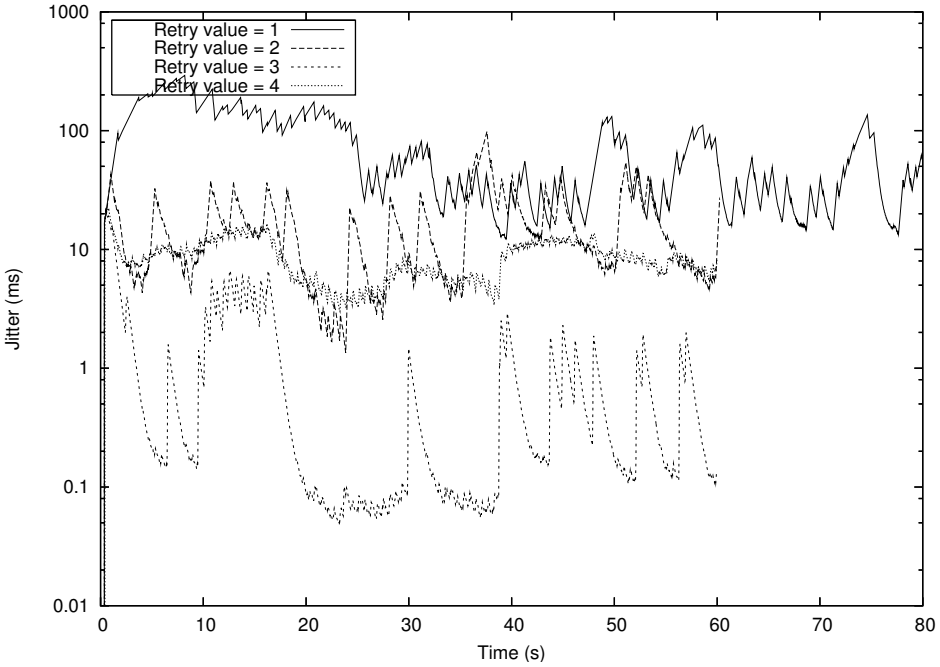


Figure 3.15: Jitter. Protocol = blocking UDP, bitrate = 3 Mbit/s, microwave = no, time = afternoon. Logarithmic scale.

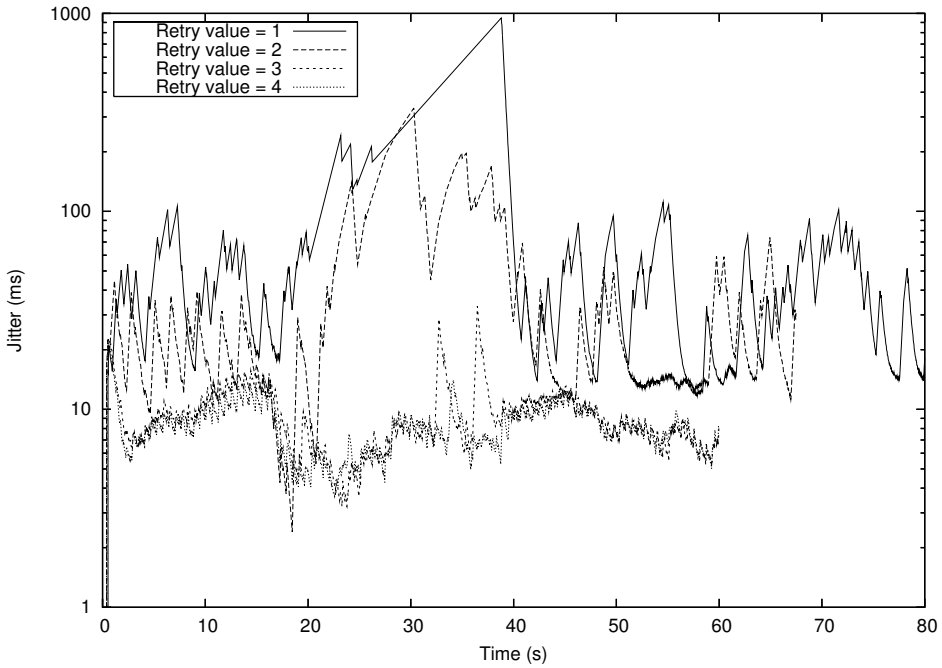


Figure 3.16: Jitter. Protocol = blocking UDP, bitrate = 3 Mbit/s, microwave = yes, time = afternoon. Logarithmic scale.

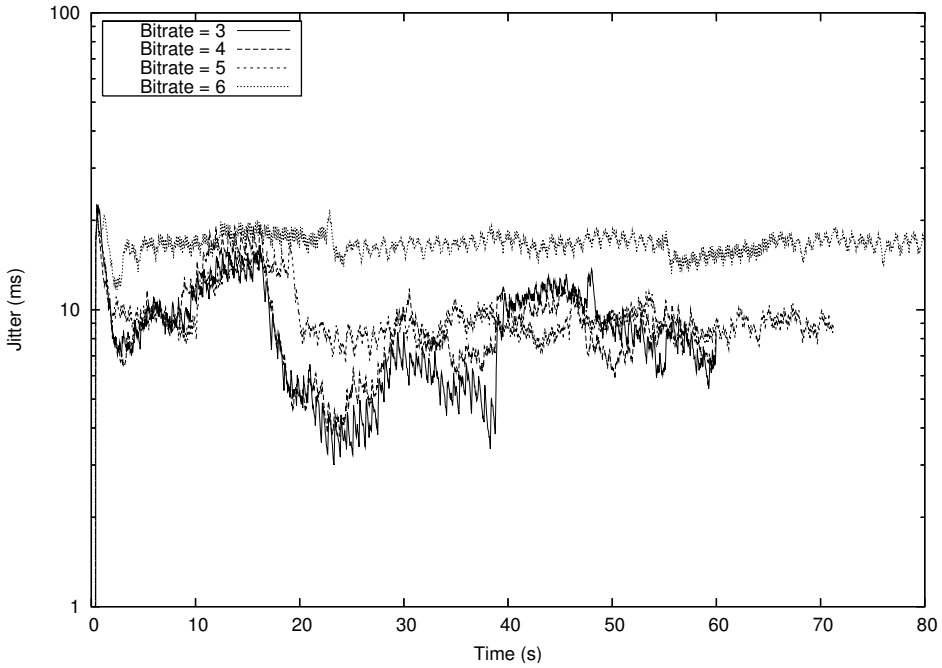


Figure 3.17: Jitter. Protocol = blocking UDP, retry value = 4, microwave = no, time = afternoon. Logarithmic scale.

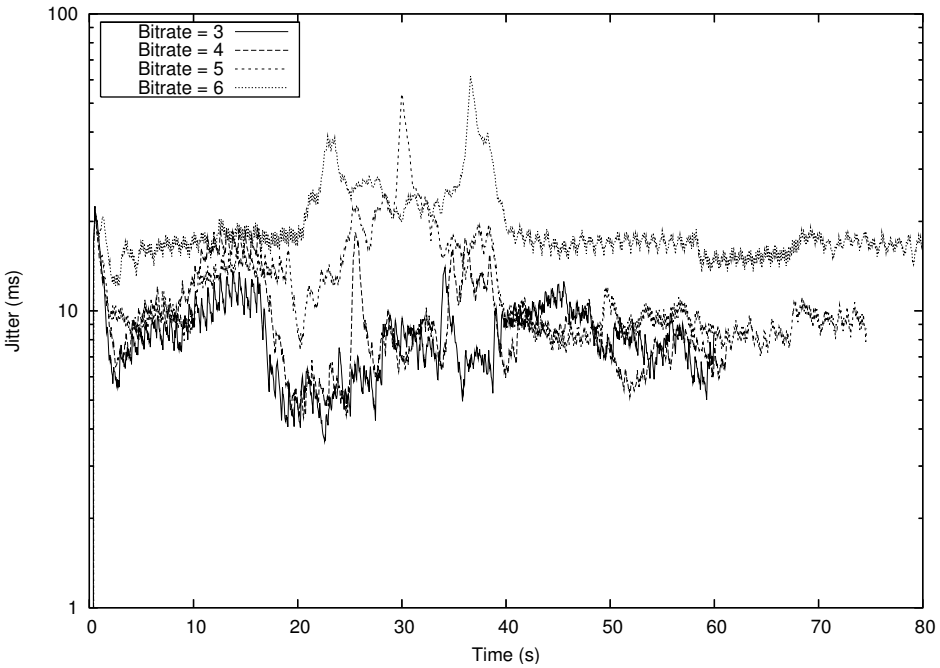


Figure 3.18: Jitter. Protocol = blocking UDP, retry value = 4, microwave = yes, time = afternoon. Logarithmic scale.

4. A high bitrate results in a higher duration, as expected. A low retry value (1 or 2) results in less throughput, which was expected, and longer stream duration, which was unexpected, as the losses caused by the lower retry value should not influence the duration of the stream as a whole. Interference also causes the duration and the losses to increase.
5. As expected, I-frames are lost relatively more often than P-frames, and P-frames more than B-frames.
6. It is a pity that the loss of a single UDP packet causes the loss of an entire MPEG frame. If we compare Table 3.3 with Tables 3.8 through 3.11, we see that losses on the UDP level are much lower than on the MPEG level.
7. UDP packet loss on the air occurs mostly one packet at a time, and not in batches. This has a big impact on frame loss, as one lost packet causes an entire frame to be discarded.
8. The size of an MPEG frame has no influence on its latency. The conjecture that larger size causes higher latency is therefore not correct.
9. For MPEG movies of 25 frames per second, one frame needs to be displayed every 40 ms. Under favourable network conditions, jitter does not exceed 20 ms. This means that only one frame needs to be buffered in order to eliminate any problems caused by jitter under favourable network conditions. If conditions are bad, one or two more may be needed.

3.7 TCP

Set-up

The sender and receiver were running the following GStreamer pipelines:

Sender: filesrc – tcpsink
Receiver: tcpsrc – mpegstat – fakesink

The sender's pipeline is a typical pipeline for TCP streaming, as seen in Section 1.6. The receiver has the regular MPEG-2 decoder and the screen sink replaced by our mpegstat filter, which generates the data we need for the analysis, followed by the fakesink which simply discards the data.

For TCP connections in Linux, the sender's socket buffer is blocking and non-lossy. We don't intentionally drop data in the GStreamer pipeline. Losses may occur on the air; however, TCP's retransmission scheme makes sure no data is lost.

Because the sender's socket buffer may block, and because of TCP's retransmission mechanism, delays may occur, which means the stream will not arrive in real-time. Therefore, a stream using TCP as transport protocol is not a live stream.

Results

Figure 3.19 shows the throughput that was measured for TCP while streaming a 6 Mbit/s movie without interference. The first thing we see is the poor throughput for retry value 1. At retry value 2, the throughput has a number of dips as well, but most of the time it keeps up with the throughputs at retry values 3 and 4, which lies just above 5 Mbit/s. Since the movie bitrate in these streams was 6 Mbit/s, in all cases the stream duration exceeds the 60 s the movie needs in real-time.

In Figure 3.20, we see what happens when interference is added, from the 20 s mark to the 40 s mark. The stream with retry value 1 comes to an almost complete halt. The other streams are affected too; generally we can say that the higher the retry value, the less it is affected by the interference.

Figure 3.21 shows the effect of the movie bitrate on the throughput of the stream. We see that the 3 Mbit/s movie does not use more than a 3 Mbit/s throughput. The same holds for 4 and 5 Mbit/s. They all end at the 60 s mark. The 6 Mbit/s movie needs more time to finish as it cannot be streamed at 6 Mbit/s.

The effect of interference can be seen in Figure 3.22. Except for a few spikes, the 3 and 4 Mbit/s movies are not affected, as opposed to the 5 and 6 Mbit/s movies. Apparently, a stream is more resilient against interference if its bitrate is smaller than the available bandwidth; it has some leeway.

The duratation of the stream under the different circumstances can be seen in Table 3.13. Clearly visible is the duration of the stream for a retry value of 1, which is disproportionate with respect to the durations for higher retry values. For retry values of 3 and up, the durations do not change that much anymore, as can most clearly be seen in the 5 and 6 Mbit/s columns. We also see that for bitrates 3 and 4, and for 5 when no interference is present, the duration is always 59.9 s. This is due to the fact that GStreamer does not display the movies

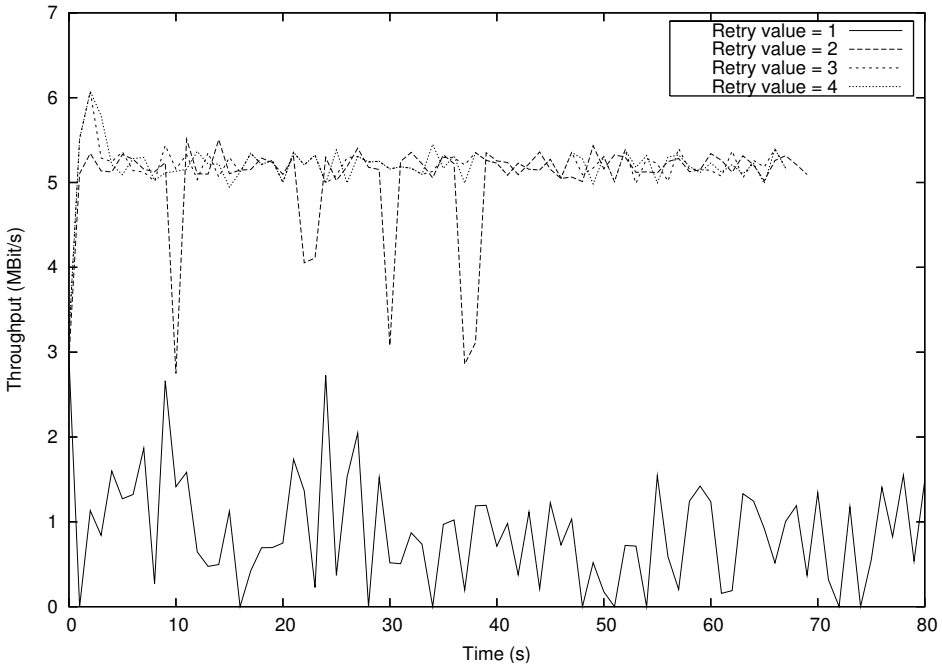


Figure 3.19: Throughput. Protocol = TCP, bitrate = 6 Mbit/s, microwave = no, time = afternoon.

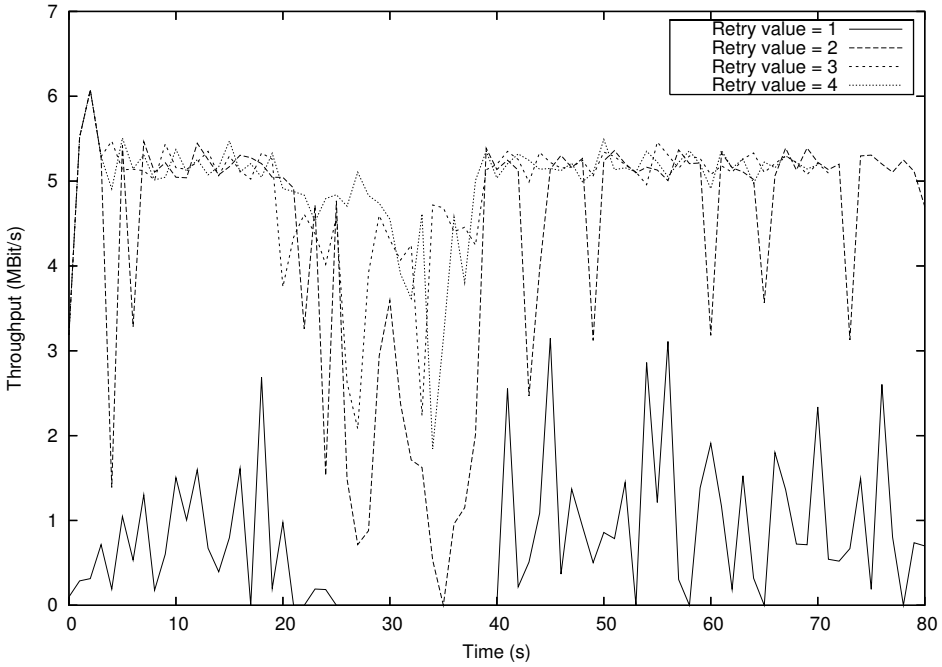


Figure 3.20: Throughput. Protocol = TCP, bitrate = 6 Mbit/s, microwave = yes, time = afternoon.

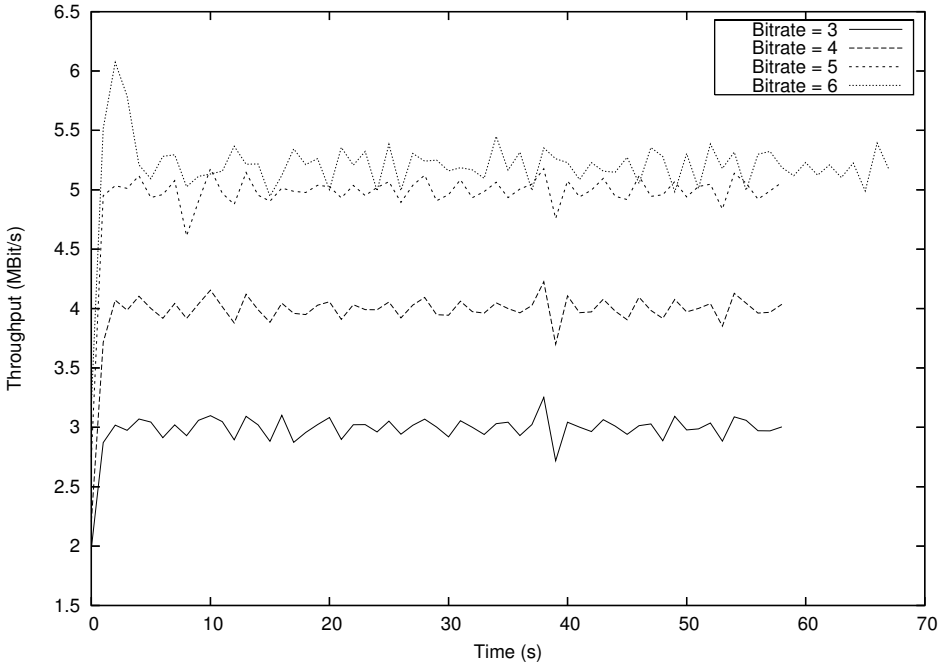


Figure 3.21: Throughput. Protocol = TCP, retry value = 4, microwave = no, time = afternoon.

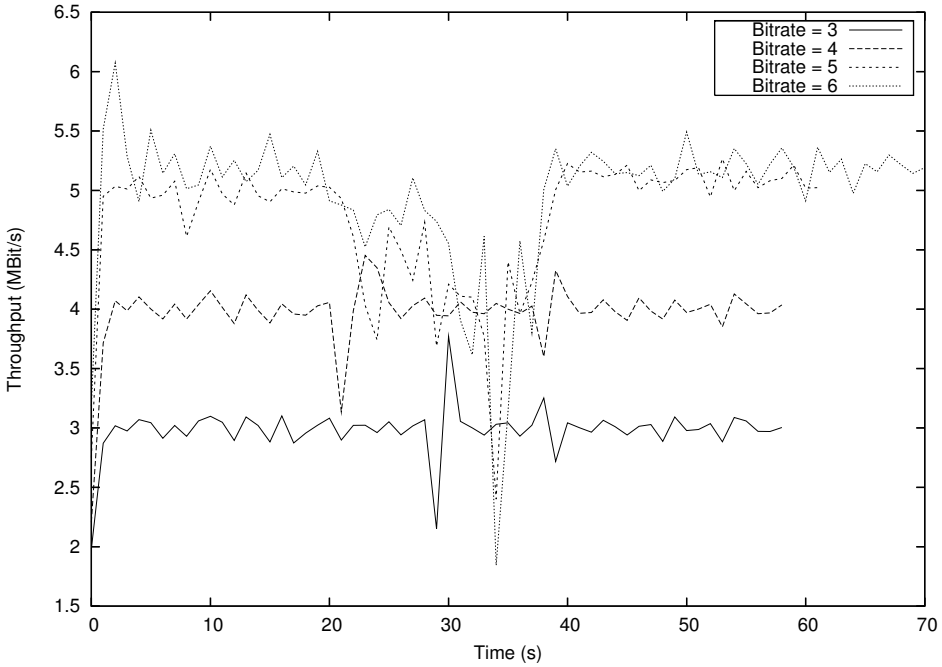


Figure 3.22: Throughput. Protocol = TCP, retry value = 4, microwave = yes, time = afternoon.

quicker than real-time, even if it had the chance. The movie will not be played faster than real time, which is as expected.

The fact that the duration for a 6 Mbit/s movie at retry value 1 is smaller than that for a 5 Mbit/s movie probably has to do with external factors such as background interference (not caused by our microwave), TCP's congestion control algorithms and the ratio of data packets and acknowledgment packets that happen to be lost on the link.

Figure 3.23 shows the latency of a 3 Mbit/s movie stream without interference, for retry values 1 through 4. At retry value 1, we see the latency growing steeply. Note that GStreamer's internal mechanics add a certain offset to the latency. In the other cases, however, the latency remains quite constant, with a few spikes when the retry value is 2. In Figure 3.24, we see the effect of adding interference to the same circumstances. This is especially clear for retry value 2, where latency suddenly rises when the microwave is activated. Interestingly, at the 40 s mark, when the microwave is finished, the latency starts dropping again as the stream catches up again, sending the frames that were late at a quicker pace. Finally, the stream is finished at the 60 s mark, despite the interference.

Figure 3.25 shows the latency without interference at retry value 4 for bitrates of 3 through 6 Mbit/s. We see that, while remaining more or less contstant for bitrates of 3 through 5 Mbit/s, the latency rises steeply for a bitrate of 6 Mbit/s. This reflects the fact that a 6 Mbit/s movie cannot be streamed in real time, since the maximal available bandwidth is a little over 5 Mbit/s. If we add interference, streaming a 5 Mbit/s movie becomes problematic as well, as seen in Figure 3.26: the latency grows rapidly while the microwave is on, and is unable to recover before the movie ends. Again, the movies with lower bitrates have more leeway which they can use to overcome the drop in bandwidth caused by the interference.

In Figures 3.27 and 3.28 we see how the size of an MPEG frame relates to its latency. Figure 3.27 shows this for a movie bitrate of 3 Mbit/s and retry values 1 and 4, with and without interference; Figure 3.28 for a retry value of 4 and bitrates 3 and 6, with and without interference. We see that the expected correlation between frame size and latency is not present.

For retry values of 3 and higher, the measured jitter is less than a microsecond, as can be seen in Figure 3.29. For a retry value of 2, we get some spikes that reach up to about 20 ms. Only for retry value 1, we get high jitter values. The rise in jitter in the 5 Mbit/s stream is possibly due to some external interference that occurred while performing the measurements. It did not occur in the morning and midday measurements.

If we add interference, we see in Figure 3.30 that for retry values 3 and 4, some spikes occur to up to 20 ms. For retry value 2, jitter rises to almost the same level that we see for retry value 1 with no interference.

If we look at Figure 3.31, we see the influence of bitrate on the jitter. For bitrates of 3 through 5 Mbit/s, jitter is low, less than half a millisecond. For a bitrate of 6 Mbit/s, the jitter rises to about 20 ms. With interference, we see in Figure 3.32 that the jitter is more erratic, with higher peaks. For bitrate 5 Mbit/s, the higher jitter persists, even when the microwave is inactive.

retry value	micro wave	bitrate 3	4	5	6
0	**no**	59.9	59.9	59.9	68.8
	yes	59.9	59.9	60.3	71.4
1	**no**	221.3	320.6	428.3	363.7
	yes	257.4	328.4	420.8	394.5
2	**no**	60.0	60.6	68.8	70.9
	yes	59.9	65.6	75.7	83.3
3	**no**	59.9	59.9	59.9	68.7
	yes	59.9	59.9	61.5	72.1
4	**no**	59.9	59.9	59.9	69.2
	yes	59.9	59.9	60.8	72.2
8	**no**	59.9	59.9	59.9	69.0
	yes	59.9	59.9	61.3	71.7
16	**no**	59.9	59.9	59.9	68.9
	yes	59.9	59.9	61.0	72.1

Table 3.13: Average movie duration in seconds for TCP

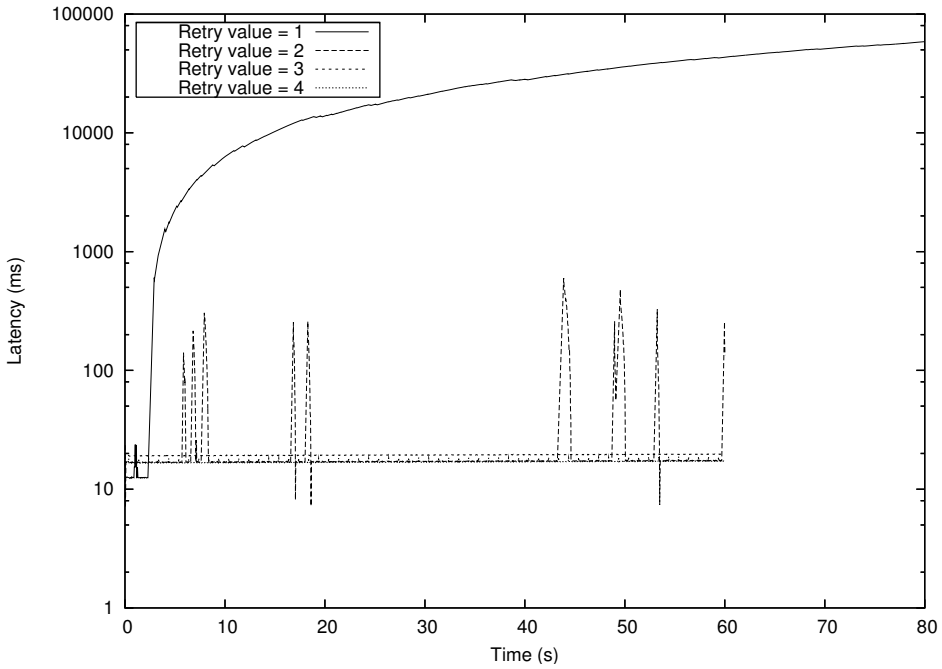


Figure 3.23: Latency. Protocol = TCP, bitrate = 3 Mbit/s, microwave = no, time = afternoon. Logarithmic scale.

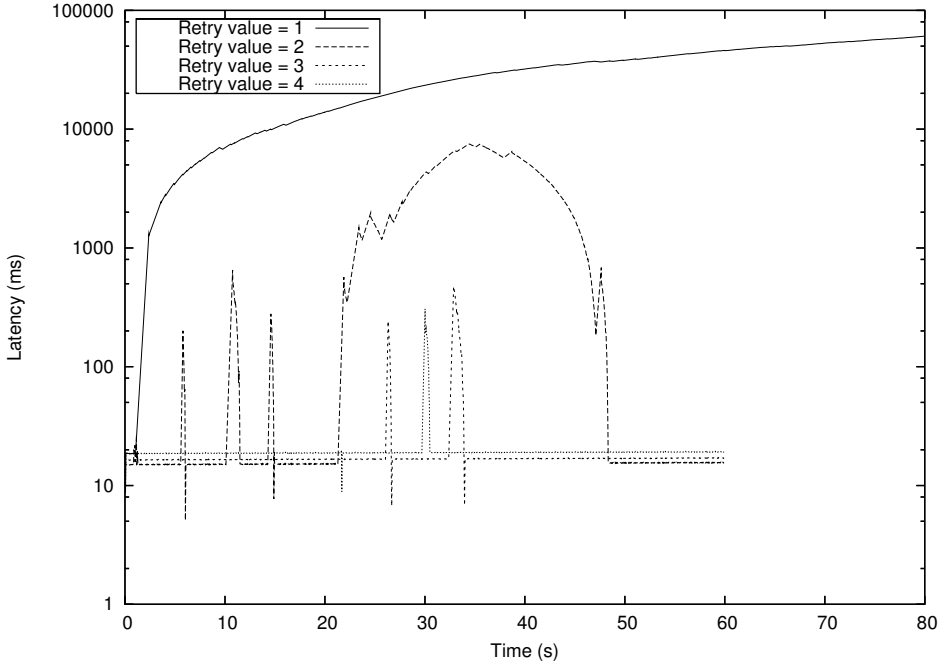


Figure 3.24: Latency. Protocol = TCP, bitrate = 3 Mbit/s, microwave = yes, time = afternoon. Logarithmic scale.

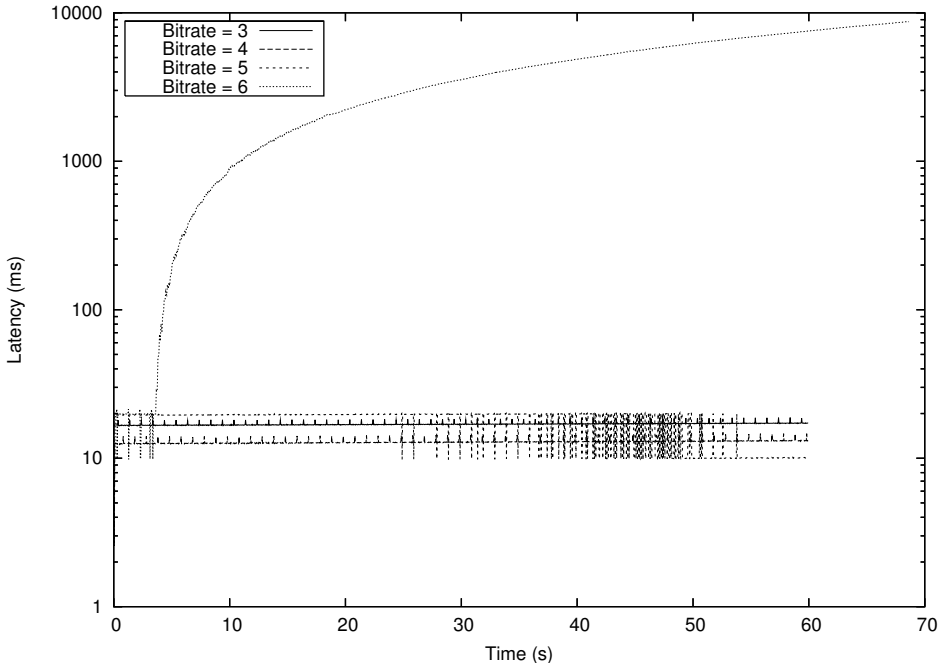


Figure 3.25: Latency. Protocol = TCP, retry value = 4, microwave = no, time = afternoon. Logarithmic scale.

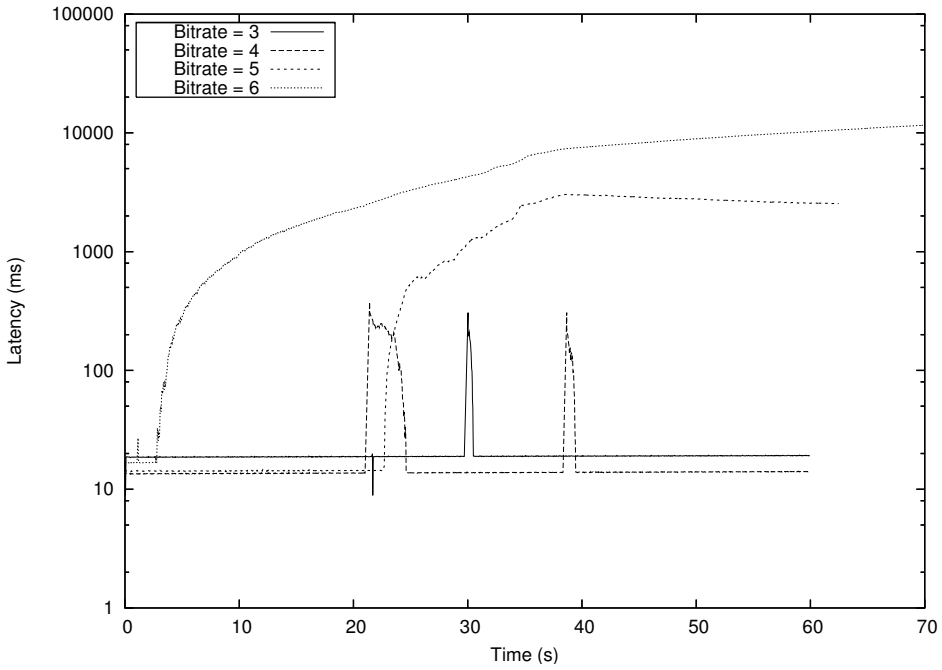


Figure 3.26: Latency. Protocol = TCP, retry value = 4, microwave = yes, time = afternoon. Logarithmic scale.

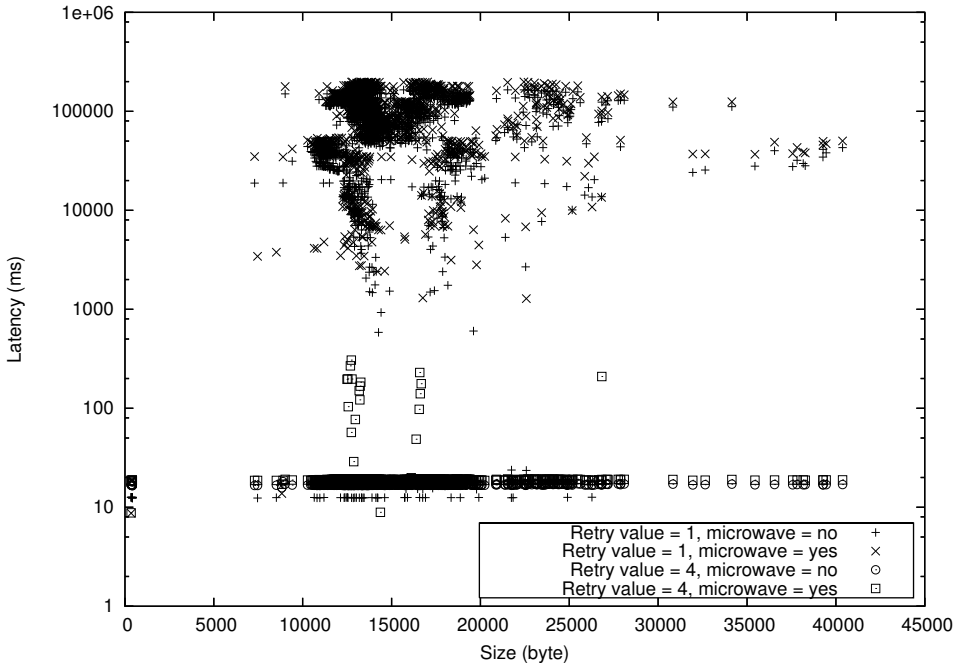


Figure 3.27: Size / latency. Protocol = TCP, bitrate = 3 Mbit/s, time = afternoon.

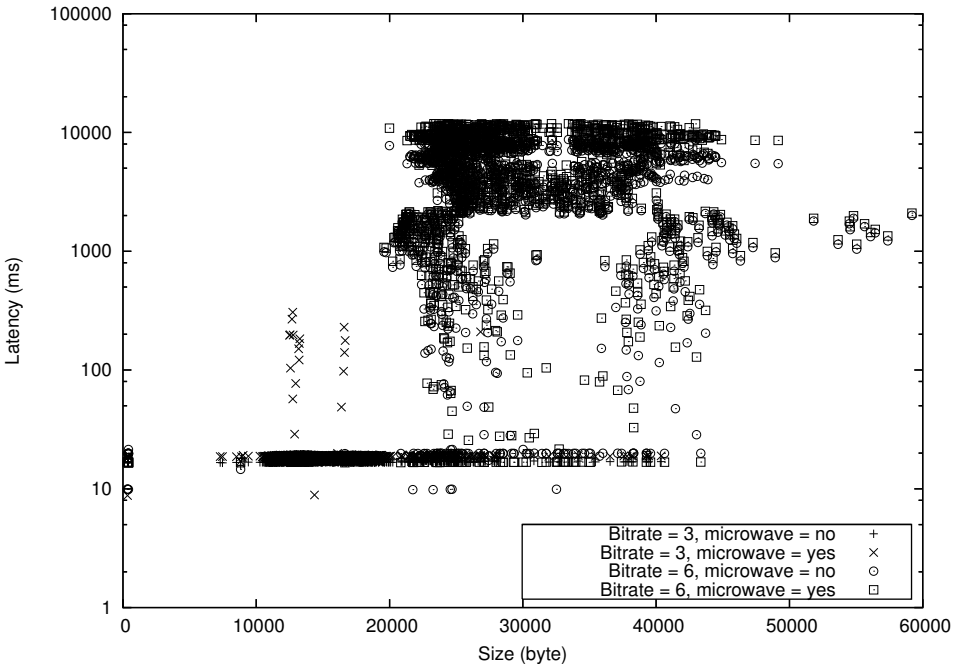


Figure 3.28: Size / latency. Protocol = TCP, retry value = 4, time = afternoon.

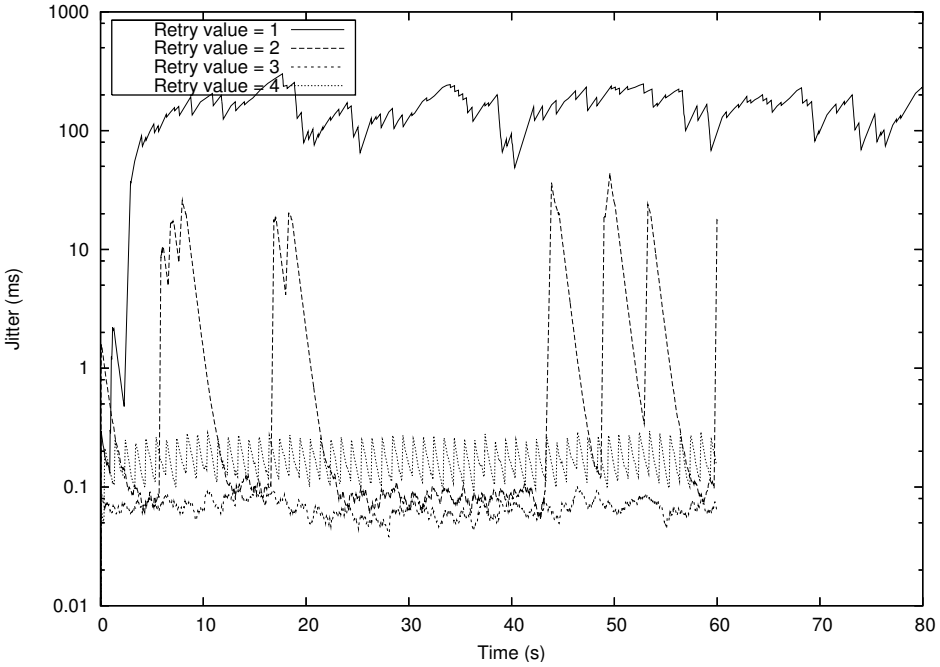


Figure 3.29: Jitter. Protocol = TCP, bitrate = 3 Mbit/s, microwave = no, time = afternoon. Logarithmic scale.

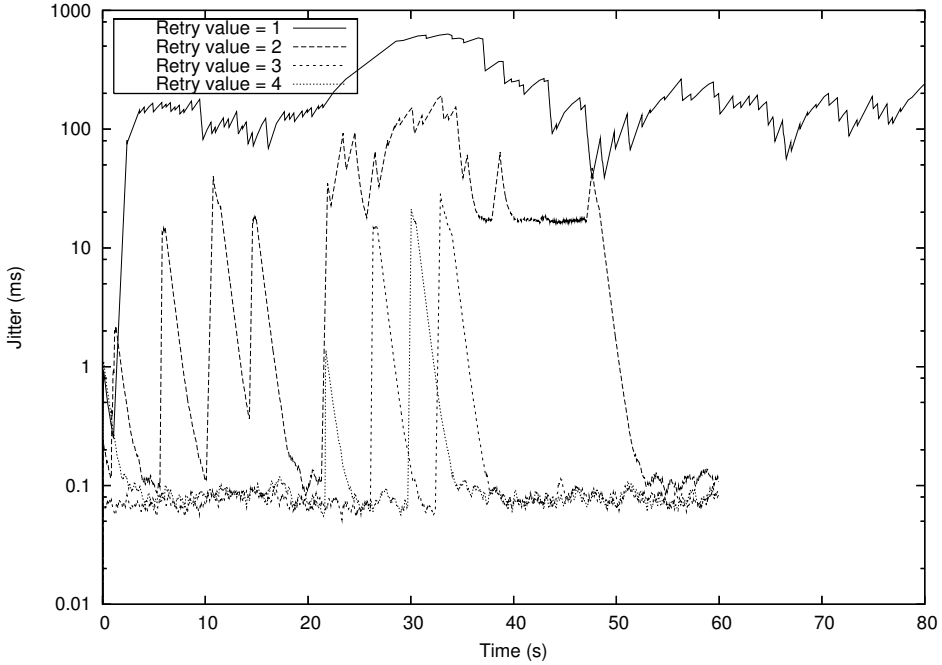


Figure 3.30: Jitter. Protocol = TCP, bitrate = 3 Mbit/s, microwave = yes, time = afternoon. Logarithmic scale.

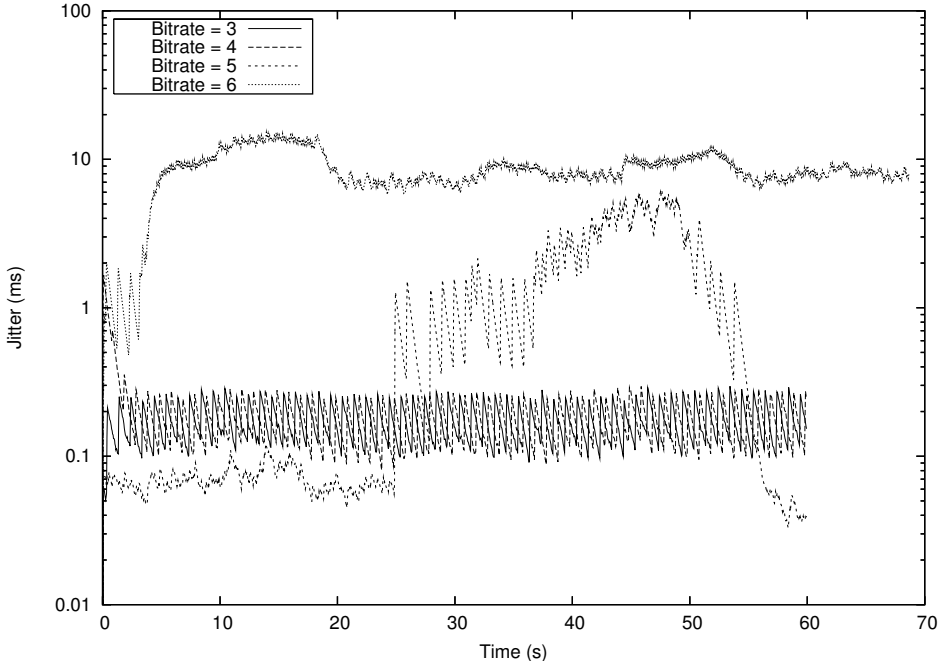


Figure 3.31: Jitter. Protocol = TCP, retry value = 4, microwave = no, time = afternoon. Logarithmic scale.

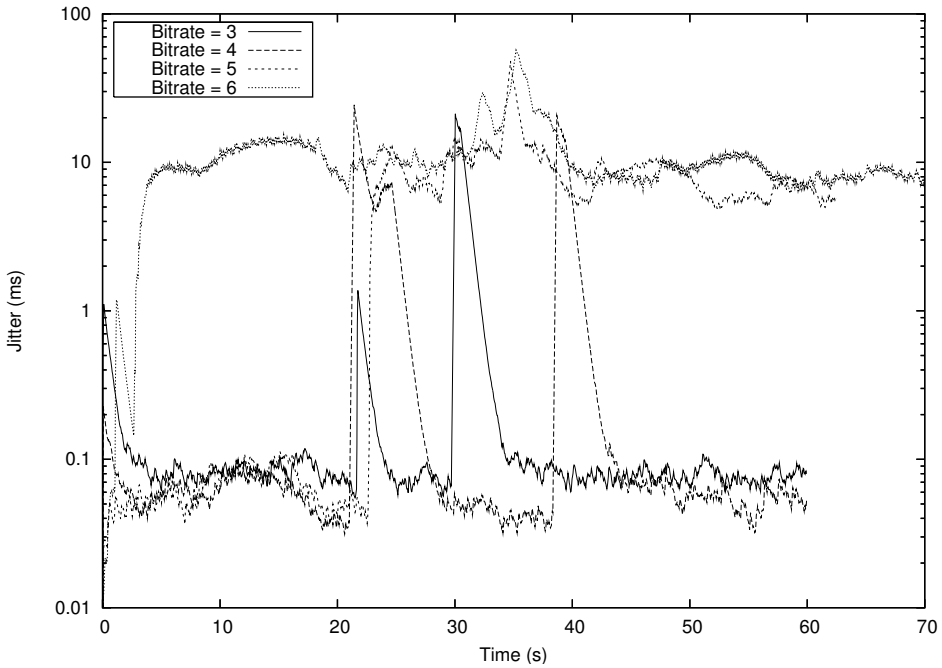


Figure 3.32: Jitter. Protocol = TCP, retry value = 4, microwave = yes, time = afternoon. Logarithmic scale.

Conclusions

1. The maximum effective throughput was slightly over 5 Mbit/s when no interference was present and retry value was 3 or higher.
2. A high bitrate results in a higher duration, as expected. A low retry value results in less throughput and longer stream duration, again as expected.
3. A stream with a bitrate of 3 or 4 Mbit/s is more resilient against interference and bad network conditions than a stream of which the bitrate is closer to (or exceeding) the maximal effective throughput.
4. At a retry value of 1, the movie needs more time to be transmitted than at a higher retry value. Also, with interference, more time is needed for transmission and more frames are lost than when no interference is present. This is due to the transmission rate, which decreases when interference is present or the retry value is 1.
5. The size of an MPEG frame has no influence on its latency. The conjecture that larger size causes higher latency is therefore not correct.
6. For MPEG movies of 25 frames per second, one frame needs to be displayed every 40 ms. Only in rare cases, when the retry value is low and interference is present, the jitter exceeds 10 ms. This means that a buffer of 1 MPEG frame suffices to eliminate most problems caused by jitter.

3.8 Non-blocking UDP

Set-up

The sender and receiver were running the following GStreamer pipelines:

Sender: filesrc – rfc2250enc – rtpmpegenc – udpsink
Receiver: udpsrc – rtpmpegparse – mpegstat – fakesink

The sender's pipeline is a typical pipeline for UDP streaming, as seen in Section 1.6. The receiver has the regular MPEG-2 decoder and the screen sink replaced by our mpegstat filter, which generates the data we need for the analysis, followed by the fakesink which simply discards the data.

Linux's default settings were maintained. These settings allow for socket buffer overflow at the sender. If the socket buffer overflows, the overflowing data is lost. Therefore, losses may occur in the sender's socket buffer. Also, data may be lost on the air. Data may also be dropped in the receiving GStreamer pipeline, if a frame is received partally, in which case the entire frame is discarded.

Since the socket buffer drops packets rather than delay the transmission, the stream is always transmitted in real time. Hence, a stream using non-blocking UDP as transport protocol is a live stream.

Results

The effective throughput is about 4 Mbit/s for retry values 2 and up, as seen in Figure 3.33, which shows the throughput for a 4 Mbit/s stream. For retry value 1, the throughput is very erratic, with occasional peaks of up to just over 4 Mbit/s. If we add interference, as seen in Figure 3.34 between 20 and 40 s, throughput drops substantially. In case of retry value 1, it even drops to zero. We have chosen to use the streams with a movie bitrate of 4 Mbit/s, instead of 6 Mbit/s which we chose for the other protocols, because at 5 and 6 Mbit/s, a phenomenon occurs that makes the data for these streams unusable. This phenomenon can be seen in Figures 3.35 and 3.36. At retry value 4, the 3 and 4 Mbit/s streams behave as expected; they are streamed at 3 and 4 Mbit/s, respectively, with a dip due to interference for the 4 Mbit/s stream. (The 3 Mbit/s stream is unaffected, as it is farther from the maximal available bandwwidth and has more leeway. Also, when the interference is present, the 4 Mbit/s stream will be influenced by the effect we describe below as well.) We would expect that the average throughput for the 5 and 6 Mbit/s streams fluctuate around the maximum available bandwidth, which we have determined at just over 4 Mbit/s for UDP streams in Section 3.5. However, the 5 Mbit/s stream fluctuates around 1.5 Mbit/s effective throughput, and for the 6 Mbit/s, the throughput is even worse. In order to understand what happens here, Figures 3.37 and 3.38 show, for a 5 Mbit/s stream and a 6 Mbit/s stream respectively, with and without microwave interference, the throughput measured at the MPEG level and the throughput measured at the UDP level. We see that, while the throughput at the MPEG level is very low, the throughput at the UDP level is much higher, and in fact at the expected level, with a dip when interference is present. We conclude that the MPEG decoder discards a lot of data, which means that a lot of MPEG frames arrive only partially. There are two reasons why a frame might arrive partially: loss on the link, and loss in the send buffer. While

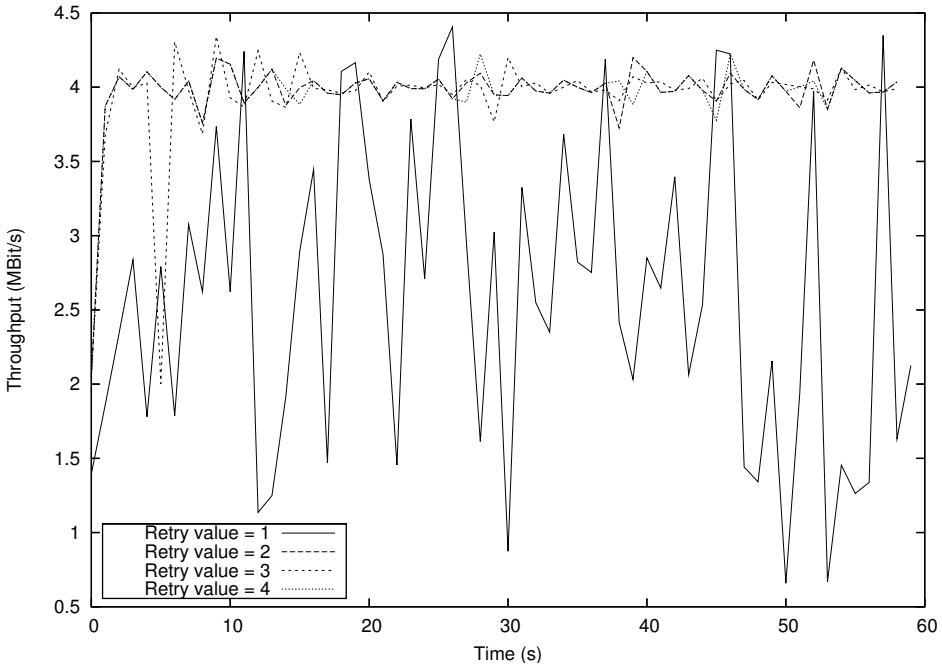


Figure 3.33: Throughput. Protocol = non-blocking UDP, bitrate = 4 Mbit/s, microwave = no, time = afternoon.

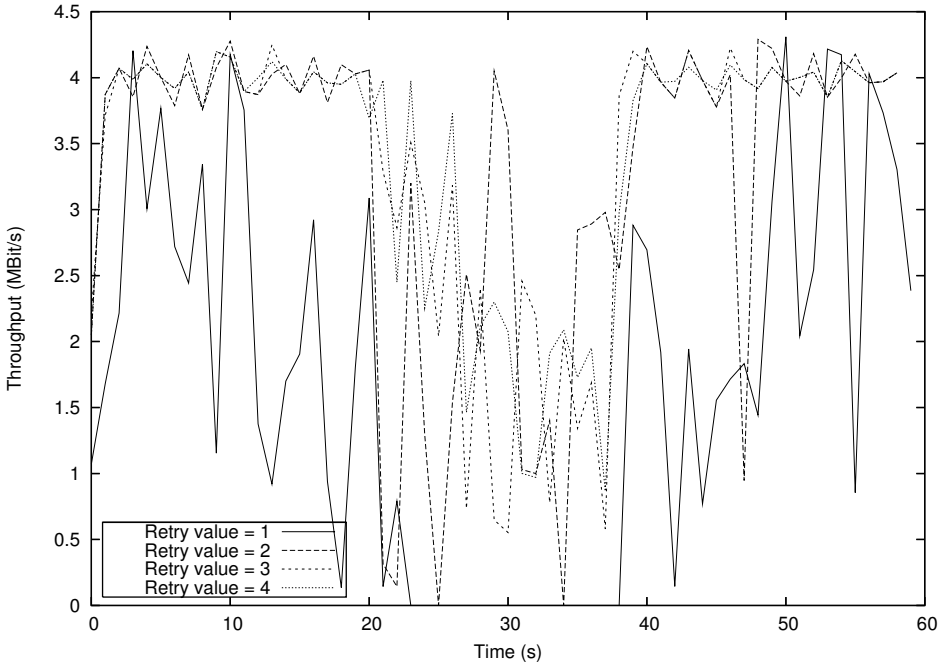


Figure 3.34: Throughput. Protocol = non-blocking UDP, bitrate = 4 Mbit/s, microwave = yes, time = afternoon.

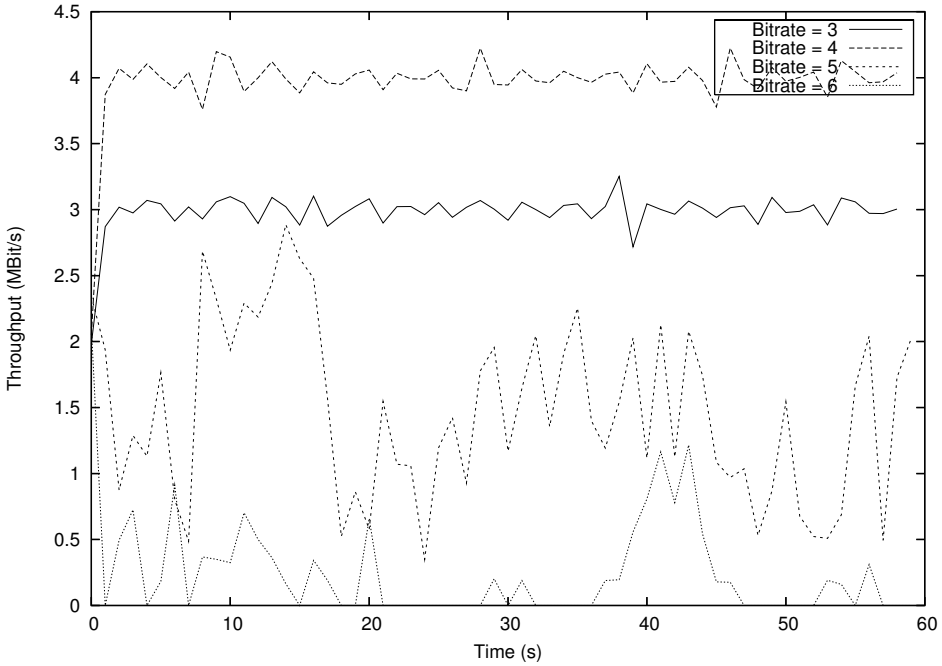


Figure 3.35: Throughput. Protocol = non-blocking UDP, retry value = 4, microwave = no, time = afternoon.

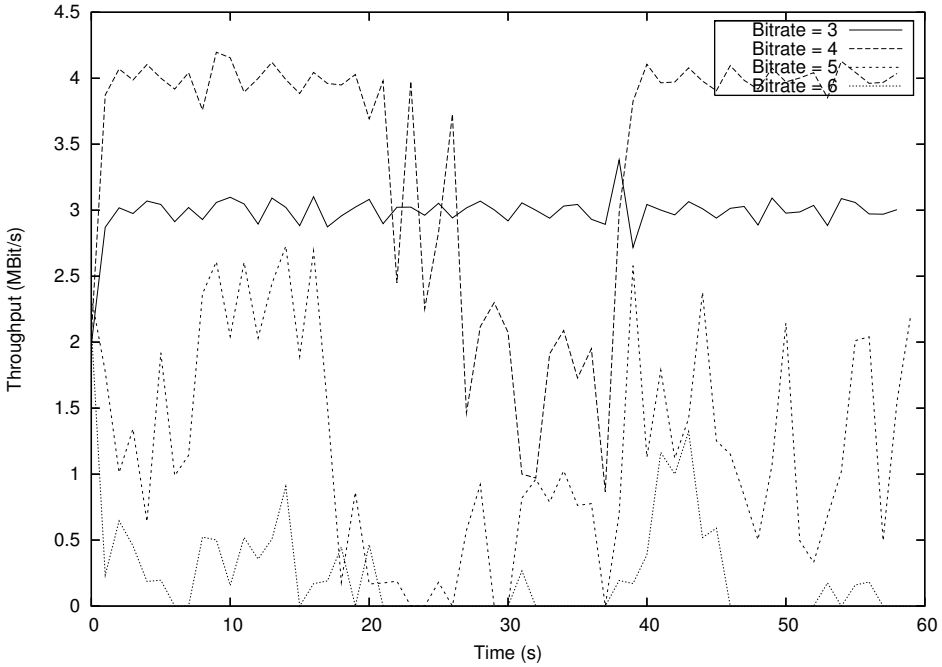


Figure 3.36: Throughput. Protocol = non-blocking UDP, retry value = 4, microwave = yes, time = afternoon.

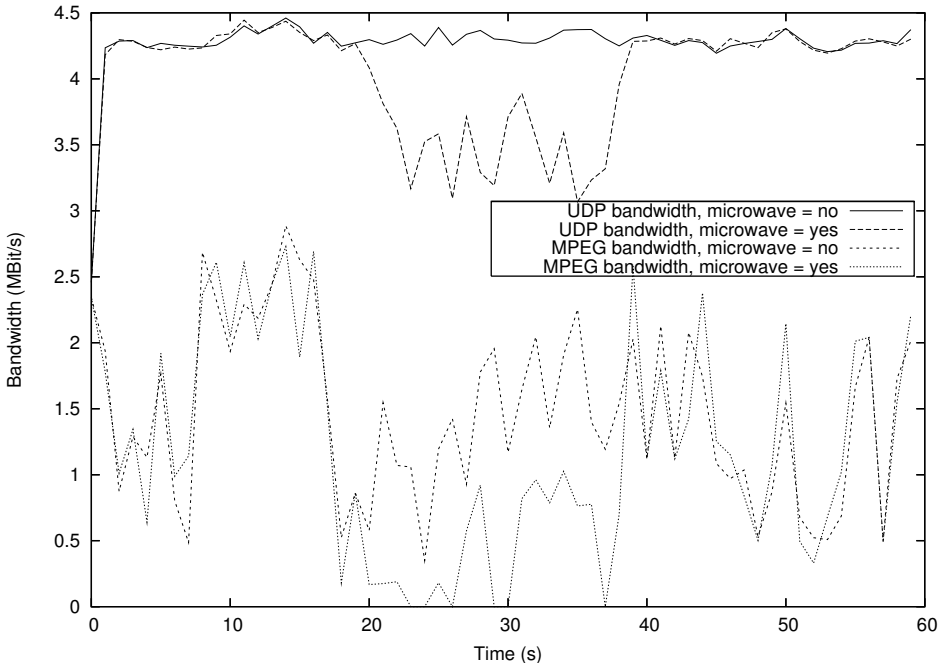


Figure 3.37: Throughput. Protocol = non-blocking UDP, retry value = 4, bitrate = 5 Mbit/s, time = afternoon.

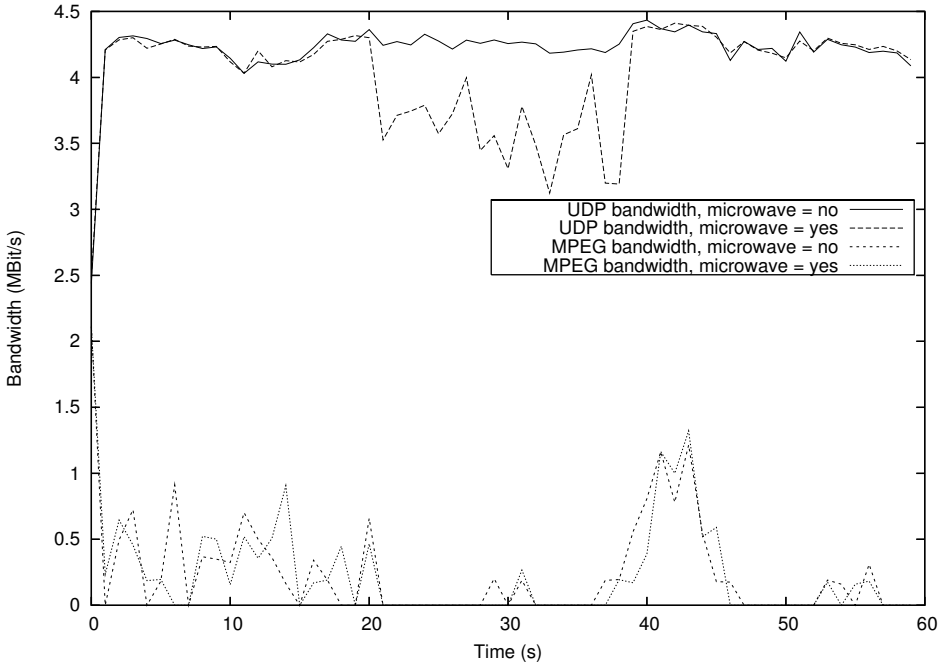


Figure 3.38: Throughput. Protocol = non-blocking UDP, retry value = 4, bitrate = 6 Mbit/s, time = afternoon.

the former undoubtedly attributes to this phenomenon, the amount of loss is too high for this to be the only cause. Loss in the send buffer is more plausible: the bitrate is too high for the link to be supported, so the send buffer gets full. Since it does not block, it overflows, discarding the ends of frames while stil transmitting the beginnings.

An optimization to help prevent this phenomenon might be to prevent transmission of packets that belong to a frame if one or more of its UDP packets are dropped in the send buffer.

Table 3.14 shows, for a 3 Mbit/s stream, for each type of MPEG frame the average amount of frames that were lost in each of the three measurement points, in addition to the percentage of that type of frame that was lost. We see that, the lower the retry value, the higher the amount of loss. Also, more loss occurs when interference is present than when it is not. Table 3.15 shows the same for a 4 Mbit/s stream.

In Tables 3.16 and 3.17, we see the amount of loss for 5 Mbit/s and 6 Mbit/s streams, respectively. The loss is more or less constant regardless of retry value. We see that the amount of loss among B-frames is relatively lower than among I and P-frames. The cause of this is probably the fact that B-frames are smaller than I and P-frames, and therefore the chance for a B-frame to overflow the send buffer is smaller than the chance for an I or a P-frame. Note also that, quite unexpectedly, in many cases, the amount loss is slightly lower when interference was present. Possibly, this is caused by transmission rate, which is lower when interference is present. This way, the packets that do get sent have a better chance of arriving at the receiver.

Table 3.18 shows, for each permutation of the parameters, how often UDP packets (as opposed to MPEG frames) were lost consecutively. The first column shows the percentage of times when a sequence of 1 UDP packet was lost. The second column shows the percentage of times when a sequence of 2 UDP packets was lost, and so on. The last two columns show the total number of lost packets, and the number of times a sequence of one packet or more was lost, respectively. We see that UDP packet losses often occur in bursts, even when network conditions are favourable. This is probably caused by the fact that, when the send buffer overflows, several packets' worth of data is discarded. However, never more than 10 packets are lost at a time. These bursts mean that MPEG frames are usually not discarded due to the loss of a single packet, but due to the loss of several. This lessens the blow from losing a single packet somewhat.

In Figure 3.39 we see the latency for a 3 Mbit/s stream without interference. Note that GStreamer's internal mechanics add a certain offset to the latency. For retry values 3 and 4, the latency fluctuates around a horizontal line, which is does not depart from. For retry values 1 and 2, the latency is more erratic and has a higher amplitude, but still it stays within its range.

With interference, we see in Figure 3.40 that the latency is more erratic and has higher peaks. The straight diagonal line we see from the 20 s mark for retry value 1 is caused by the fact that no data arrived in this period of time. Gnuplot, the program that was used to generate the graphs in this thesis, handled this by simply drawing a straight line between the two points.

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	1.0	1.0	0.0	0.0	19.7	2.0	20.7	1.4
	yes	0.7	0.7	0.3	0.1	0.7	0.1	1.7	0.1
1	**no**	21.0	20.8	74.7	18.7	166.0	16.6	261.7	17.5
	yes	46.7	46.2	181.0	45.2	433.0	43.4	660.7	44.1
2	**no**	2.3	2.3	28.7	7.2	50.3	5.0	81.3	5.4
	yes	21.3	21.1	77.7	19.4	180.3	18.1	279.3	18.6
3	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	4.7	4.6	16.0	4.0	33.3	3.3	54.0	3.6
4	**no**	1.3	1.3	0.0	0.0	2.7	0.3	4.0	0.3
	yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	0.7	0.7	3.3	0.8	6.0	0.6	10.0	0.7
16	**no**	1.0	1.0	0.7	0.2	2.7	0.3	4.3	0.3
	yes	1.0	1.0	7.0	1.8	16.7	1.7	24.7	1.6
	total	101		400		998		1499	

Table 3.14: Non-blocking UDP: average frame losses for 3 Mbit/s movie

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	12.7	12.5	52.0	13.0	73.7	7.4	138.3	9.2
1	**no**	38.7	38.3	129.3	32.3	281.7	28.2	449.7	30.0
	yes	57.3	56.8	216.0	54.0	503.7	50.5	777.0	51.8
2	**no**	1.0	1.0	3.7	0.9	7.0	0.7	11.7	0.8
	yes	23.0	22.8	87.3	21.8	192.7	19.3	303.0	20.2
3	**no**	0.3	0.3	1.0	0.2	2.0	0.2	3.3	0.2
	yes	15.7	15.5	51.0	12.8	84.7	8.5	151.3	10.1
4	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	14.7	14.5	55.0	13.8	69.7	7.0	139.3	9.3
8	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	18.0	17.8	69.3	17.3	96.3	9.7	183.7	12.3
16	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	yes	14.3	14.2	53.3	13.3	71.7	7.2	139.3	9.3
	total	101		400		998		1499	

Table 3.15: Non-blocking UDP: average frame losses for 4 Mbit/s movie

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	93.7	92.7	369.3	92.3	495.7	49.7	958.7	64.0
	yes	93.7	92.7	368.7	92.2	591.0	59.2	1053.3	70.3
1	**no**	95.3	94.4	377.0	94.2	651.7	65.3	1124.0	75.0
	yes	95.7	94.7	377.0	94.2	746.3	74.8	1219.0	81.3
2	**no**	93.7	92.7	363.0	90.8	498.0	49.9	954.7	63.7
	yes	95.7	94.7	368.0	92.0	621.0	62.2	1084.7	72.4
3	**no**	95.3	94.4	373.3	93.3	487.3	48.8	956.0	63.8
	yes	92.3	91.4	368.3	92.1	606.0	60.7	1066.7	71.2
4	**no**	95.0	94.1	370.7	92.7	491.7	49.3	957.3	63.9
	yes	92.7	91.7	370.7	92.7	587.0	58.8	1050.3	70.1
8	**no**	93.7	92.7	369.3	92.3	493.7	49.5	956.7	63.8
	yes	90.7	89.8	368.7	92.2	597.7	59.9	1057.0	70.5
16	**no**	94.7	93.7	369.0	92.2	490.7	49.2	954.3	63.7
	yes	92.7	91.7	371.0	92.8	601.0	60.2	1064.7	71.0
	total	101		400		998		1499	

Table 3.16: Non-blocking UDP: average frame losses for 5 Mbit/s movie

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	98.0	97.0	390.7	97.7	927.0	92.9	1415.7	94.4
	yes	98.3	97.4	390.3	97.6	923.7	92.6	1412.3	94.2
1	**no**	98.7	97.7	393.0	98.2	951.0	95.3	1442.7	96.2
	yes	98.7	97.7	391.3	97.8	949.7	95.2	1439.7	96.0
2	**no**	99.0	98.0	391.7	97.9	928.0	93.0	1418.7	94.6
	yes	99.0	98.0	390.0	97.5	924.0	92.6	1413.0	94.3
3	**no**	98.7	97.7	390.0	97.5	920.7	92.3	1409.3	94.0
	yes	98.7	97.7	391.0	97.8	924.0	92.6	1413.7	94.3
4	**no**	98.3	97.4	390.0	97.5	923.0	92.5	1411.3	94.2
	yes	98.3	97.4	390.0	97.5	923.0	92.5	1411.3	94.2
8	**no**	98.3	97.4	391.7	97.9	926.3	92.8	1416.3	94.5
	yes	99.0	98.0	388.3	97.1	931.0	93.3	1418.3	94.6
16	**no**	98.7	97.7	390.3	97.6	926.7	92.9	1415.7	94.4
	yes	97.7	96.7	390.0	97.5	925.7	92.8	1413.3	94.3
	total	101		400		998		1499	

Table 3.17: Non-blocking UDP: average frame losses for 6 Mbit/s movie

retry value	**bit rate**	**micro wave**	1	2	3	4	5-10	10+	**lost pkts**	**loss seq's**
1	**3**	**no**	23.9	13.4	11.4	11.9	39.4	0.0	2017	612
		yes	22.2	12.5	14.1	12.5	38.7	0.0	4164	1251
	4	**no**	18.1	12.3	11.8	11.0	46.9	0.0	4416	1240
		yes	15.8	11.1	11.4	10.2	51.5	0.0	8288	2237
	5	**no**	17.9	16.8	14.7	11.6	38.9	0.0	15372	4563
		yes	21.3	16.0	12.5	11.8	38.4	0.0	11872	3597
	6	**no**	13.0	13.0	12.7	12.0	49.3	0.0	31539	8486
		yes	14.4	12.2	12.9	11.7	48.7	0.0	21921	5954
2	**3**	**no**	8.3	9.1	13.4	5.9	63.4	0.0	1034	254
		yes	23.7	10.2	8.4	8.2	49.5	0.0	1370	392
	4	**no**	28.6	14.3	7.1	21.4	28.6	0.0	43	14
		yes	18.2	13.2	9.1	10.8	48.7	0.0	2482	692
	5	**no**	20.8	17.3	16.4	12.9	32.6	0.0	17549	5499
		yes	19.3	16.6	15.0	11.6	37.6	0.0	15966	4817
	6	**no**	12.3	13.5	13.2	12.8	48.3	0.0	40099	10803
		yes	12.8	13.6	13.7	12.7	47.3	0.0	34358	9333
3	**3**	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		yes	32.1	12.2	12.2	6.1	37.4	0.0	399	131
	4	**no**	16.7	33.3	16.7	16.7	16.7	0.0	17	6
		yes	17.3	14.1	13.2	9.0	46.4	0.0	2084	590
	5	**no**	18.9	18.0	15.9	12.4	34.9	0.0	17439	5340
		yes	17.5	15.4	14.2	12.4	40.5	0.0	19049	5554
	6	**no**	12.1	13.0	13.4	12.9	48.7	0.0	40100	10752
		yes	12.1	12.4	12.7	11.6	51.3	0.0	38618	10231
4	**3**	**no**	75.0	16.7	8.3	0.0	0.0	0.0	16	12
		yes	0.0	0.0	0.0	0.0	0.0	0.0	0	0
	4	**no**	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		yes	18.3	17.6	14.4	11.5	38.2	0.0	1948	584
	5	**no**	20.1	18.7	15.7	12.1	33.3	0.0	17542	5485
		yes	17.5	16.6	14.0	11.0	40.9	0.0	19577	5736
	6	**no**	12.5	13.4	12.9	12.7	48.5	0.0	40222	10837
		yes	11.9	12.4	12.9	11.5	51.2	0.0	40853	10818

Table 3.18: Non-blocking UDP: Probability (in %) of consecutive packet loss

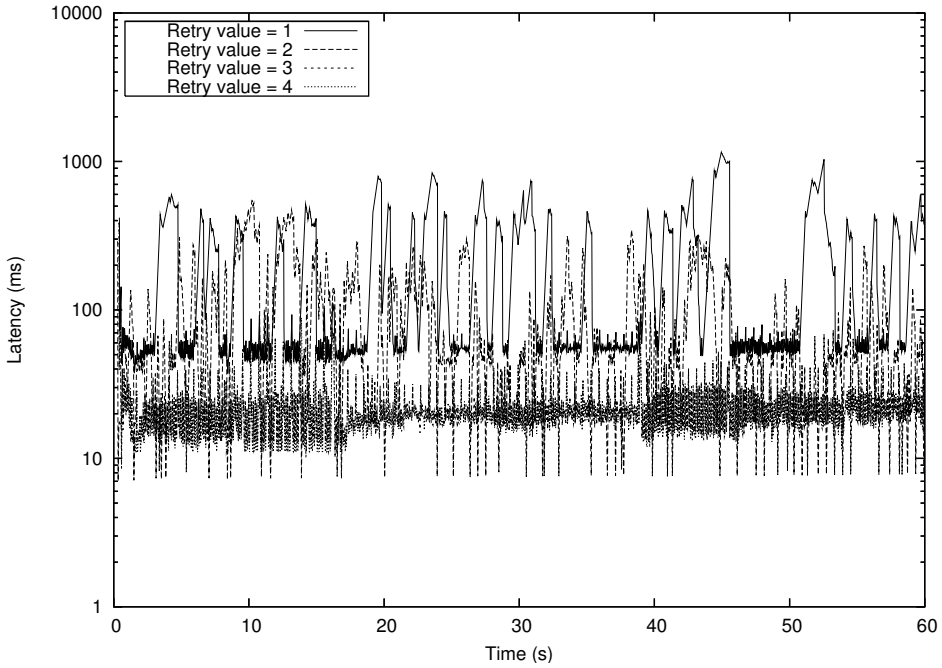


Figure 3.39: Latency. Protocol = non-blocking UDP, bitrate = 3 Mbit/s, microwave = no, time = afternoon. Logarithmic scale.

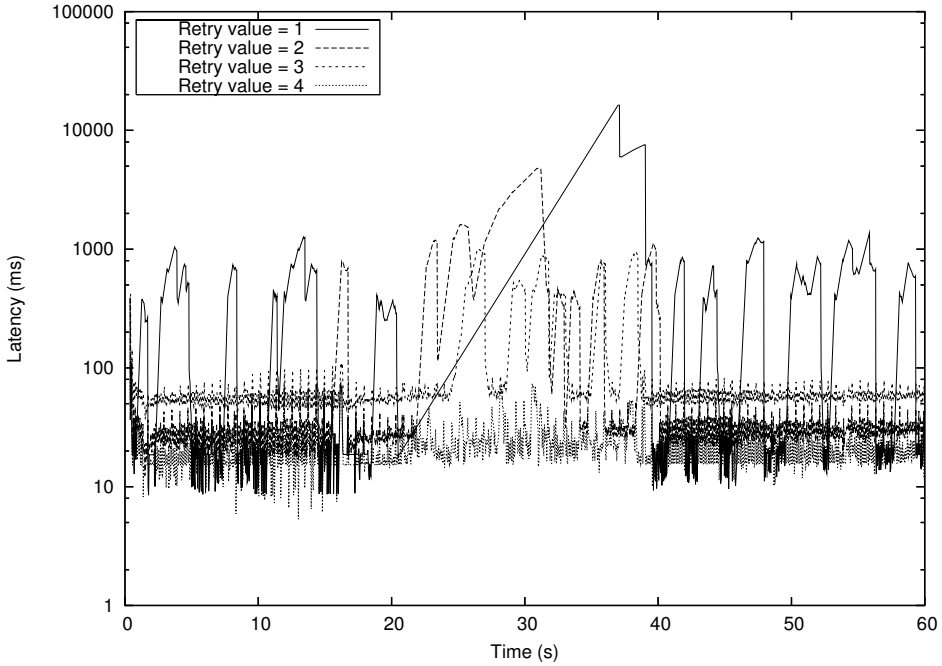


Figure 3.40: Latency. Protocol = non-blocking UDP, bitrate = 3 Mbit/s, microwave = yes, time = afternoon. Logarithmic scale.

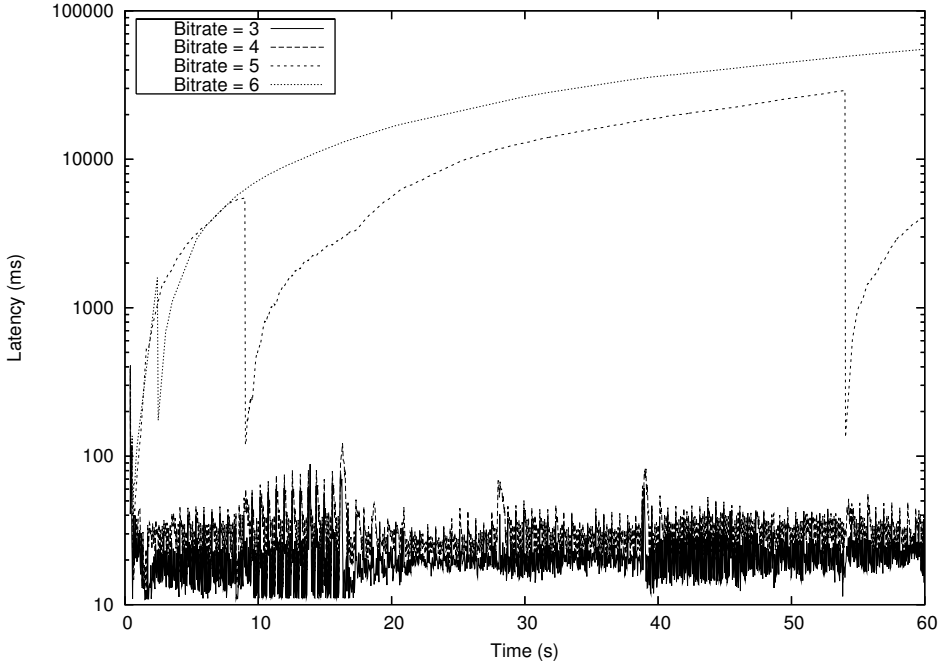


Figure 3.41: Latency. Protocol = non-blocking UDP, retry value = 4, microwave = no, time = afternoon. Logarithmic scale.

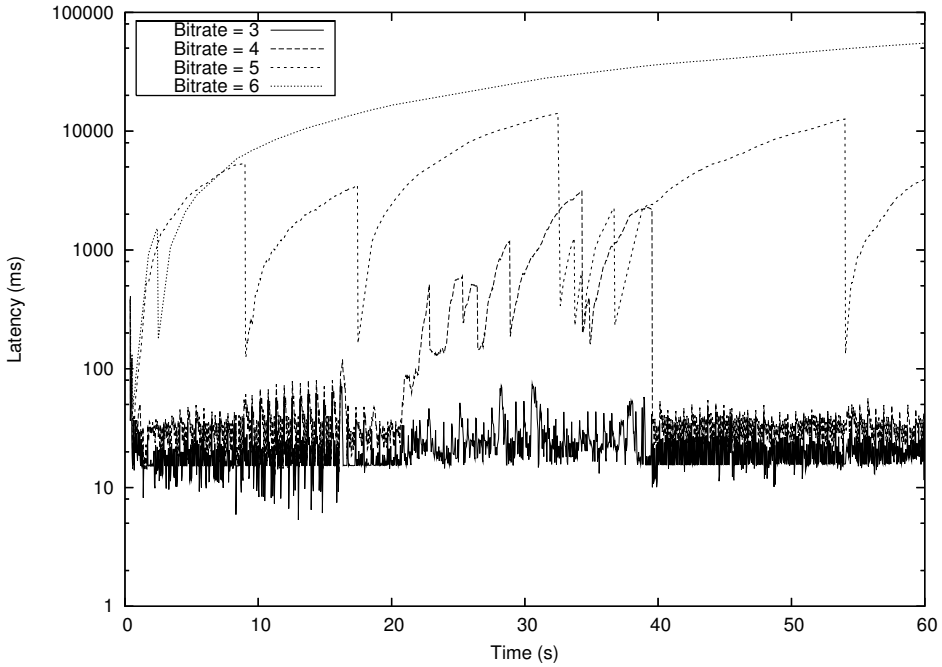


Figure 3.42: Latency. Protocol = non-blocking UDP, retry value = 4, microwave = yes, time = afternoon. Logarithmic scale.

In Figures 3.41 and 3.42, we see the latency for retry value 4, at different bitrates. At 3 and 4 Mbit/s, latency fluctuates around a horizontal line, except when interference is present. At 5 and 6 Mbit/s, the latency rises rapidly. This is caused by the fact that many frames are lost, including I-frames, which contain the GOP headers. The timestamp of a frame is relative to the GOP header's timestamp. This means that, if a GOP header is lost, the nth frame of a GOP has the same timestamp as the nth frame of the next GOP, which causes the difference between the timestamp and the wall clock time to grow quickly. Whenever a GOP header does come through, the latency is 'reset'. We see this in Figures 3.41 and 3.42 in the vertical drops most visible in the 5 Mbit/s line: these drops mark the intact arrival of a GOP header. For the 6 Mbit/s stream, a few GOP headers arrive in during the first couple of seconds of the stream, after which no GOP header comes through anymore. Needless to say that these figures do not give an accurate picture of the actual latency under these circumstances.

Figures 3.43 and 3.44 show the relation between MPEG frame size and latency with and without interference, for retry values 1 and 4 at bitrate 3 Mbit/s, and for bitrates 3 and 6 Mbit/s at retry value 4 respectively. Note that the latencies for bitrate 6 and retry value 4 are unreliable due to the phenomenon described above. Nevertheless, we discern no relation between the frames' latencies and their sizes.

In Figure 3.45, we see how the jitter develops in time for a 3 Mbit/s stream, without interference. Figure 3.46 show the same, with interference. For sufficiently high retry values, the jitter does not exceed 20 ms. In the presence of interference, the jitter peaks higher than that, though not in the case of retry value 4.

Figure 3.47 and 3.48 show the jitter for retry value 4, with and without interference respectively, at different movie bitrates. Note that the lines corresponding to the 5 and 6 Mbit/s streams are to be taken with a grain of salt, due to the incorrect timestamp values that were measured for these streams. At 3 and 4 Mbit/s, the jitter remains below 20 ms, except during the period of interference for the 4 Mbit/s stream.

Conclusions

1. An effective throughput of 4 Mbit/s was measured when no interference was present, the retry value was 3 or higher, and the movie bitrate did not exceed the effective throughput.
2. If the movie bitrate exceeds the effective throughput, overflow will occur in the send buffer, causing frames to be sent partially. These partial frames are subsequently discarded by the receiver, upon arrival. As a consequence of this loss, the throughput, while at the expected level of 4 Mbit/s when looking at UDP packets, drops dramatically when looking at MPEG frames. It should be possible to get much better results with some optimization, even without having to use IFD.
3. All graphs show that the duration of each stream was 60 s, as expected.
4. Relatively, the amount of loss among B-frames is smaller than the amount of loss among I or P-frames, which are more likely to overflow the send buffer due to their bigger size. This was expected.
5. Latency and jitter measurements for MPEG frame data become distorted when GOP headers are lost, because individual frames' timestamps cannot be decoded correctly as a result.

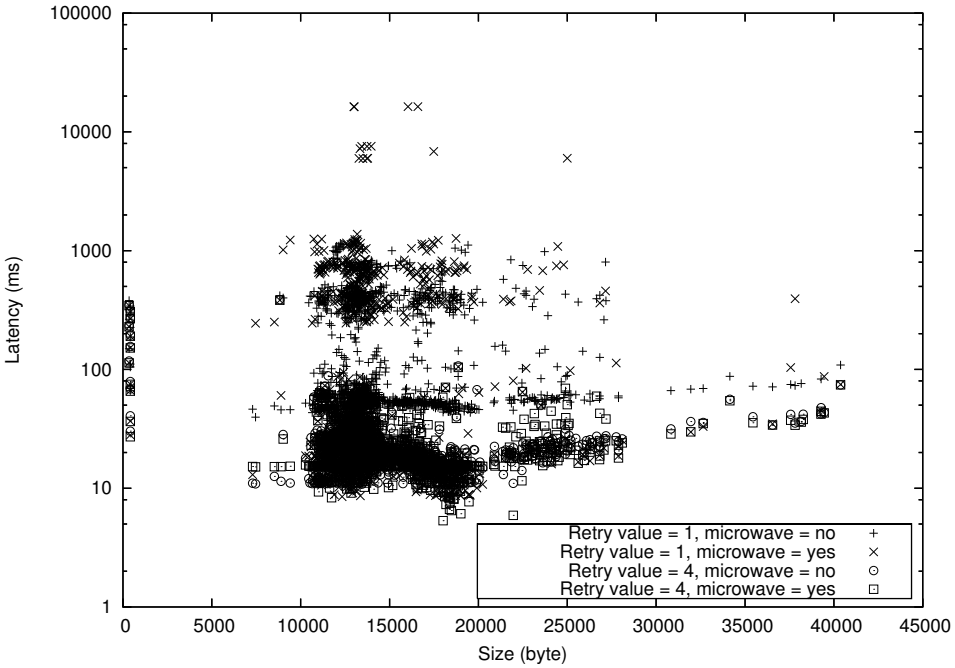


Figure 3.43: Size / latency. Protocol = Non-blocking UDP, bitrate = 3 Mbit/s, time = afternoon.

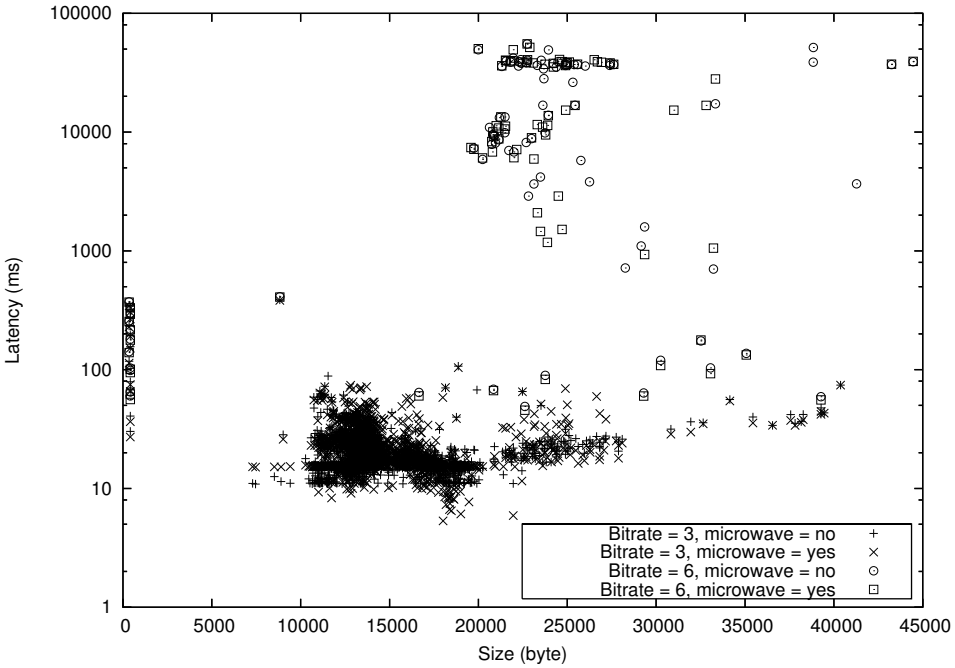


Figure 3.44: Size / latency. Protocol = Non-blocking UDP, retry value = 4, time = afternoon.

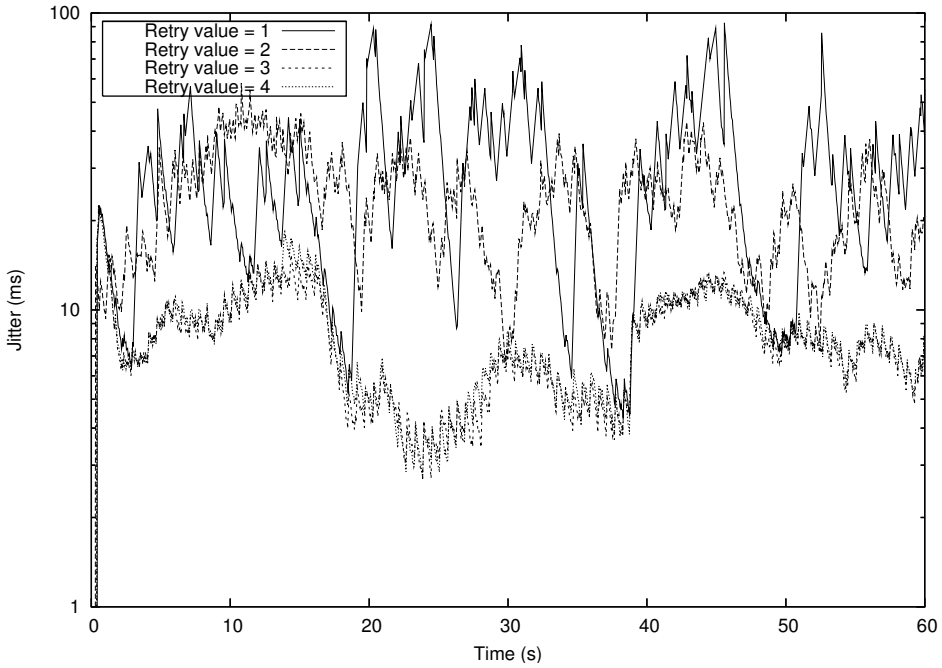


Figure 3.45: Jitter. Protocol = non-blocking UDP, bitrate = 3 Mbit/s, microwave = no, time = afternoon. Logarithmic scale.

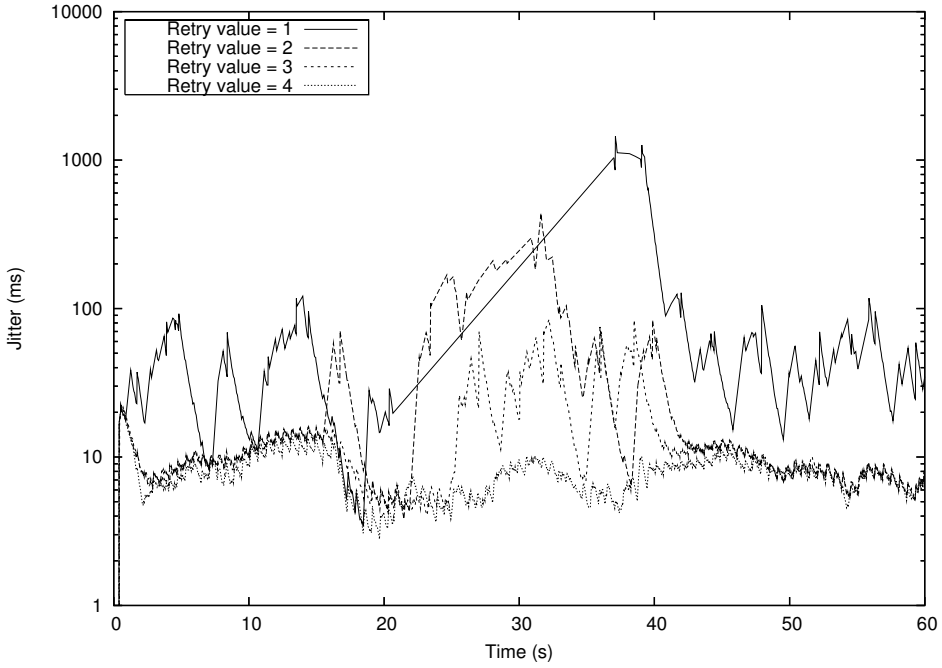


Figure 3.46: Jitter. Protocol = non-blocking UDP, bitrate = 3 Mbit/s, microwave = yes, time = afternoon. Logarithmic scale.

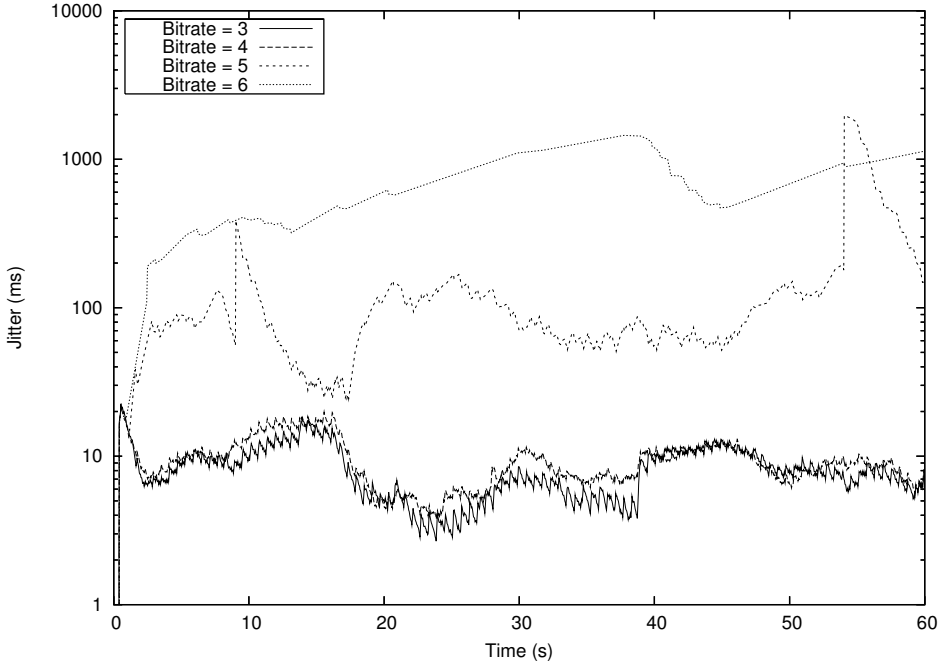


Figure 3.47: Jitter. Protocol = non-blocking UDP, retry value = 4, microwave = no, time = afternoon. Logarithmic scale.

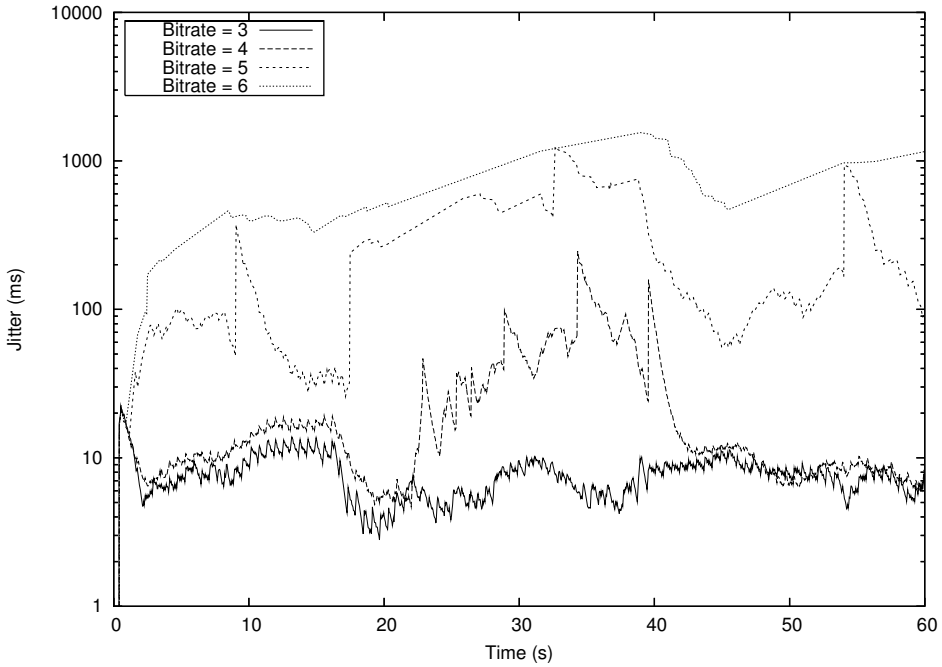


Figure 3.48: Jitter. Protocol = non-blocking UDP, retry value = 4, microwave = yes, time = afternoon. Logarithmic scale.

6. The size of an MPEG frame has no influence on its latency. The conjecture that larger size causes higher latency is therefore not correct.
7. For MPEG movies of 25 frames per second, one frame needs to be displayed every 40 ms. Under favourable network conditions, jitter does not exceed 20 ms. This means that only one frame needs to be buffered in order to eliminate any problems caused by jitter under favourable network conditions.

3.9 TCP with IFD

Set-up

We used the following GStreamer pipelines:

Sender: filesrc – ifdsink
Receiver: tcpsrc – mpegstat – fakesink

The sender's pipeline is a typical pipeline for TCP streaming, as seen in Section 1.6, but with the tcpsink replaced by the ifdsink we presented in Chapter 2. The receiver has the regular MPEG-2 decoder and the screen sink replaced by our mpegstat filter, which generates the data we need for the analysis, followed by the fakesink which simply discards the data.

For TCP connections in Linux, the sender's socket buffer is blocking and non-lossy. Losses may occur on the air; however, TCP's retransmission scheme makes sure no data is lost here. We don't intentionally drop data in the receiver's GStreamer pipeline. However, we do intentionally drop data in the sender's GStreamer pipeline if network conditions are unfavourable. Since this happens before writing the data to TCP, TCP's retransmission scheme does not retransmit this data.

Because the sender's socket buffer may block, and because of TCP's retransmission mechanism, delays may occur. However, when delays occur, IFD detects this and drops data, allowing the stream to be transmitted in real-time. Hence, a stream using TCP with IFD as transport protocol is a live stream, as opposed to a stream using TCP without IFD.

Note that we use the regular IFD algorithm, as depicted in Figure 1.1, and *not* the optimized version, depicted in Figures 1.2 and 1.3. The reason for this is that an implementation of the optimized algorithm was not yet available at the time the measurements were taken.

Results

In Figure 3.49 we see the MPEG throughput of a 6 Mbit/s movie stream in the afternoon test point. For retry value 1, the throughput is quite low at about 1 Mbit/s, but for retry value 2, it is already at 5 Mbit/s, with occasional peaks downward. For retry values 3 and 4, throughput is almost constant at 5 Mbit/s. When we add interference, as seen in Figure 3.50, we see a big dip between the 20 s and 40 s marks, which is the time in which interference was present. We see that, the higher the retry value, the better the throughput despite the interference.

Figure 3.51 shows the throughput at retry value 4, for the different bitrates. We see that the streams' throughputs are stable at the level of their respective bitrates, except for the 6 Mbit/s stream, which remains at about 5 Mbit/s. When interference is introduced, the 3 and 4 Mbit/s streams are unaffected, while the 5 and 6 Mbit/s streams experience a downfall during the period of time where interference is present.

In Table 3.19 the amount of loss for each type of MPEG frame is shown for a 3 Mbit/s movie stream. We see that, at retry value 1, many frames are lost, while at retry value 3 and higher, hardly any frames are lost. The effects of interference on the link are still noticeable at retry value 3, but for retry value 4, this is barely the case anymore. At retry value 1 without

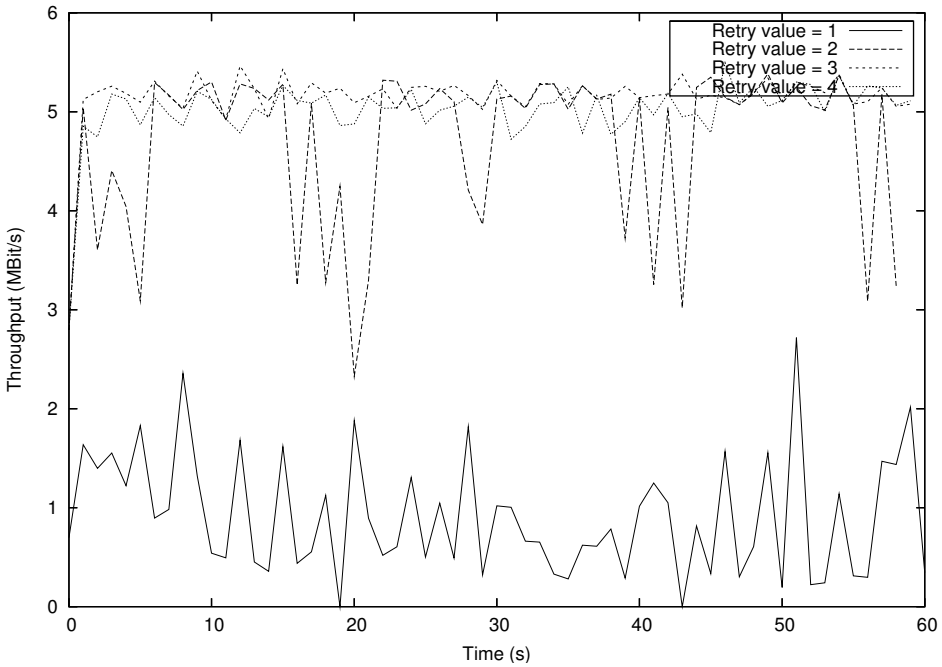


Figure 3.49: Throughput. Protocol = IFD, bitrate = 6 Mbit/s, microwave = no, time = afternoon.

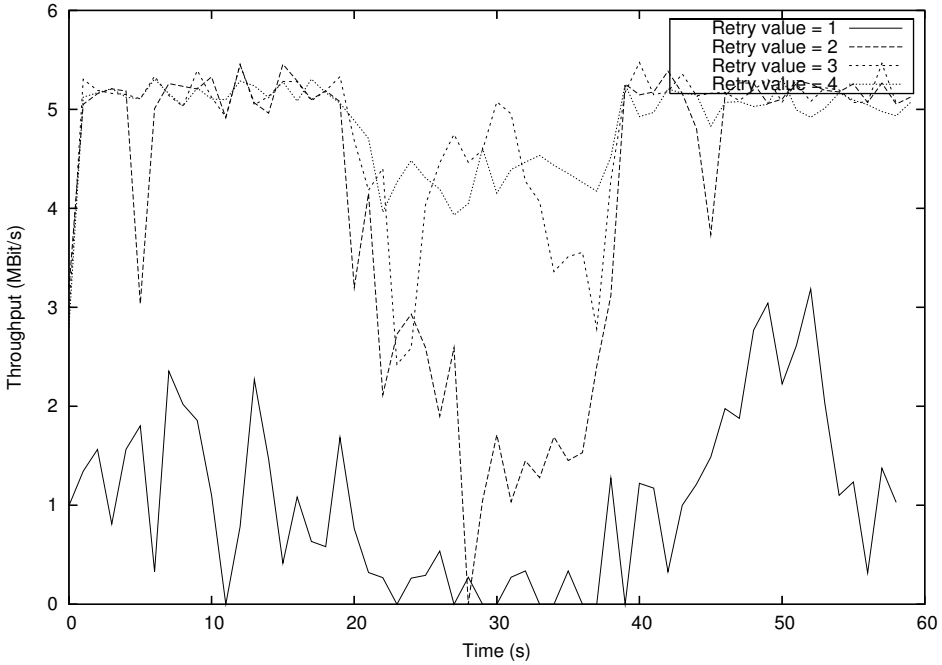


Figure 3.50: Throughput. Protocol = IFD, bitrate = 6 Mbit/s, microwave = yes, time = afternoon.

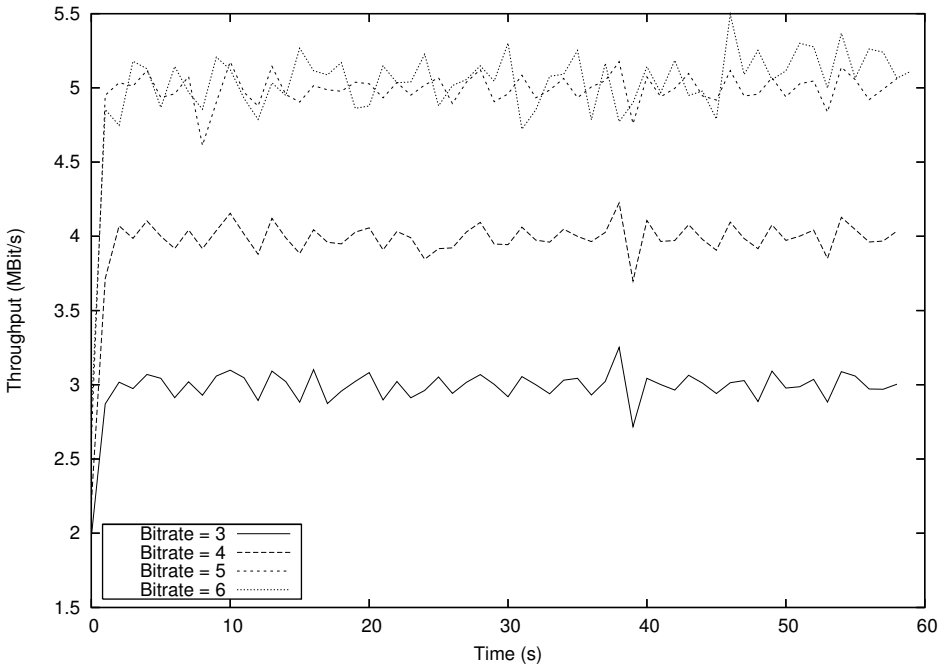


Figure 3.51: Throughput. Protocol = IFD, retry value = 4, microwave = no, time = afternoon.

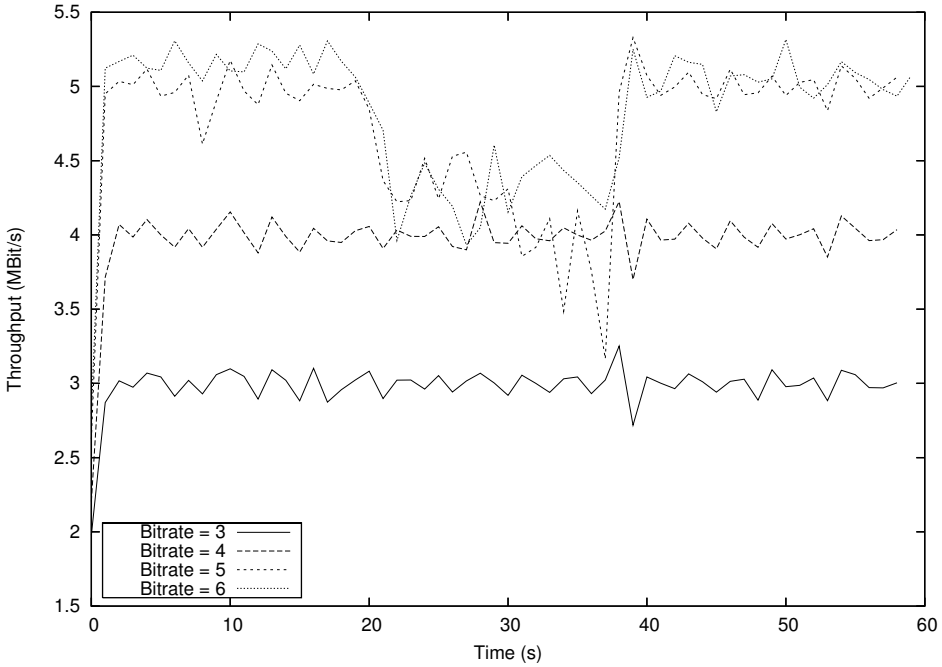


Figure 3.52: Throughput. Protocol = IFD, retry value = 4, microwave = yes, time = afternoon.

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	0.0	0.0	1.0	0.2	0.3	0.0	1.3	0.1
	yes	0.0	0.0	1.0	0.2	0.7	0.1	1.7	0.1
1	**no**	8.0	7.9	220.0	55.0	616.0	61.7	844.0	56.3
	yes	28.0	27.7	288.0	72.0	755.7	75.7	1071.7	71.5
2	**no**	0.0	0.0	8.0	2.0	21.3	2.1	29.3	2.0
	yes	4.0	4.0	52.0	13.0	138.3	13.9	194.3	13.0
3	**no**	0.0	0.0	1.0	0.2	0.7	0.1	1.7	0.1
	yes	0.7	0.7	10.3	2.6	29.7	3.0	40.7	2.7
4	**no**	0.0	0.0	1.0	0.2	0.7	0.1	1.7	0.1
	yes	0.0	0.0	1.7	0.4	2.7	0.3	4.3	0.3
8	**no**	0.0	0.0	1.0	0.2	0.3	0.0	1.3	0.1
	yes	0.0	0.0	1.0	0.2	0.3	0.0	1.3	0.1
16	**no**	0.0	0.0	1.0	0.2	0.7	0.1	1.7	0.1
	yes	0.0	0.0	1.0	0.2	0.7	0.1	1.7	0.1
	total	101		400		998		1499	

Table 3.19: IFD: average frame losses for 3 Mbit/s movie

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	0.0	0.0	1.0	0.2	0.0	0.0	1.0	0.1
	yes	0.0	0.0	1.0	0.2	6.7	0.7	7.7	0.5
1	**no**	9.7	9.6	264.0	66.0	796.0	79.8	1069.7	71.4
	yes	31.7	31.4	325.7	81.4	861.3	86.3	1218.7	81.3
2	**no**	0.0	0.0	4.3	1.1	10.3	1.0	14.7	1.0
	yes	5.3	5.3	64.0	16.0	176.0	17.6	245.3	16.4
3	**no**	0.0	0.0	1.0	0.2	1.0	0.1	2.0	0.1
	yes	0.0	0.0	14.0	3.5	37.0	3.7	51.0	3.4
4	**no**	0.0	0.0	1.0	0.2	1.3	0.1	2.3	0.2
	yes	0.0	0.0	3.3	0.8	6.7	0.7	10.0	0.7
8	**no**	0.0	0.0	1.0	0.2	0.3	0.0	1.3	0.1
	yes	0.0	0.0	1.0	0.2	6.3	0.6	7.3	0.5
16	**no**	0.0	0.0	1.0	0.2	0.3	0.0	1.3	0.1
	yes	0.0	0.0	1.7	0.4	16.0	1.6	17.7	1.2
	total	101		400		998		1499	

Table 3.20: IFD: average frame losses for 4 Mbit/s movie

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	0.0	0.0	1.0	0.2	0.7	0.1	1.7	0.1
	yes	0.0	0.0	1.0	0.2	82.7	8.3	83.7	5.6
1	**no**	18.7	18.5	325.7	81.4	920.7	92.3	1265.0	84.4
	yes	37.3	37.0	350.7	87.7	928.0	93.0	1316.0	87.8
2	**no**	0.3	0.3	5.7	1.4	15.0	1.5	21.0	1.4
	yes	4.3	4.3	75.7	18.9	216.0	21.6	296.0	19.7
3	**no**	0.0	0.0	1.0	0.2	1.0	0.1	2.0	0.1
	yes	0.3	0.3	10.3	2.6	73.3	7.3	84.0	5.6
4	**no**	0.0	0.0	1.0	0.2	0.7	0.1	1.7	0.1
	yes	0.0	0.0	1.0	0.2	77.0	7.7	78.0	5.2
8	**no**	0.0	0.0	1.0	0.2	1.3	0.1	2.3	0.2
	yes	0.0	0.0	1.3	0.3	68.3	6.8	69.7	4.6
16	**no**	0.0	0.0	1.0	0.2	1.3	0.1	2.3	0.2
	yes	0.0	0.0	4.7	1.2	75.7	7.6	80.3	5.4
	total	101		400		998		1499	

Table 3.21: IFD: average frame losses for 5 Mbit/s movie

retry value	micro wave	I #	I %	P #	P %	B #	B %	total #	total %
0	**no**	0.3	0.3	1.0	0.2	229.7	23.0	231.0	15.4
	yes	0.0	0.0	1.0	0.2	304.7	30.5	305.7	20.4
1	**no**	22.3	22.1	338.3	84.6	921.0	92.3	1281.7	85.5
	yes	37.3	37.0	343.7	85.9	945.3	94.7	1326.3	88.5
2	**no**	1.0	1.0	28.0	7.0	285.3	28.6	314.3	21.0
	yes	4.7	4.6	87.0	21.8	408.0	40.9	499.7	33.3
3	**no**	0.0	0.0	1.0	0.2	230.3	23.1	231.3	15.4
	yes	0.3	0.3	18.7	4.7	319.3	32.0	338.3	22.6
4	**no**	0.0	0.0	1.0	0.2	243.3	24.4	244.3	16.3
	yes	0.0	0.0	2.7	0.7	314.0	31.5	316.7	21.1
8	**no**	0.3	0.3	1.7	0.4	234.7	23.5	236.7	15.8
	yes	0.0	0.0	1.3	0.3	310.7	31.1	312.0	20.8
16	**no**	0.0	0.0	1.0	0.2	232.7	23.3	233.7	15.6
	yes	0.0	0.0	1.0	0.2	308.0	30.9	309.0	20.6
	total	101		400		998		1499	

Table 3.22: IFD: average frame losses for 6 Mbit/s movie

interference, the amount of frame loss among B-frames is 61.7 %, but only 7.9 % for I-frames. This clearly indicates that the ifdsink element works as expected.

In Tables 3.20, 3.21 and 3.22, we see the amounts of loss for a 4, 5 and 6 Mbit/s movie stream, respectively. We see a slight increase in the amounts of loss of B-frames as the bitrate increases. While a 6 Mbit/s movie bitrate exceeds the capability of the 5 Mbit/s link, we see that the amount of loss among I-frames is comparable to that of a 3 Mbit/s stream, while the amount of loss among B-frames increases dramatically. This is another sign that the ifdsink does its job well.

Latency is shown in Figures 3.53 through 3.56. Note that GStreamer's internal mechanics add a certain offset to the latency. We see in Figures 3.53 and 3.54 that, except at retry value 1, latency is almost constant, except for some rather high peaks in case of interference. In Figures 3.55 and 3.56, we see that the latency is rather constant for varying movie bitrates as well, except for the occasional peak. Only for the 6 Mbit/s stream, and for the 5 Mbit/s stream in the presence of interference, the latency is somewhat erratic and also substantially higher than at lower bitrates.

Figures 3.57 and 3.58, where the size of the MPEG frames is plotted against the latency of the same frame, respectively for a bitrate of 3 Mbit/s with retry values 1 and 4, and for a retry value of 4 with bitrates of 3 and 6 Mbit/s, show that there is no clear relation between the size of an MPEG frame and its latency.

In Figure 3.59, we see that the jitter of a 3 Mbit/s stream for most retry values rarely exceeds 1 ms, except when retry value is 1. If we introduce interference, as seen in Figure 3.60, we see especially high peaks for retry values 2 and 3. At retry value 4, the jitter manages to remain under 5 ms.

In Figure 3.61, we see that the jitter for a 6 Mbit/s stream fluctuates more or less between 10 ms and 20 ms. At lower bitrates, the jitter never exceeds 10 ms and usually stays below 1 ms. If we introduce interference, as seen in Figure 3.62, we see some more peaks, especially for the 5 Mbit/s movie. However, still the jitter rarely exceeds 20 ms, and does this only at bitrates of 5 and 6 Mbit/s.

Conclusions

1. The effective throughput is 5 Mbit/s for most retry values. For retry value 1, the throughput is only about 1 Mbit/s. So, for good performance, the retry value must be higher than 1.
2. A stream with a bitrate of 3 or 4 Mbit/s is more resilient against interference and bad network conditions than a stream of which the bitrate is closer to (or exceeding) the maximal effective throughput.
3. The loss tables indicate that the implementation of the ifdsink, as presented in Chapter 2, works quite effectively: I-frames are almost never discarded, P-frames very infrequently, and B-frames often, as expected.
4. The size of an MPEG frame has no influence on its latency. The conjecture that larger size causes higher latency is therefore not correct.

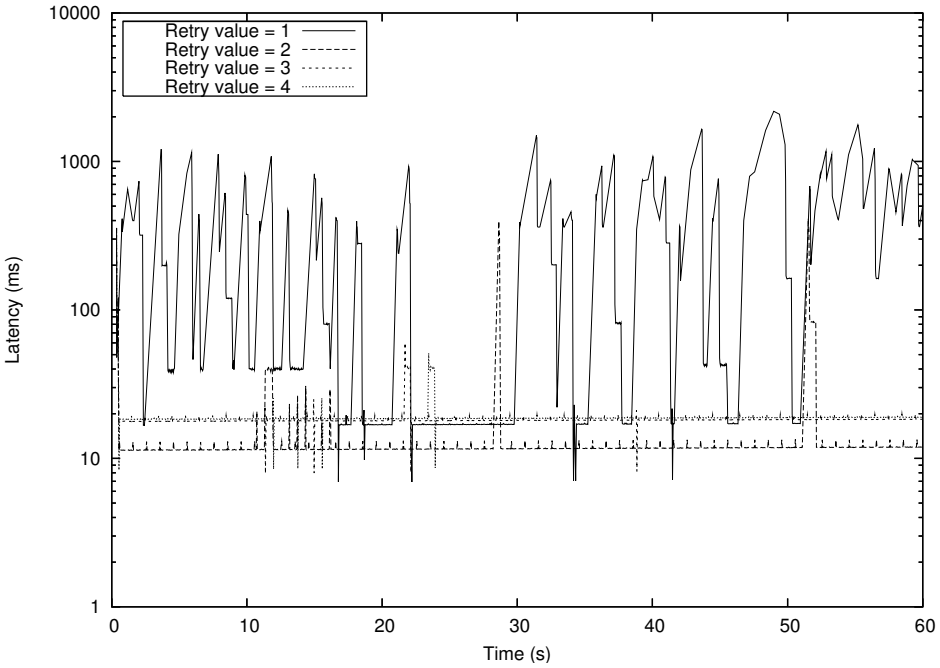


Figure 3.53: Latency. Protocol = IFD, bitrate = 3 Mbit/s, microwave = no, time = afternoon. Logarithmic scale.

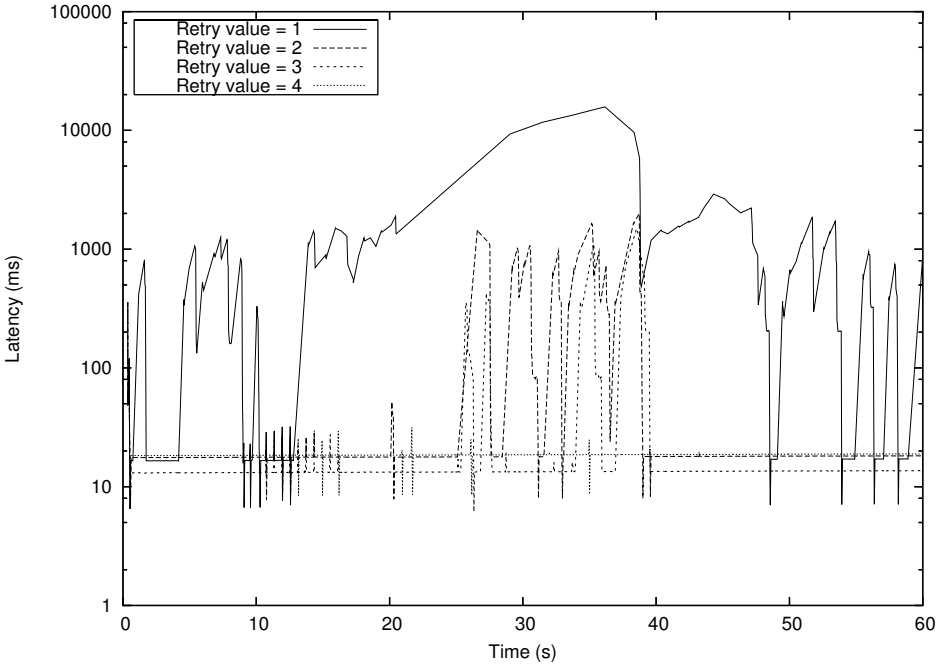


Figure 3.54: Latency. Protocol = IFD, bitrate = 3 Mbit/s, microwave = yes, time = afternoon. Logarithmic scale.

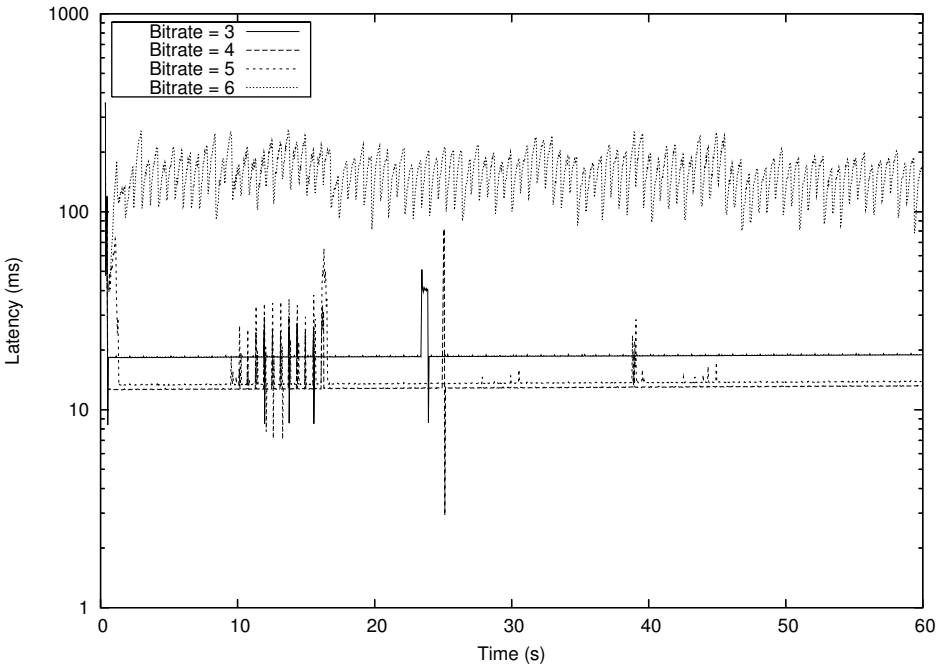


Figure 3.55: Latency. Protocol = IFD, retry value = 4, microwave = no, time = afternoon. Logarithmic scale.

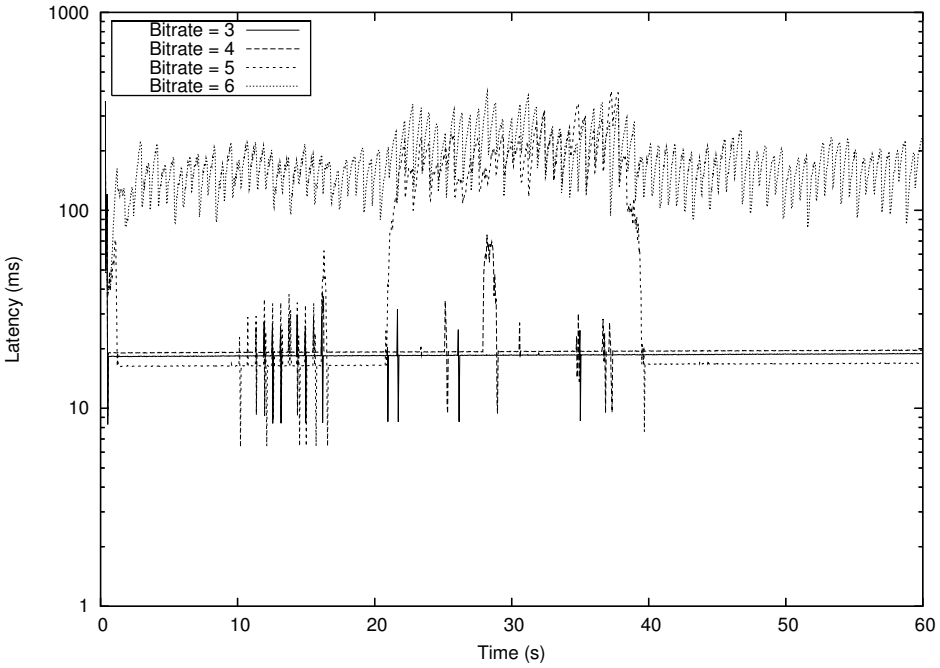


Figure 3.56: Latency. Protocol = IFD, retry value = 4, microwave = yes, time = afternoon. Logarithmic scale.

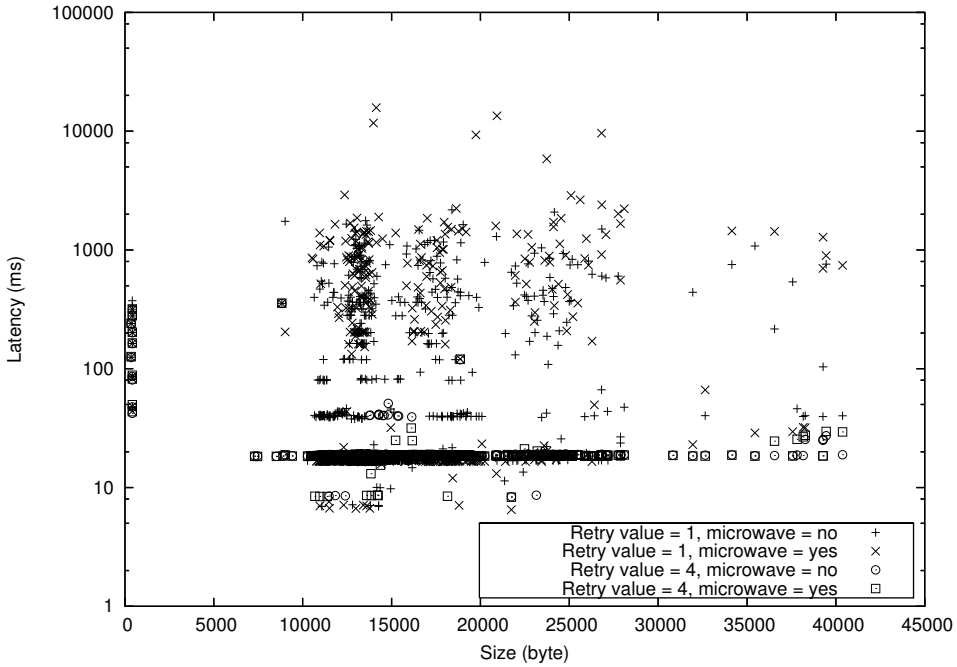


Figure 3.57: Size / latency. Protocol = IFD, bitrate = 3 Mbit/s, time = afternoon.

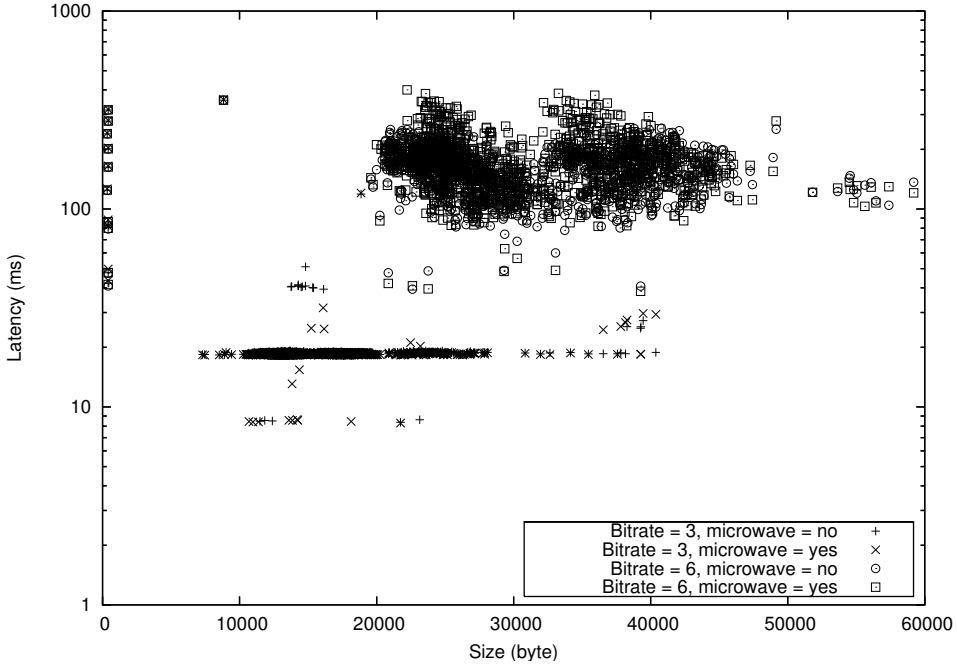


Figure 3.58: Size / latency. Protocol = IFD, retry value = 4, time = afternoon.

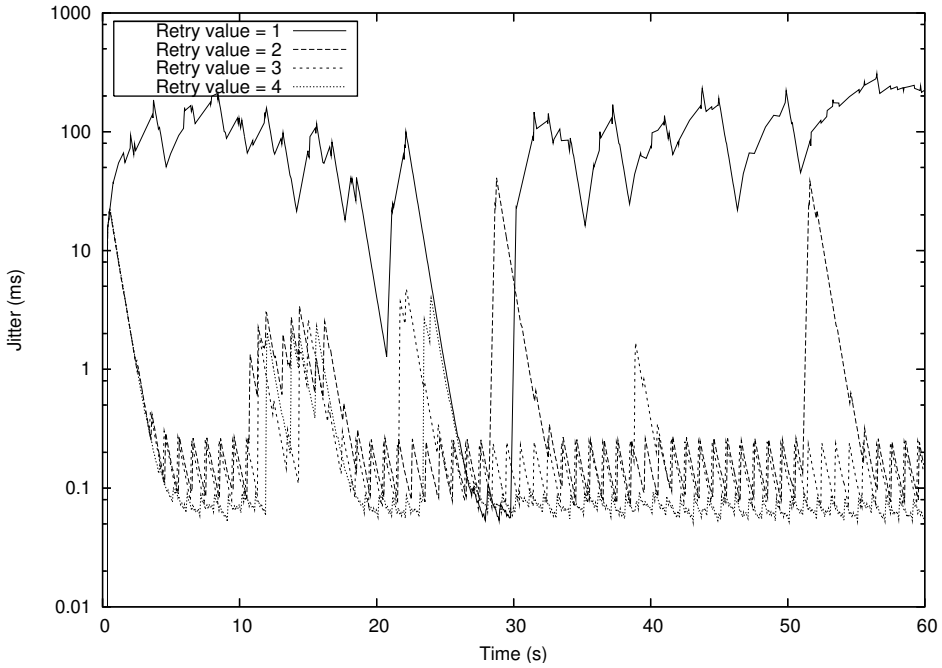


Figure 3.59: Jitter. Protocol = IFD, bitrate = 3 Mbit/s, microwave = no, time = afternoon. Logarithmic scale.

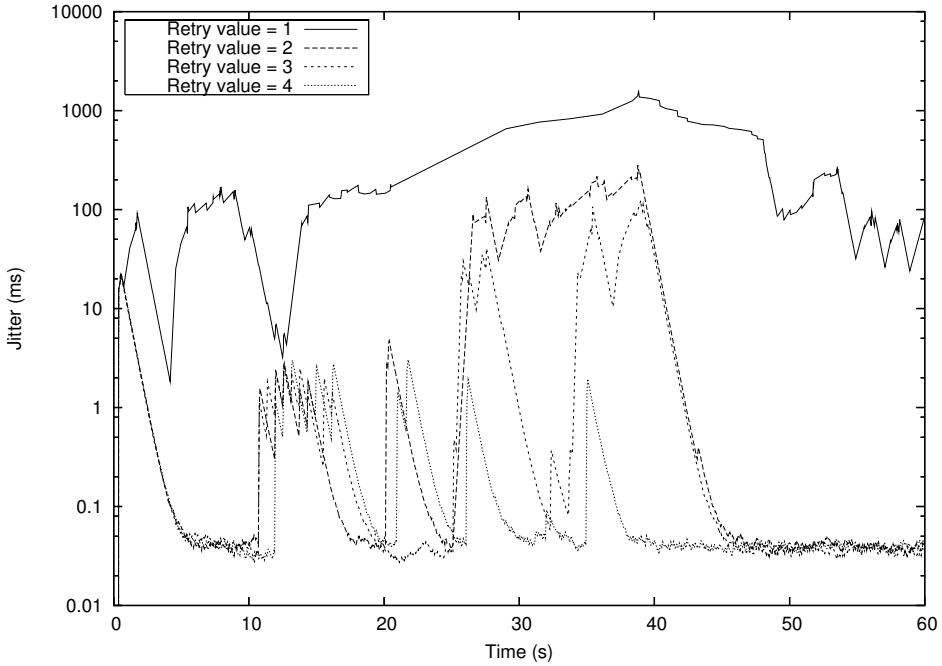


Figure 3.60: Jitter. Protocol = IFD, bitrate = 3 Mbit/s, microwave = yes, time = afternoon. Logarithmic scale.

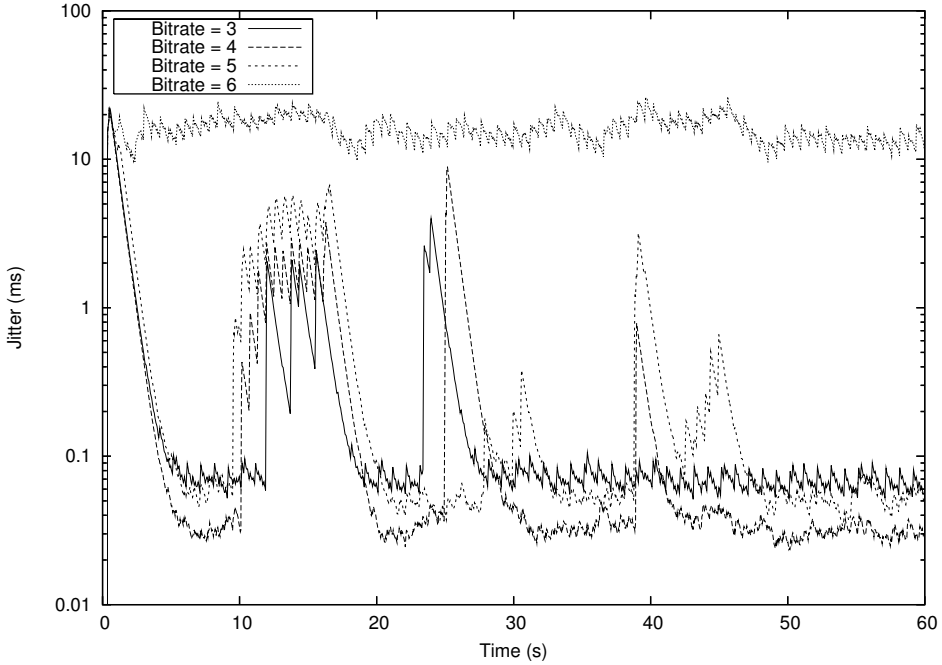


Figure 3.61: Jitter. Protocol = IFD, retry value = 4, microwave = no, time = afternoon. Logarithmic scale.

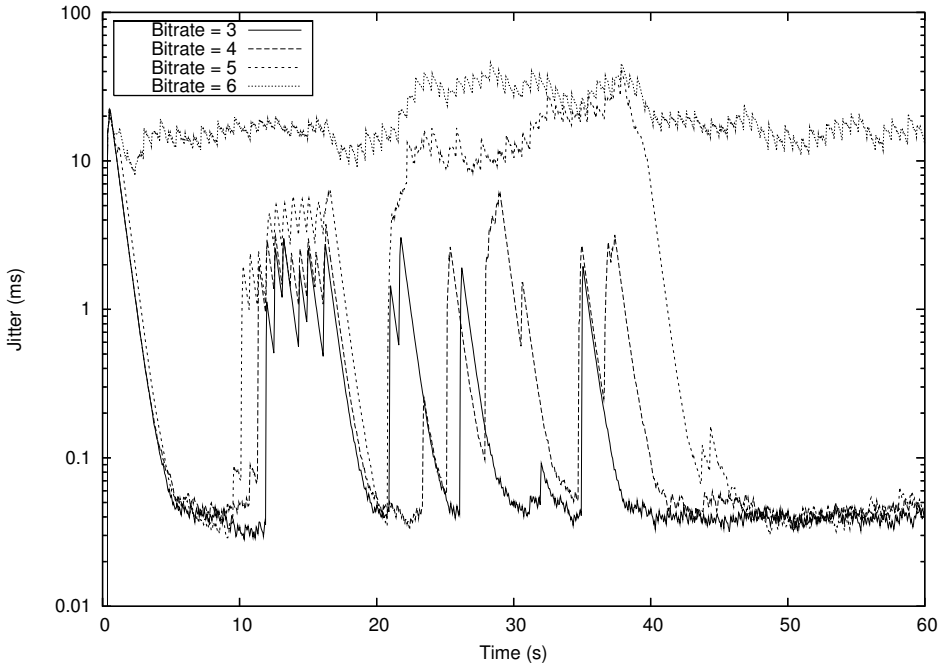


Figure 3.62: Jitter. Protocol = IFD, retry value = 4, microwave = yes, time = afternoon. Logarithmic scale.

5. For MPEG movies of 25 frames per second, one frame needs to be displayed every 40 ms. Only in rare cases, when the retry value is low and interference is present, the jitter exceeds 10 ms. This means that a buffer of 1 MPEG frame suffices to eliminate most problems caused by jitter.
6. All graphs show that, in all cases, the streams end after 60 s, as expected. Movies are not simply cut off after 60 s, as in that case, many more of the evenly distributed I-frames would have been lost, which is not the case. This means that IFD streams are indeed live.

3.10 Discussion

The first thing to note when comparing the protocols' measurements with each other, is the throughput. We would expect UDP to achieve a higher throughput than TCP. This was indeed the case in our preliminary Iperf measurements. With high retry value and no interference, we measured an effective throughput of about 5 Mbit/s for TCP, and about 6 Mbit/s for UDP. However, the throughputs for MPEG streams under the same circumstances yield a throughput of only 4 Mbit/s for UDP (blocking and non-blocking, see Figures 3.5 and 3.33), while TCP's remains 5 Mbit/s (see Figures 3.19 and 3.49). The probable cause for this is the size of the outgoing packets. The packet size for TCP, both for Iperf and for GStreamer MPEG streams, is maximal, in this case 1500 bytes. Iperf uses maximally sized packets for UDP as well. However, the RFC 2250 protocol for MPEG packetization that was used in GStreamer's UDP streams, causes the size of the packets to be much smaller [5]. Other research shows that the size of individual packets can have an influence on transmission speed over wireless LANs [20]. So, to achieve the best performance on wireless networks, UDP packets need to be as large as possible.

Also, we see that for all protocols, the stream needs about a second to attain its nominal level of throughput. The cause for this is the movie we used: the first GOP consists of very small image frames. For TCP, the slow start algorithm may also be a factor, but as it is also very apparent for UDP, it is probably negligible.

In Tables 3.7 and 3.13, we see that, for all retry values except 1, the stream's duration is longer for blocking UDP than it is for TCP. The duration is a function of the throughput for non-live protocols: the lower the throughput, the higher the duration of a stream. This is consistent with the results for throughput.

For retry value 1, UDP out-performs TCP in speed. This is probably due to the fact that, in this case, TCP's retransmission scheme and congestion control [14, 17] cause the stream to slow down more than strictly necessary. Also, the amount of packet loss on the air becomes an important factor for blocking UDP at retry value 1.

When comparing the amount of loss of MPEG frames for the blocking UDP protocol (see Tables 3.8 through 3.11) with that of non-blocking UDP (Tables 3.14 through 3.17), we see that the latter is much higher under almost any circumstances, and especially in the presence of interference. While loss for blocking UDP is due to losses on the air only, the loss for non-blocking UDP is caused by a combination of loss on the air and loss due to send buffer overflow. When interference is present, the amount of loss increases for both protocols, though much more steeply for non-blocking UDP than for blocking UDP. The wireless card notices the interference, and rather than risking loss on the air, it starts transmitting at a lower rate. This causes the send buffer to overflow more quickly, amounting to an increase in loss.

If we compare the amount of loss for non-blocking UDP (Tables 3.14 through 3.17 with the amount of loss for IFD (Tables 3.19 through 3.22), we see that, for bitrates of 3 and 4 Mbit/s, the amount of loss for IFD is smaller than the amount of loss for non-blocking UDP, except for retry value 1. This is consistent with the observations on throughput we made in the previous paragraph, that TCP achieves higher throughputs than UDP. We also observe that in general, for non-blocking UDP, relatively more I-frames are dropped than P-frames, and relatively more P-frames than B-frames. For IFD, this is the other way round, with 0 % I-frame loss in most cases. For bitrates of 5 and 6 Mbit/s, we can only say that, unlike non-blocking UDP, IFD manages to successfully transmit quite a few MPEG frames.

For a bitrate of 6 Mbit/s and retry values of 3 and up, when no interference is present, it even manages not to drop any I-frames, even though the available bandwidth is smaller than the stream's bitrate. We can therefore safely say that IFD on TCP performs better than Non-blocking UDP. Note, however, that using IFD on top of UDP will probably yield results comparable to IFD on TCP.

Latencies for the UDP protocols fluctuate much, as seen in Figures 3.9 through 3.12 for blocking UDP and 3.39 through 3.42 for non-blocking UDP, while latencies for the TCP protocols are reasonably constant, as seen in Figures 3.23 through 3.26 for TCP without IFD and Figures 3.53 through 3.56 for TCP with IFD. Note that this is the latency for the MPEG frames, and not the UDP or TCP packets. These fluctuations in latency lead to higer jitter values and require more buffering to compensate for them.

It is not entirely clear why this happens. We know that, while GStreamer sends out bursts of packets every 40 ms for UDP (for a movie of 25 frames per second), for TCP it lets the receiver handle the timing while it simply sends a packet as soon as it becomes available. So, we might conclude that, for UDP, larger MPEG frames have higher latencies. However, we know from Figure 3.13, amongst others, that this is not the case. Possibly, though, it may depend on the number of packets that were necessary to transmit a frame, or on their respective sizes. Although these are functions of the MPEG frame size, they also depend on the RFC 2250 protocol.

The difference in jitter between UDP (Figures 3.15 through 3.18 and 3.45 through 3.48) and TCP (Figures 3.29 through 3.32 and 3.59 through 3.62) is directly related to the differences in latency. TCP with and without IFD has very smooth latency graphs, and therefore, with values in the order of 1 ms, the jitter remains small, compared to UDP, with values in the order of 10 ms or higher.

4 Conclusions and future work

4.1 Conclusions

1. GStreamer is quite suitable as a framework for MPEG video streaming. It can be used to set up streams over a variety of protocols, including UDP/RTP, TCP, and HTTP. Moreover, it is easily extended to support other protocols, such as IFD. Howevr, it uses much CPU power and memory and may not be suitable for all environments.
2. The IFD GStreamer element, as presented in Chapter 2, works well. Hardly any I-frames are lost, whereas I-frames are the most likely to be lost of all frame types when no IFD is present.
3. Three factors play an important role in the performance of the IEEE 802.11b protocol: the retry value, the transmission rate, and the size of the transmitted packets.
4. For good performance, for any protocol, the retry value must be 3 or higher. Retransmissions on the IEEE 802.11b layer are more rare than previously assumed. For a retry value of 1 –one transmission, no retransmissions– performance is bad. A retry value of 2 is a big improvement, and a retry value of 3 is better still. After that, the performance stabilises, increasing only slightly.
5. For MPEG-2 video streams over wireless IEEE 802.11b LANs, TCP performs better than UDP, due to the variation in packet sizes for the latter. This variation is caused by the RFC 2250 packetization algorithm, which is necessary for UDP based streams. If the retry value is set to 1, however, UDP outperforms TCP. Note that this conclusion is valid for the wireless home environment only. On the internet, for example, delays caused by TCP retransmissions are much longer. Also, because the internet is mostly wired, the influence of packet size is quite different, as smaller frames do not cause the delays they do in wireless networks.
6. Non-blocking UDP performs extremely poorly when the stream's bitrate exceeds the available bandwidth on the link. The larger the bitrate, the more quickly the send buffer overflows and partial frames are transmitted. These partial frames are then discarded by the MPEG parser on the receiver, because they are incomplete. As a result, there is a lot of traffic on the link of which only a small portion is useful. A simple optimization is probably enough to avoid this problem, even without IFD.
7. Streams with a bitrate that is much lower than the available bandwidth on the link are less susceptible to delays or losses caused by interference than streams with a bitrate close to, or exceeding, the available bandwidth. A bitrate that is 1 Mbit/s less than the available bandwidth on the link when no interference is present, is enough to avoid problems when microwave interference is present (i.e.: a 4 Mbit/s stream on a 5 Mbit/s link for TCP, a 3 Mbit/s stream on a 4 Mbit/s link for UDP).
8. There is no relation between MPEG frame size and latency.

9. Jitter is much smaller for TCP streams than it is for UDP streams. However, in both cases, the jitter seldom exceeds the time between two MPEG frames (40 ms for a movie of 25 frames per second). Therefore, in most cases, buffering one frame on the receiver is enough to overcome problems caused by jitter. To be safe, one or two extra frames can be buffered.

4.2 Recommendations for future work

1. To do more research into the behaviour of the IEEE 802.11b protocol. Due to the unreliability of the sniffer, the exact amount of retransmissions was impossible to determine, as we have seen in Section 3.5. Also, different hardware may exhibit different behaviour. For example, with another wireless card it may be possible to set a fixed transmission rate, which we were unable to do. When we set the retry value to 1, the card we have used would detect a deterioration of the network, and lower the transmission rate as a result, making it difficult to see the effects of the retry value and of the transmission rate in isolation.
2. To investigate the influence of packet size on the transmission speed over IEEE 802.11 networks. Furthermore, it would be interesting to see the influence of the packetization of the RFC 2250 protocol on the transmission speed, through the packet size.
3. To perform measurements using IEEE 802.11g or IEEE 802.11a as internetwork protocol. It is likely that the behaviour of streams over these protocols is similar to that of IEEE 802.11b, but we are not certain. Especially in the case of IEEE 802.11g, it is worthwhile to do more research, as it is IEEE 802.11b's successor.
4. To perform measurements with more than one simultaneous stream, possibly using IEEE 802.11g instead of b to accommodate for the higher bandwidth requirements. The measurements in this thesis only considered the streaming of one movie at a time, but it is possible in the home that two persons want to watch two different movies at the same time. This may cause interference on the wireless level, and TCP's congestion control may also pose some interesting problems.
5. To compare the the performance of IFD over UDP with that of IFD over TCP. The measurements in this thesis only cover UDP without IFD.
6. To adapt the ifdsink element in such a way that it is also compatible with UDP. This would be most useful for comparing IFD over UDP and TCP since, at present, IFD for UDP lives in the network layer and IFD for TCP lives in the application layer. By adapting ifdsink for UDP, we would have a similarly implemented IFD in the application layer, which makes the comparison more fair.
7. To port all OASIS specific GStreamer plugins to the new API. GStreamer has changed its plugin API after version 0.7.1. All of the GStreamer plugins written by OASIS cluster members are written for the old API. Existing plugins are often improved upon, new plugins appear frequently and also the documentation for the newer versions grows more complete each version. It is a pity not to be able to take advantage of this, just for the sake of a handful of plugins that can easily be ported.

A GStreamer

A.1 Background

GStreamer is a great application, but it is still quite rough around the edges. In this appendix, I hope to save the reader the time and frustration that it cost me to get GStreamer to work the way I wanted to.

Currently, the two most widely used versions of GStreamer are 0.6 and 0.8, where 0.6 is being phased out step by step in favour of 0.8. However, at the time of writing, the OASIS cluster still uses version 0.6. An important thing to know is that GStreamer's plugin API changed between versions 0.7.1 and 0.7.2. These APIs are mutually incompatible, so 0.6 plugins are incompatible with the 0.8 base system, and vice versa. It is therefore always very important to know with which version one is working. Since the OASIS cluster currently works with version 0.6, I will address only this version here, and ignore version 0.8.

GStreamer can be installed globally and locally. A global install means that it is available for all users of the computer it is installed on. A local install lives in one's home directory and is therefore inaccessible to other users. It is easier to install GStreamer globally; however, this requires root privileges and affects the global state of the system, which may not be desirable. For development purposes, a local install is more practical, because it does not require additional installation after compilation.

Also, it is important to note that the GNOME desktop environment uses GStreamer internally. This can lead to a lot of complications. It is therefore advisable to use KDE, or some other window manager. If this is not possible (for instance because you believe KDE is evil), do not install GStreamer globally, for this will thoroughly mess up your system. Also, take additional care that the 'new' GStreamer's configuration files do not conflict with the existing one's.

A.2 How to install GStreamer

The first step is to obtain GStreamer. I have prepared a CD-ROM which contains GStreamer 0.6.5, along with the standard plugins and a number of OASIS-cluster specific ones. Follow the instructions in the README file, and everything should work fine. It describes the procedure both for a global and for a local install. The CD-ROM can be obtained from Jeffrey Kang or Peter van der Stok.

Another CD-ROM, prepared by Ralph Meijer, also exists. It contains a CVS snapshot of GStreamer that was checked out somewhere between version 0.6.5 and 0.7.0. I have been unable to compile it properly, though not for lack of trying. I would recommend against using it.

A third approach is to download GStreamer as a package from your favourite distribution. However, this enforces a global install and limits the possibility to extend or modify GStreamer.

It is also possible to gather and install all components manually. Note that this only works well for a global install. To do this, you require the following items:

1. The file gstreamer-0.6.5.tar.gz from the download section of the GStreamer website [4].
2. The file gst-plugins-0.6.5.tar.gz, ibid.
3. The file gst-plugins-0.7.1.tar.gz, ibid. From this file, we need the tcp plugin, which is not available in earlier versions of GStreamer.
4. Ralph Meijer's gst-plugins source tree. From this file, we need the following plugins: udp, mpegstream and rtp. These all contain modifications and additions with respect to the original versions from gst-plugins-0.6.5. See also Subsection A.6.1.

Note that the following instructions are written with a Debian distribution in mind; however, they should also work for other distributions. Steps that are specifically related to Debian are marked as such.

1. Untar all files into separate directories. We will assume these directories are called /path/to/gstreamer-0.6.5, /path/to/gst-plugins-0.6.5, etc. Substitute these with the appropriate directories. Make sure that the paths are all absolute, and not relative from the current working directory.
2. Go to the /path/to/gstreamer-0.6.5 directory.
3. Run ./configure and make
4. Become root and run make install
5. Run gst-register
6. Run gst-launch fakesrc num_buffers=5 ! fakesink to test if everything works.
7. Go to the /path/to/gst-plugins-0.6.5 directory.
8. Run configure as follows:
   PKG_CONFIG_PATH=/path/to/gstreamer-0.6.5/pkgconfig LDFLAGS="-L/usr/local/lib -L/usr/X11R6/lib" CPPFLAGS="-I/usr/local/include -I/usr/X11R6/include" ./configure --disable-qcam --disable-ffmpeg --disable-cdrom --disable-cdparanoia --disable-v4l --disable-v4l2. Install any dependencies it requires, and re-run until it exits with no errors. Figure A.1 lists dependencies that are likely to occur.
9. Run make
10. Become root and run make install
11. Go to the /path/to/gst-plugins-0.7.1 directory.
12. Open the file configure.ac and change the line

    AS_VERSION(gst-plugins, GST_PLUGINS_VERSION, 0, 7, 1, 1, GST_ERROR="-Wall", GST_ERROR="-Wall -Werror")

    into

    AS_VERSION(gst-plugins, GST_PLUGINS_VERSION, 0, 6, 0, 1, GST_ERROR="-Wall", GST_ERROR="-Wall -Werror")

13. Run NOCONFIGURE=yes ./autogen.sh. See Table A.2 for more information on getting autogen to work in Debian.
14. Run configure as follows:
PKG_CONFIG_PATH=/path/to/gstreamer-0.6.5/pkgconfig LDFLAGS="-L/usr/local/lib -L/usr/X11R6/lib" CPPFLAGS="-I/usr/local/include -I/usr/X11R6/include" ./configure --disable-qcam --disable-ffmpeg --disable-cdrom --disable-cdparanoia --disable-v4l --disable-v4l2.
15. Do *not* run make from this directory.
16. Go to the /path/to/gst-plugins-0.7.1/gst/tcp directory.
17. Run make
18. Become root and run make install
19. Go to the directory containing Ralph Meijer's gst-plugins source tree.
20. Open the file configure.ac and change the line

    AS_VERSION(gst-plugins, GST_PLUGINS_VERSION, 0, 7, 0, 1, GST_ERROR="-Wall", GST_ERROR="-Wall -Werror")

    into

    AS_VERSION(gst-plugins, GST_PLUGINS_VERSION, 0, 6, 0, 1, GST_ERROR="-Wall", GST_ERROR="-Wall -Werror")
21. Run NOCONFIGURE=yes ./autogen.sh
22. Run configure as follows:
PKG_CONFIG_PATH=/path/to/gstreamer-0.6.5/pkgconfig LDFLAGS="-L/usr/local/lib -L/usr/X11R6/lib" CPPFLAGS="-I/usr/local/include -I/usr/X11R6/include" ./configure --disable-qcam --disable-ffmpeg --disable-cdrom --disable-cdparanoia --disable-v4l --disable-v4l2.
23. Do *not* run make from this directory.
24. Go to the gst/mpegstream subdirectory.
25. Open the file gstrfc2250enc.c and add the line #include <string.h>
26. From this directory, run make and make install
27. Go to the gst/rtp subdirectory.
28. Open the file gstrtpmpegparse.c and find and remove the four lines containing the g_print statement.
29. Run make and make install
30. Go to the gst/udp subdirectory and run make and make install
31. Run gst-register --gst-plugin-path=/usr/local/lib/gstreamer-0.6

A.3 How to use GStreamer

A GStreamer pipeline can be built by writing a program in C, or in another language for which the appropriate bindings are available. How to do this, is described in GStreamer's

Plugin	Debian package
colorspace	hermes1-dev
httpsrc	libghttp-dev
mpeg2dec	libmpeg2-4-dev
xvideosink	lesstif-dev

Table A.1: Dependencies for important GStreamer plugins. Note that configure fails to mention the lesstif dependency for xvideosink.

When using autogen for the first time, it will probably complain about missing dependencies. These are fairly straightforward to install with your distro's package system. However, the version of libtool distributed by Debian at the time of writing is incompatible. You will need to download and build it manually.
If you get one or more of the following error messages:

aclocal: configure.ac: 8: macro 'AM_DISABLE_STATIC' not found in library
aclocal: configure.ac: 27: macro 'AM_PROG_LIBTOOL' not found in library

then libtool and aclocal are not in the same directory. To solve this problem, run the following command:

export ACLOCAL_FLAGS="-I /usr/local/share/aclocal"

This assumes libtool is installed in /usr/local/bin. If libtool is installed in another directory, use a corresponding prefix. See the following url for more information:

http://gstreamer.freedesktop.org/data/doc/gstreamer/head/faq/html/chapter-cvs.html#autogen-libtool

Table A.2: Getting autogen to work in Debian.

Application Developer's Manual [19]. Note that this manual is written with version 0.8 in mind,and a number of things may not work with GStreamer 0.6.

However, it is much easier to invoke GStreamer from the command line. Since GStreamer command lines tend to be long, it is advisable to write some small shell scripts that wrap the most common functions, such as play, transmit, receive.

Below, I will give some examples. I will assume the GStreamer binaries are in the search path. If not, prepend the appropriate path to the gst-launch command.

- gst-launch filesrc location="movie.mpg" ! mpeg2dec ! xvideosink
  Reads the file movie.mpg, parses the MPEG data and displays it on the screen.
- gst-launch filesrc location="movie.mpg" ! rfc2250enc ! rtpmpegenc ! udpsink host="127.0.0.1" port="4444"
  Reads the file movie.mpg, encodes and packetizes it according to the RFC 2250 and RTP protocols, and transmits it using UDP to 127.0.0.1:4444.
- gst-launch udpsrc port="4444" ! rtpmpegparse ! mpeg2dec ! xvideosink
  Listens to UDP port 4444 for an MPEG stream, and, when it arrives, decodes the RTP and RFC 2250 packets, decodes the MPEG, and displays the result on the screen.
- gst-launch udpsrc port="4444" ! rtpmpegparse ! filesink location="received.mpg"
  Listens to UDP port 4444 for an MPEG stream, and, when it arrives, decodes the RTP and RFC 2250 packets and writes the resulting MPEG stream to the file received.mpg.
- gst-launch filesrc location="movie.mpg" ! tcpsink host="localhost" port="4444"
  Reads the file movie.mpg and transmits it over TCP to localhost on port 4444. No RFC 2250 or RTP encoding is necessary, because TCP itself will take care of order and packetization.
- gst-launch filesrc location="movie.mpg" ! ifdsink host="localhost" port="4444"
  Reads the file movie.mpg and transmits it over TCP, using the IFD protocol, to localhost on port 4444.
- gst-launch tcpsrc port="4444" ! mpeg2dec ! xvideosink
  Listens to TCP port 4444 for an MPEG stream, and, when it arrives, decodes the MPEG stream and displays it on the screen.
- gst-launch tcpsrc port="4444" ! mpegstat ! fakesink sync=true
  Listens to TCP port 4444 for an MPEG stream, and, when it arrives, displays statistics on the stream. The fakesink element is necessary because mpegstat is not a sink. The sync=true (false by default) assures that synchronisation with the clock takes place.
- gst-launch httpsrc location="http://www.philips.com/movie.mpg" ! mpeg2dec ! xvideosink
  Streams http://www.philips.com/movie.mpg and displays it on the screen.

Note that these pipelines require an MPEG file containing only a video stream. If a file contains both audio and video, a demux element must be included, or a Segmentation Fault will follow.

A.4 How to install a GStreamer plugin

Sometimes, a plugin comes as a separate tarball. In that case, verify if it is written for the correct GStreamer version. One way of verifying this, is to open the configure.ac file and find the line beginning with AS_VERSION. It should look something like this:

AS_VERSION(gst-plugins, GST_PLUGINS_VERSION, 0, 6, 0, 1, GST_ERROR="-Wall", GST_ERROR="-Wall -Werror")

If the version number in the middle is 6, as it is in this example, everything is fine. If the version number is 8, the plugin will not work with GStreamer 0.6. If the number is 7, it the best way to find out if it will work is to change the number to 7 and compile it, to see if it works.

When this is done, you can simply make it, as described in the accompanying README file.

It is also possible to install a single plugin from an entire gst-plugins repository. For instance, in Section A.2 I explain how to merge the tcp plugin of GStreamer 0.7.1 into a 0.6.5 install. See Steps 11 to 18 for a description of how to do this.

A.5 How to write a GStreamer plugin

Writing a plugin is fairly well described in GStreamer's Plugin Writer's Guide [1]. It describes all you need to know to write a plugin from scratch.

However, it is of course also possible to modify an existing plugin. In this case, make sure to change the identifiers and string literals containing the plugin's name to reflect the name of your new plugin, in order to avoid conflicts.

Another possibility is to add a new element to an existing plugin, without interfering with the elements already present in the plugin. I have done this for my mpegstat element, because I wanted to reuse the MPEG parsing functions available in this plugin without extracting this functionality completely and moving it to another plugin (which I later did for the IFD plugin anyway). To do this, execute the following steps. Let's assume the new element's name is mpegstat and the plugin to add it to is mpegstream; substitute these with the names that apply to your own situation.

1. Go to the /path/to/gst-plugins/gst/mpegstream directory.
2. Add the files gstmpegstat.c and gstmpegstat.h to this directory.
3. Make sure these two files contain all required functions, with appropriate names.
4. Open gstmpegstream.c and add lines for the new element, analogous to the lines for the elements that already exist.
5. Open Makefile.am and add the new element, analogous to the elements that already exist.
6. From /path/to/gst-plugins, run NOCONFIGURE=yes ./autogen.sh

7. See whether /path/to/gst-plugins/gst/mpegstream/Makefile.in contains the name of the new element.
8. From /path/to/gst-plugins, run configure as indicated in the README file.
9. From /path/to/gst-plugins/gst/mpegstream, run make
10. If working with a global install, become root and run make install
11. Run gst-register with the appropriate parameters (see Section A.2).

A.6 GStreamer hacks

In this section, I will present a number of 'hacks' that did not fit in other sections in this appendix.

A.6.1 'unkown payload_t'

When RTP packets are spread over more than one UDP packet, the receiving instance of GStreamer produces many times the following error message: unkown payload_t: n [sic], where n is a number that is different each time the message is shown. This error occurs when the udpsink element splits GStreamer buffers it receives from the rtpmpegenc into two separate UDP packets. The RTP header, that should immediately follow the UDP header, is therefore absent in the second packet, which is the cause for the error.

This error is fixed in the GStreamer tarballs I prepared, but remains in the standard 0.6.5 udp plugin. To solve it, open the gstudpsink.c file and find the gst_udpsink_chain function. Comment out the entire for-loop that starts with the following line:

```
for (i = 0; i < GST_BUFFER_SIZE (buf); i += udpsink->mtu) {
```

Make sure that the last line of the function, gst_buffer_unref (buf);, remains intact.

A.6.2 httpsrc element speedup

A video that is streamed over HTTP using the httpsrc element does not slow down when the link's available bandwidth exceeds the movie's bitrate. To put it in other words, a stream that begins with the httpsrc element disregards timestamps. To solve this problem, execute the following steps:

1. Locate the gsthttpsrc.c file in the GStreamer plugin directory structure.
2. Find the function gst_httpsrc_get
3. At the end of this function, add the following line:

   ```
   GST_BUFFER_TIMESTAMP(buf) = -1;
   ```

4. While we're at it, we can fix a memory leak as well. Replace the following lines:

   ```
   if (readbytes == 0) {
       gst_element_set_eos (GST_ELEMENT(src));
       return NULL;
   }
   ```

   with the following lines:

```
if (readbytes == 0) {
    gst_buffer_unref (buf);
    gst_element_set_eos (GST_ELEMENT(src));
    return GST_BUFFER (gst_event_new (GST_EVENT_EOS));
}
```

A.6.3 Remove out-of-sync warning

When GStreamer buffers arrive more than two seconds late, GStreamer prints out-of-sync messages such as these:

(process:1893): GStreamer-WARNING **: abnormal clock request diff: ABS(-7810483445) > 2000000000

As these lines are printed for each buffer, they can become quite a nuisance for non-live streams, especially when working with the mpegstat element. (mpegstat prints a * after each frame that was more than two seconds late as well.)

To prevent GStreamer from printing this message, open the gstsystemclock.c file in the /path/to/gstreamer/gst directory. Toward the end of this file, in the function gst_system_clock_wait , the following two lines need to be commented out:

```
g_warning ("abnormal clock request diff : ABS(%" G_GINT64_FORMAT
           ") > %" G_GINT64_FORMAT, diff, clock->max_diff);
```

This will prevent the message from being printed.

A.6.4 Remove time synchronisation from udpsink

To remove the timestamp synchronisation from udpsink, so a stream is transmitted over UDP at top speed, instead of the speed dictated by the time stamps within the MPEG stream, open the gstudpsink.c file. In the gst_udpsink_chain function, comment out the following lines of code:

```
if (udpsink->clock) {
    GstClockID id = gst_clock_new_single_shot_id (udpsink->clock,
                                                  GST_BUFFER_TIMESTAMP (buf);

    GST_DEBUG (0, "udpsink: clock wait: %" G_GUINT64_FORMAT "\n",
                                                  GST_BUFFER_TIMESTAMP (buf);
    gst_element_clock_wait (GST_ELEMENT (udpsink), id, NULL);
    gst_clock_id_free (id);
}
```

Note that you should use a blocking kernel send buffer if you do this, or else most of the data will be lost in a buffer overflow on the sender. To do this, look up the network interface's txqueuelen value using the ifconfig tool, and write this value minus one to the /proc/sys/net/core/wmem_max file.

Bibliography

[1] Richard John Boulton, Erik Walthinsen, Steve Baker, Leif Johnson, and Ronald S. Bultje. *GStreamer Plugin Writer's Guide*. http://gstreamer.freedesktop.org/documentation.

[2] Ethereal. http://www.ethereal.com.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *RFC 2068: Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, January 1997.

[4] GStreamer. http://www.gstreamer.net.

[5] D. Hoffmann, G. Fernando, V. Goyal, and M. Civanlar. *RFC 2250: RTP Payload Format for MPEG1/MPEG2 Video*. Internet Engineering Task Force, January 1998.

[6] IEEE-SA Standards Board. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications*. IEEE, 1999.

[7] IEEE-SA Standards Board. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band*. IEEE, 2003.

[8] Iperf. http://dast.nlanr.net/Projects/Iperf.

[9] Jeffrey Kang, Harmke de Groot, Peter van der Stok, Dmitri Jarnikov, Iulian Nitescu, and Felix Ogg. Robust video streaming over wireless in-home networks. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, pages 193–212. Springer, 2005.

[10] Sergei Kozlov, Peter van der Stok, and Johan Lukkien. Adaptive scheduling of MPEG video frames during real-time wireless video streaming. In *IEEE WoWMoM2005*, pages 460–462, June 2005.

[11] Ralph Meijer. Volund: a research vehicle for networked video streaming. Master's thesis, Technische Universiteit Eindhoven, 2004.

[12] Martin W. Murhammer, Orcun Atakan, Stefan Bretz, Larry R. Pugh, Kazunari Suzuki, and David H. Wood. *TCP/IP Tutorial and Technical Overview*. IBM Corporation, International Technical Support Organization, 1998.

[13] J. Postel. *RFC 768: User Datagram Protocol*. Internet Engineering Task Force, August 1980.

[14] J. Postel. *RFC 793: Transmission Control Protocol*. Internet Engineering Task Force, September 1981.

[15] H. Schulzrinne. *RFC 1890: RTP Profile for Audio and Video Conferences with Minimal Control*. Internet Engineering Task Force, January 1996.

[16] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RFC 1889: RTP: A Transport Protocol for Real-Time Applications*. Internet Engineering Task Force, January 1996.

[17] W. Stevens. *RFC 2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. Internet Engineering Task Force, January 1997.

[18] P.D.V. van der Stok, D. Jarnikov, S. Kozlov, and I.C. Kang. Techniques for video streaming over perturbed wireless network segments. *Philips Research Eindhoven*, 2004.

[19] Wim Taymans, Steve Baker, and Andy Wingo. *GStreamer Application Developer's Guide*. http://gstreamer.freedesktop.org/documentation.

[20] George Xylomenos and George C. Polyzos. TCP and UDP performance over a Wireless LAN. In *IEEE Infocom*, pages 439–446, 2004.