Note: THIS IS JUST AN EXCERPT FOR ViTooKi!

Read Michael Kropfbergers full Ph.D. Thesis for more details!
`http://www.kropfberger.com`

CHAPTER 1 ViTooKi - The Video ToolKit

ViTooKi – the **Vi**deo-**Too**l**Ki**t [1] is an open source client/server framework which was developed at our department. It is capable of sending, adapting and displaying MPEG-1/2/4 video streams, streaming and playing MP3 audio streams, and opens container formats like `.mp4` or `.avi`. It comes with full support for adaptive standard compliant multimedia streaming and proxy caching. It consists of a multi-platform core library and various applications using this ViTooKi core library. The core library is using FFmpeg [2] and/or XviD [3] , therefore it is very open for extensions for new video and audio formats and container formats.

Other toolkit projects on video streaming are MPEG4IP [4], Helix [5] and VideoLAN [6], none of them offers the needed capabilities nor the necessary combinations of dynamic adaptation and stream switching and retransmission. They also ignore the state of the client-side buffer. Such a buffer-aware architecture was published in [7]. It adds buffers for the displaying engine to compensate decoding jitter and implements frame dropping, but does not support stream switching or packet retransmission. ViTooKi implements all of these extensions and features, which were also discussed in detail prior in this thesis.

One of the use cases for ViTooKi, and the topic of Peter Schojer's Ph.D. thesis [8], is a caching proxy which, instead of deleting outdated content to reduce the needed cache size, first tries to reduce the media size by quality reduction [9]. Other available applications are an adaptive streaming server, an MPEG-21 compliant multivideo player with configurable terminal capabilities, a multivideo transcoder, a DVD ripper and an MPEG-7 video annotation tool. Many active student projects are constantly increasing the amount of available tools, all leveraging the simple-to-use ViTooKi library. Appendix B lists some convenience functions which can be leveraged for statistical analysis, frame prioritizations and quality measurements.

This chapter will explain the ViTooKi design in detail, including the integration of all the previously introduced technologies like smooth buffer streaming, retransmissions, in-stream adaptation and stream switching. Adaptation methods are implemented as exchangeable and chainable Adaptators as described in Section **??**. In ViTooKi, those adaptor chains are used eg. on the server/proxy side to transcode videos in real-time to fit the terminal properties specified by a client (eg. the display resolution) or to adjust to changing network bandwidths. Transcoding implementations exist for the temporal (B- and P-frame dropping), spatial (resolution), quality domain (quantization) and color reduction.

1.1 Generic Streaming Environment

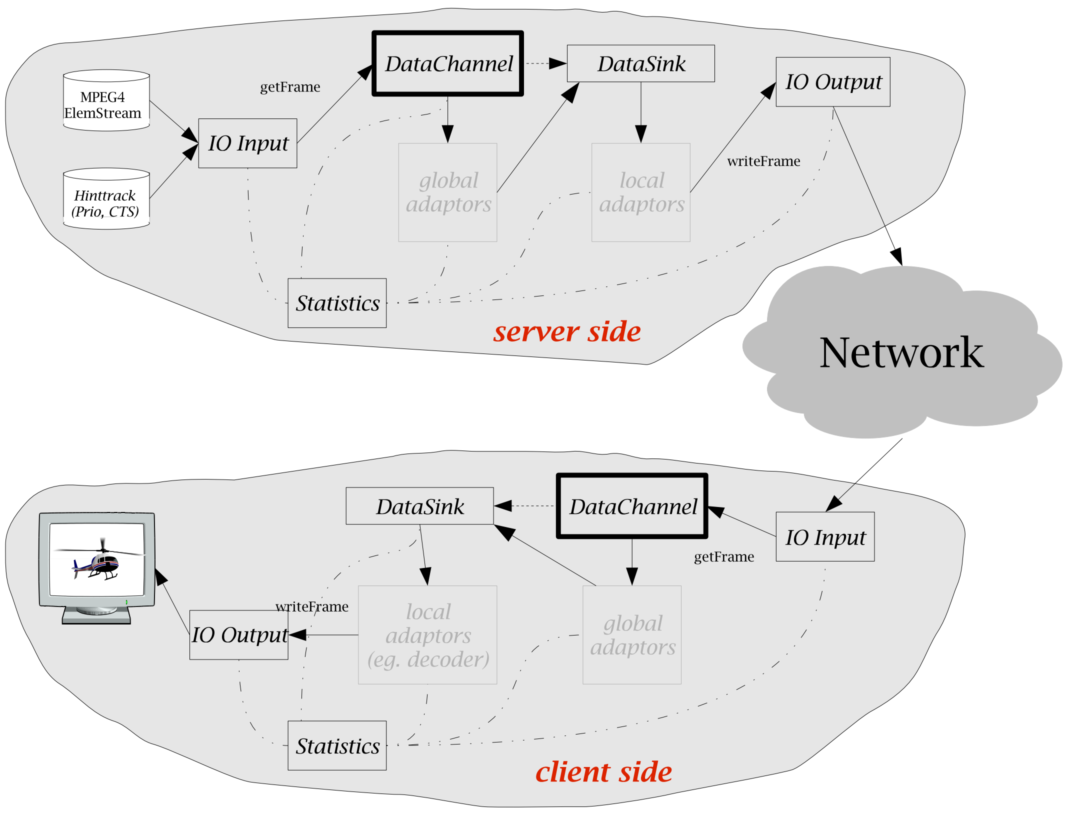

Figure 1.1: Class overview for client-server architecture in ViTooKi

Figure 1.1 shows the basic idea behind a client-server streaming architecture with all directly involved classes from the ViTooKi library. First, we will describe the server side. There is an *IO Input* Class which reads frame by frame from an MPEG-4 elementary stream by using the `getFrame()` method. Those frames are passed to the *DataChannel*, which is connected to an arbitrary number of attached *DataSinks*, which represent the open client connections.

The DataChannel passes each frame to its global adaptors, if there are any. Those adaptors might do grayscaling or could generate various statistics on by-passing frames (see Figure 1.2 for all actually available adaptors within their class hierarchies).

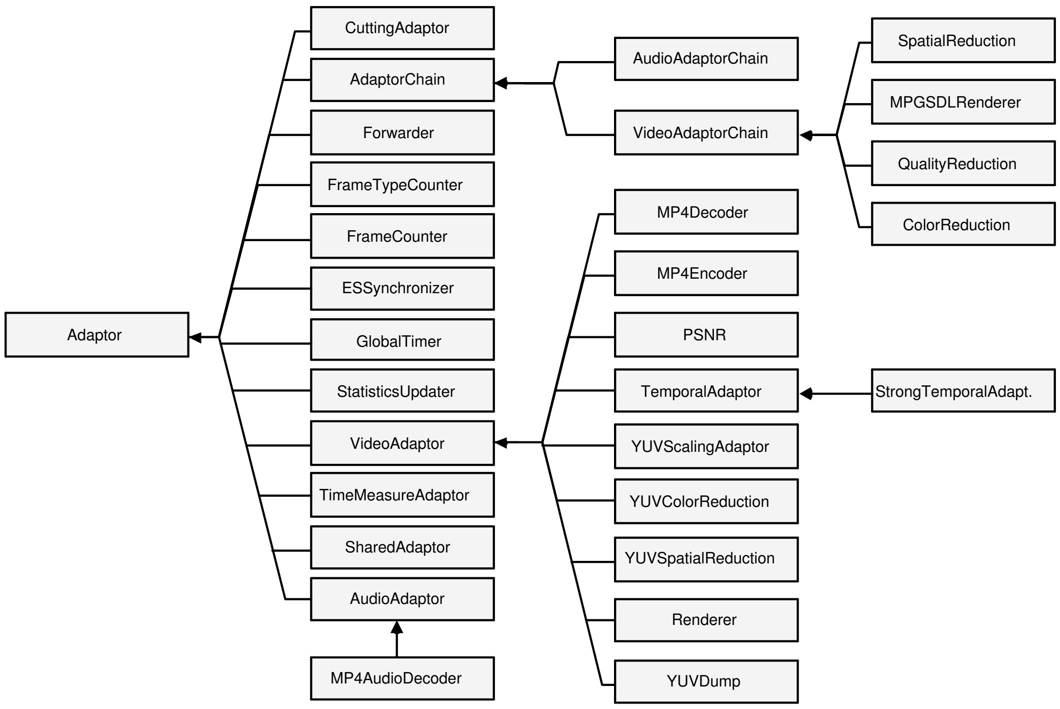

Figure 1.2: ViTooKi adaptors within their class hierarchies

A somehow adapted (but maybe also unchanged) frame is then passed to each DataSinks' local adaptors. This can be used to post-processthe frames especially for this connected client. After this "private" adaptation, the frame is passed to the *IO Output* class via the `IO::writeFrame()` method. The output can be directed to a file storage output, but in the streaming case, a network output class sends data via the network to reach a remote client.

On the client side, everything is more or less the same but inversed. The frame is received by the *IO Input* and then passed to the DataChannel via the `getFrame` method. After applying some global adaptors, the frame is handed over to the local adaptors. For a video player, this might first be a synchronization adaptor, delaying further processing according to the frame's presentation timestamp, and after that, a decoding adaptor, to transform the frame from its compressed domain into raw YUV.

This frame is processed by the adaptors and then passed to the *IO Output* class via the `IO::writeFrame()` method. *IO Output* finally displays the frame on the screen.

1.1.1 Server-Side RTP Class

The following subsections describe a specialized *IO* class which integrates RTP support according to RFC 1889 [10] into ViTooKi for both sending and receiving. Low-level RTP communication is based on the UCL Common Multimedia Library [11], which was extended by extra features like NACK/ACK feedback [12] and retransmission [13]. Those extensions are described in Section **??** and **??** and their performance was analyzed in Section **??**.

As described in the general overview of ViTooKi, every stream is handled by a *DataChannel* which runs as a seperate posix thread. On the server-side, after reading an encoded MPEG-4 frame from a file, this frame is passed to a *DataSink* with its attached Output, which is our `Rtp` class. `Rtp::writeFrame()` is then called with the encoded MPEG-4 frame. Figure 1.3 shows that this frame is packetized in the *Packetization Engine* and is, accoring to RFC 3016 [14], fragmented into seperate access units of *Maximum Transfer Unit (MTU)* size of the underlying network (eg. ethernet uses 1500 bytes). Those network packets do not only store the payload, but also such information like the frame type, presentation timestamp and frame priority for dropping behavior (see Chapter **??**). They are inserted in a list, called *preQ*, sorted by their timestamps.

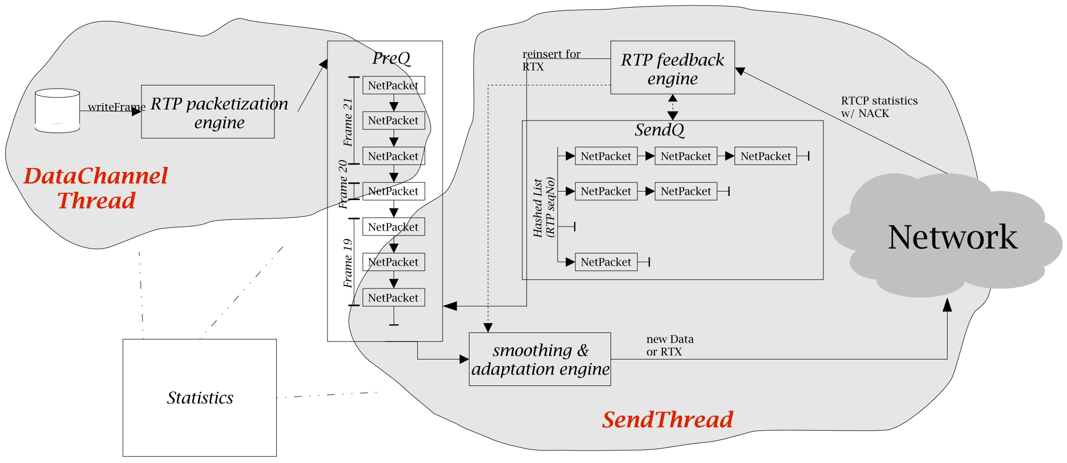

Figure 1.3: Concurrent threads at the server-side of the Rtp class

With the initialization of the `Rtp` class, a seperate *sendThread* is started. This thread always tries to pop packets out of the *preQ*. According to the packets' priority and the measured available network bandwidth (for more details see section 1.2.1), this popped packet is dropped by the *smoothing & adaptation engine* or sent out on the network. If a packet is sent out, it will be stored into the *sendQ*, which is used for possible retransmissions and statistics. Before the *sendThread* tries to pop the next packet, it sleeps a calculated amount of time. Obeying these time slots, only a certain amount of packets is sent out in one streamout second ($Ssec$). This amount will then roughly match the reqested streamout rate for this second. The exact streamout rate will not be met all times because of packet size boundaries, so the used bandwidth is always varying within one MTU (maximum transfer unit). Further, since TCP-friendly behaviour[1] is envisioned, the available bandwidth is always re-measured and the streamout bandwidth is adjusted in a AIMD fashion (additive increase, multiplicative decrease). This leads even more to a spiky streamout bandwidth and shortly delayed reaction.

The *sendThread* also checks the network for incoming RTP feedback messages. If retransmission is requested for lost packets, they are picked out of the *sendQ* and are reinserted into the *preQ*, again sorted by their timestamps, which are always lower than any other packet already stored in the *preQ*. By using the timestamps, retransmission packets are sorted among each other as well. This prioritization allows the immediate and in-time send-out of urgently needed data. Since we assume only a low need of retransmissions, we always resend requested packets if they could arrive in-time, more or less ignoring the increasing bandwidth hereof. They are not adapted within the *smoothing & adaptation engine*, because we do not want to break dependencies of partially received full frames, where eg. only one single packet is missing. This works well, since the packet loss is detected anyway and leads to future adaptation and hereby reaches network stabilization very soon.

1.1.2 Client-Side RTP Class

At the client side, there is a *recvThread* which checks for incoming RTP packets, which might be either new data or retransmitted packets (see Figure 1.4).

Every received packet is inserted into the *preQ*, according to its timestamp. The *recvThread* is keeping track of the last incoming RTP sequence number $seqNo_n$, so if a new packet arrives and it's RTP sequence number $seqNo_m, m > n + 1$ is not the direct successor ($seqNo_{n+1}$), an *immediate feedback RTCP* packet is generated and the server is informed of the missing packet(s) $seqNo_{n+1} \dots seqNo_{m-1}$. This silently ignores the fact that UDP does not guarantee in-order arrival of packets and can be solved by an introduced timer event, which waits a certain time before sending the RTCP feedback. Maybe the missing packet will arrive during this configurable period of time. Still, according to [15], receiving out of order messages is very rare in

[1]TCP-friendly means to be cooperative to other long-term high bandwidth connections like eg. `ftp`

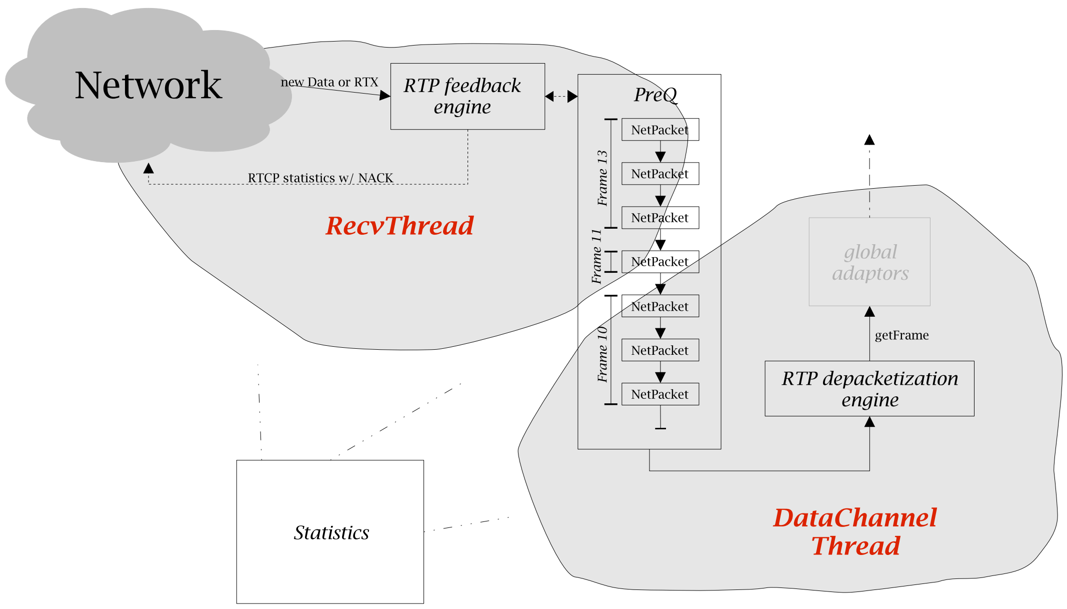

Figure 1.4: Concurrent threads at the client side of the Rtp class

practice, so receiving a message with a too high sequence number is a good sign that the expected message was unrecoverably lost.

At the same time when the *recvThread* is started, the concurrent *DataChannel* thread tries to extract the very first frame with the `Rtp::getFrame()` method. Obviously, in the beginning, no packets have arrived yet, so the *DataChannel* thread goes into *prebuffering state* and waits for a definable amount of time (normally 8 seconds). After that, it assumes, that the *recvThread* has stored a significant number of packets in the *preQ*. Then the *DataChannel* thread continues and starts to pop packets from the *preQ*. Whenever the *DataChannel* thread runs out of data again, a new *prebuffering* period is introduced, but this also means a complete blocking of the client side player and is hereby a highly unwanted scenario. To avoid this buffer underflow, server-side adaptation and switching is applied.

Since `getFrame()` has to return a full and complete (decodable) frame, we have to combine the network packets into one single frame again. This is done by reading the packets' timestamps and their final marker bit. If any of the consecutive packets is missing, the full frame is discarded and the `getFrame()` method is called again to extract the next full frame. If an error occurs at this point, it is definitely too late for retransmission, since the current frame has to be decoded and displayed next. Still, the code could be extended by some kind of forward error correction like Priority Encoding Transmission (PET) [16, 17] or even incomplete frames could be passed to the decoder in favor of independently decodable video packets based on MPEG-4 video slice bounds [18] (though this feature is neither implemented in the FFmpeg nor XviD MPEG-4 codec implementation).

1.2 Bandwidth Smoothing and Adaptation

1.2.1 Bandwidth Consumption

To avoid client buffer overrun or underrun, the server side RTP class tries to send out packets for each streamout second $Ssec_n$ with a certain streamout bandwidth $streamBW$, which was calculated according to the proposed algorithm in Section **??**.

Since this $streamBW$ is not guaranteed on best-effort networks, we first have to measure the really available $real\_netBW$. By receiving NACKs from the client side and looking up the according network packets from the $sendQ$, we can subtract the not received packet sizes from the sent packet sizes. Further, we have to make sure, that we only subtract NACKs from the according streamout second $Ssec_n$. So SSP_n denotes the set of all packets p_i in a certain $Ssec_n$ and $SSLP_n$ the set of all lost packets q_j in $Ssec_n$, where obviously $SSLP_n \subseteq SSP_n$.

$$real\_netBW_{Ssec_n} = \sum_{i=1}^{numFrm(SSP_n)} size(p_i) - \sum_{j=1}^{numFrm(SSLP_n)} size(q_j)$$

Assuming that no NACK packets are lost themselves on their way from the client to the server[2], after adding some network delay, we have a very exact knowledge of the $real\_netBW$ for $Ssec_n$. Since we have to rely on stable measurements, we wait a full second until we really take the values for further calculations.

With this knowledge of this already outdated $real\_netBW$, we want to set the new streamout rate. Generally, we could fully take the resulting (but outdated) $real\_netBW$ as the new $netBW$ used as the new streamout rate for $Ssec_{n+1}$.

$$netBW_{Ssec_{n+1}} = real\_netBW_{Ssec_{n-1}}$$

This approach would have two big disadvantages:

- Since we always only measure sent packet sizes minus data loss, we cannot detect an *increase* of available bandwidth.

- Other high bandwidth streams could starve, since we would always try to fill up the available bandwidth as much as possible. This problem arises especially

[2]Those NACK packets are sent from the client to the server, so it is the other direction, which is very likely uncongested.

in the Internet, where TCP traffic would be virtually eliminated by greedy and ruthless UDP bursts. To avoid this, we need to be *TCP friendly [19]*.

Many approaches compete to be the most TCP-friendly congestion control [20, 21, 22, 23, 24, 25]. Algorithmic complexity and environmental restrictions of the above cited TCP-friendly algorithms forced us to refrain from their usage. Find a more detailed comparison and critical analysis of different TCP friendly congestion control approaches in [26] and [27]. With the following approach, a general TCP friendliness could be achieved. Although TCP uses additive increase and multiplicative decrease (AIMD) on bandwidth adjustment, we implemented a three-level adjustment scheme which was proposed coarsly similar in [28] and [29].

The ViTooKi network implementation uses packet loss thresholds to adjust the future network bandwidth estimation. In contrast to wireless networks, on wired networks, virtually no packet loss is caused by errors on the cable. So packet loss is always a sign for congestion and we have to react by reducing our streamout bandwidth. When modern Internet routers are available on the packets' way to their destination, those routers can even drop some packets in adavance as a forewarning, whenever the fill level of their internal packet queues are getting full. This is called *random early detection (RED)* [30]. So when routers are dropping arbitrary packets in advance, the ViTooKi server is able to adjust the $netBW$ (and the data rate by adaptation) even before the network really heavily congests and hereby also avoids unnecessary retransmissions.

If the packet loss ratio is above a critical value, we decrease the $netBW$ for $Ssec_{n+1}$ by a substatial, but not too large percentage of only 20% (TCP would drop down by 50%). If packet loss is below a certain value (eg. 1%), we increase our $netBW$, which allows us to better use available bandwidth. If packet loss is steady within acceptable bounds, we keep our bandwidth steady too. Following this strategy over a longer period of time, we converge to the well-known TCP sawtooth graph for bandwidth, but with smaller drop-downs. On the other hand we are less greedy and introduce plateaus and steps. This allows us to maintain a higher and more stable average bandwidth (see Figure 1.5).

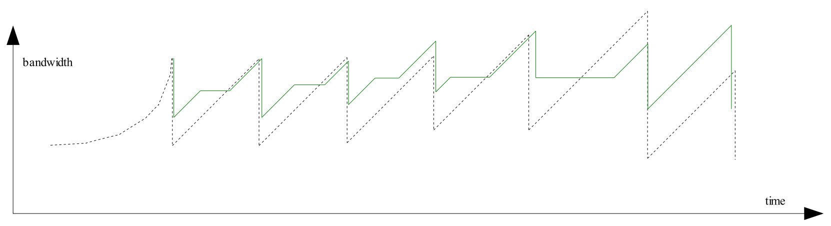

Figure 1.5: TCP sawtooth vs. our TCP friendly approach

1.2.2 Adaptation by B-frame Dropping

Calculating and adjusting the *netBW* to fit the real network bandwidth is an important issue. Still, if we cannot stream with the precalculated and needed *streamBW*, we will run out of data on the client side sooner or later. Therefore we have to adapt the data meant for the next streamout second to exactly match the *netBW*. This allows the server to send out enough packets worth for one streamout second, but with less "quality". All further streamout seconds are totally independent from this adaptation, and the basis of the pre-calculated *streamBW* will allow that the buffer bounds on the client will not be endangered by any means.

As discussed in Section **??**, the cheapest and fastest adaptation is B-frame dropping. This could simply mean not to stream out 30 frames per second with the pattern `IBBPBBPBBPBBPBBPBBPBBPBBPBBPBB`, but to drop 30% of the B-frames, by any of the previously discussed frame prioritization algorithms, where unique priority values are assigned to the frames.

1.2.2.1 Priority-based Dropping Using a Hint File

Since there are better methods for gaining good priority values than simple timely uniform distribution, these (mostly off-line calculated) priority values can be stored in special hint-files. When using good hint-files, the system knows even better which frames can be dropped without losing too much quality.

Eg. those hint-files can be generated with a quality aware algorithm (QCTVA) (see Section **??**), which works in the uncompressed domain on the basis of PSNR values. This file could also be written by an MPEG-7 based tool to annotate metadata information. Whenever streamout starts, the frames are dropped or kept according to their priority values by using the ViTooKi *smoothing & adaptation engine.*

Timely uniform distribution of dropped frames is relatively cheap in comparison to other "intelligent" off-line built prioritization algorithms and it avoids choppiness. The following example pattern nicely shows the timely uniform dropping behaviour: `IB-PB-PB-PB-PB-PB-PB-PB-PB-PB-`. This is the default prioritization algorithm within ViTooKi, if no pre-calculated priority hint-file is available. Section **??** shows, how this timely distributed dropping behaviour is simply mapped to unique priority values. The following approach always uses prioritized frames, no matter where the priorities stem from.

Since the server-side RTP class knows about all packets' priorities in the *preQ*, it is easy to extract those packets that should be sent out in the next streamout second. Further, the unimportant packets are dropped.

1.2.2.2 The Smoothing and Adaptation Engine

For adaptation within ViTooKi, the problem of different frame sizes and varying bitsizes of video seconds ($VsecBS$) has to be tackled, since those variances only average out to $streamBW$ for the full video. According to Section **??**, if the pre-calculated $streamBW$ is streamed out regularily every second, sending as many frames as possible, the client will be able to display the full stream (assuming a large enough prefetch time). One streamout second ($Ssec$) might contain parts of various $Vsecs$, depending on each $Vsec's$ bitsize $VsecBS$. For adaptation, the system has to adapt each streamout second ($Ssec$) to fit into the given $netBW$, which will finally reduce the overall average bandwidth.

According to the example in Figure 1.6, we assume a $streamBW$ of 137 kbps but the measured network bandwidth is only 110 kbps. So to fit the available $netBW$, the data has to be reduced by 20%. The frames are sorted by priorities and, in a loop, the highest priority numbers are chopped off until the $netBW$ can be sufficed. The *smoothing & adaptation engine* does not really drop frames immediately, but only marks them for dropping. When a frame should be sent out, a special *dropMe* flag stored with this frame is checked for the final decision.

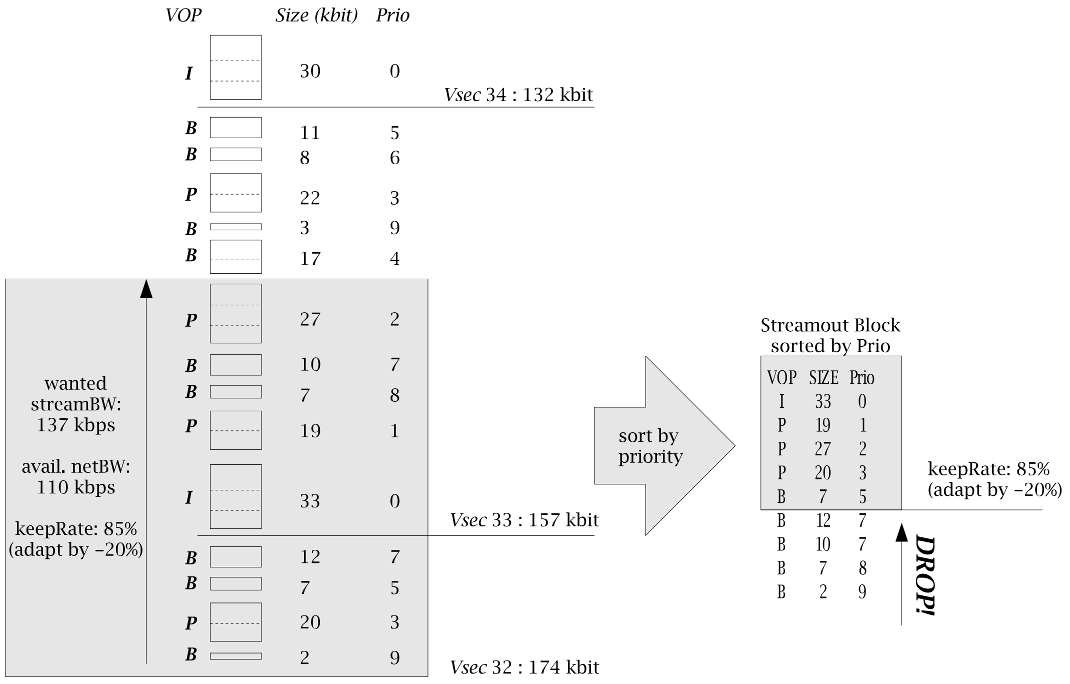

Figure 1.6: Priority-based B-frame adaptation

Please note, that the prior discussion and Figure 1.6 always assumed priority values and adaptation of full frames. Because of the underlying network, the *preQ* only stores the already demultiplexed packets at MTU size, so all full frames are split up into pieces. Since adaptation has to happen on a frame-by-frame basis, multiple packets which belong together have to be all marked for dropping. This gets even more complicated by the fact, that some frames might be already partially sent, so the remaining packets have to be excluded from adaptation to minimize wasted bandwidth. Setting up those rules, the remaining packets of already partially dropped frames have to be forcibly marked. This assures, that only full frames are either sent out or dropped.

1.2.3 Retransmission of Lost Packets

For retransmission, the NACK messages introduced in Section **??** are used by the client-side to send retransmission requests and the extension on RTP retransmission discussed in Section **??** is used by the server-side to resend important packets.

Packet loss on the client-side is detected by missing RTP sequence numbers, so a NACK packet is sent out. The server inserts the retransmission packet depending on its timestamp, which determines if it will still arrive on time. If any retransmitted packet gets lost, the client has to re-request this packet again. The loss of a retransmission packet can be identified either by a missing RTP sequence number in the retransmission stream, or – if already the according NACK request was lost – by a timer on the retransmission request. Re-requests of lost retransmission packets are done on the original stream's feedback channel. If there would be NACKs and re-retransmissions on the retransmission channel itself, the doubly sent packets would loose their original RTP sequence number themselves and could not be integrated into the original stream properly. Further they would break the RTCP network statistics since the accumulated bitrate would be an inseparable combination of normal data and retransmission packets.

Still, it is important for "fair use" bandwidth sharing or bandwidth reservations, to always see the cumulated bandwidth of the original bitrate and the bitrate needed for retransmissions. So ViTooKi has to adapt the original stream even more, in order to make room for the necessary retransmissions (retransmissions are always sent out with high priority). It is assumed, that after adapting to the newly calculated bandwidth, the network stabilizes again and the number of retransmissions is hereby also decreasing. When the newly calculated network bandwidth fits the real bandwidth again, no packet loss will occur.

1.2.4 Adaptation vs. Buffer Management

1.2.4.1 Client-Side Buffer

The server side is capable of estimating the buffer fill level of the client side, since it knows which packets (respectively their timestamps) were already sent out. There is still some uncertainty if a packet has already reached the client and was stored in the client's *preQ*. To solve that, the system needs quite adequate estimates on the network delay, which can be calculated on the basis of NACK packets for already sent packets. Exactly speaking, the packet's sendout time and the incoming NACK packet only allows calculation of the full round-trip time (RTT), but this is taken as a conservative approach. The general formula $delay = RTT/2$ does not hold anyway, if the network is congested only in one direction. But exactly this situation is very likely to happen for streaming, since the upstream channel is only used for RTCP packets whereas the high data load is congesting the downstream channel. Because of this, some unexactness is accepted and the full RTT is taken into account.

$$Buf\_ahead_n = (last\_sent\_timestamp - RTT) - Ssec_n$$

Buf_ahead_n reflects the amount of seconds of still stored data in the client buffer, because the server's $Ssec_n$ is taken as the equivalent of the client's actual playout time $Psec_n$. $last\_sent\_timestamp$ in the above formula is considered as a floating point number describing the presentation time in seconds of the currently sent out frame.

1.2.4.2 Streaming Strategies

For the following, the pre-calculated *streamBW* is used as the wanted streamout rate, whereas *netBW* depicts the really available network bandwidth, which is normally varying within a certain range around the *streamBW*. Taking into account not only the ability to adapt the data to meet a certain *netBW*, but also the previously discussed client buffer fill level, there are various choices with the available network bandwidth:

Available *netBW* exceeds *streamBW* If the available *netBW* exceeds *streamBW*, instead of only sending out *streamBW* and leave some bandwidth unused, ViTooKi sends more than *streamBW*, which will fill up the client buffer faster. Nevertheless, if the available network bandwidth is extremely high, our TCP-friendly system will not use all of it, just a little more than *streamBW* (eg. +15%). This scenario will happen on a local network, where eg. a 390 kbps video is watched by the user. An intelligent system (like ours) will not "stream" with the maximum of eg.

1000 kbps, since this would degrade streaming to normal file transfer and obviously would require enormous client buffer capacities.

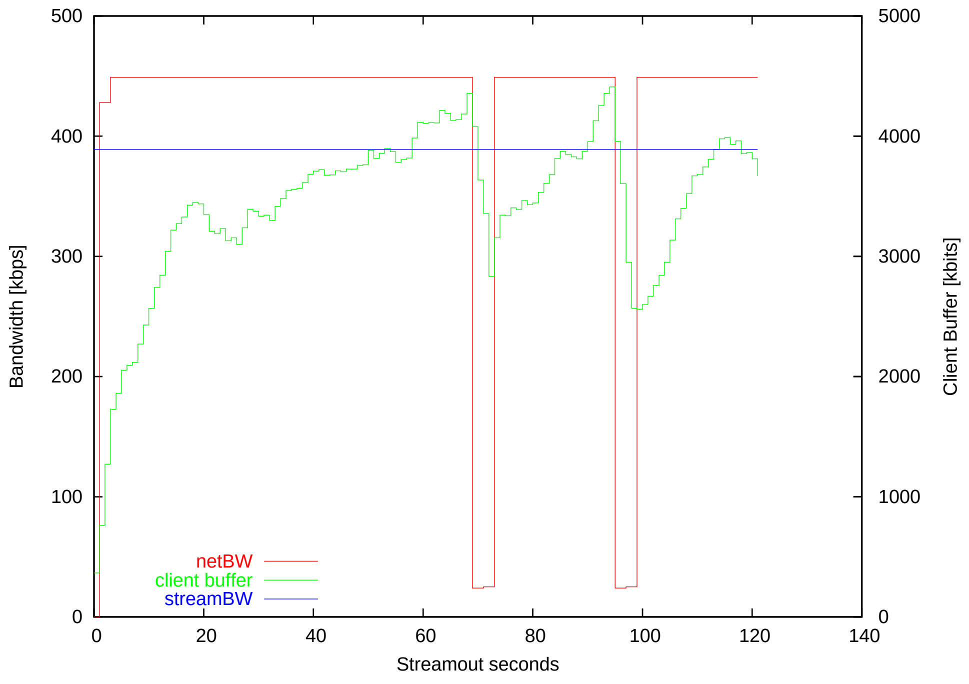

Figure 1.7: Overfull buffers force *netBW* reduction

Still, under perfect network conditions, when the *streamBW* is always exceeded even by a little, this will lead to high and overfull buffers, too. Keeping the buffer fill level always very high does not break any of the previously defined and calculated *streamBW* requirements and obviously still guarantees seamless displaying of the video. Only if the buffer fill level reaches a high-water level, the streamout rate has to be massively reduced to not completely over-fill the buffers [3].

Figure 1.7 shows a streamout scenario with a *streamBW* of 390 kbps. Since the network is never congested in this scenario, *netBW* is increased to a steady level of 450 kbps. Buffers are filled very fast and at $Ssec_{67}$ they reach the high-water level of 90%. Therefore *netBW* is reduced drastically to drain buffers again until they fall below 70% filling level. After that, *netBW* is reset to the old value of 450 kbps. The same procedure is triggered again at $Ssec_{95}$.

Even this heavy draining phase will not break the *streamBW* requirements, as long as the buffers are maximally drained by the same amount of the previously

[3]Please note, that this strategy provides more stability over possibly future network fluctuations, but an almost full buffer also introduces long delays until the buffer is played out, whenever the system performs stream switching.

extra-streamed data. So draining is safe as long the following equation holds:

$$\sum_{k=1}^{n} streamBW \leq \sum_{k=1}^{m} (streamBW + Einc_k) + \sum_{k=m+1}^{n} (streamBW - Edec_k), \forall m < n$$

In the time from $Ssec_1 \ldots Ssec_n$, instead of always streaming out $streamBW$, if $netBW$ allows it, the system first increases each $streamBW$ by some amount $Einc_k$ in the time from $Ssec_1 \ldots Ssec_m$, then the streaming rate can be dropped to $streamBW - Edec_k$ later from $Ssec_{m+1} \ldots Ssec_n$, so that the overall sum is still larger or equal to always streaming the exact $streamBW$.

Available *netBW* is below *streamBW* If $netBW$ falls below $streamBW$, and the client buffer is in a medium fill state, the system adapts to meet the $netBW$ as described in Section **??**. If the client buffer is already overly full, it just sends out unadapted data up to $netBW$ kbits. Apparently, this second approach will drain the client buffer faster but video quality is kept constant. This also envisions the fact that people are distracted by too frequent changes in quality [31]. This draining will also not break the requirements, as long as there was a high bandwidth phase before and the above formula holds.

Fill up Buffers regardless to available *netBW* To interact even more intelligently with the client buffer, the system will also adapt the data by some small percentage even if $netBW$ equals $streamBW$. This is useful whenever the client-side buffer is below a low-level watermark and buffer underrun is near. By this the system sends the exact $netBW$, but more (adapted) $Vsecs$. This will fill up the client buffer even more ($Vsec$-wise, not byte-wise) and brings back the client buffer in a stable state faster.

1.3 Switching

To prepare a multimedia presentation for *stream switching*, several quality versions of this video have to be encoded. Since the ViTooKi MuViSever is measuring the available network every second, it decides to switch down, whenever there is too much adaptation needed (eg. it is not useful to drop more than 10 fps) or the actual $netBW$ falls below 75% of the actual stream's bandwidth. It decides to switch up whenever the bandwidth is 30% above the actual stream's $streamBW$.

Whenever available bandwidth is above or below these thresholds, the `CacheManager` class, which controls all available video streams, is queried for the corresponding `MetaObject` of the actually streamed video, whereas this `MetaObject` stores all available versions of the stream, the so-called *switch set*. If another version fits better to the newly measured bandwidth, the next possible switchpoint, reflected by an I-frame, is searched. Then the old stream is continued to be streamed out until the switchpoint is reached and all old frames until then are sent out to the client (see Figure 1.8). After that, the new stream is set active, a header frame with the new decoder configuration is sent and then the server continues with the next I-frame.

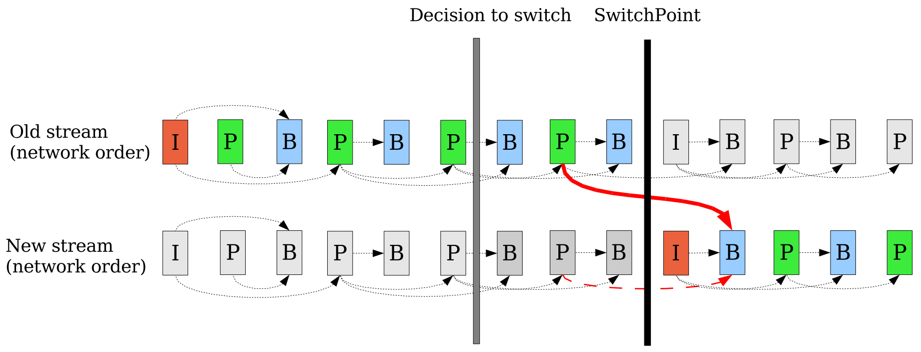

Figure 1.8: Streams are switched at the next available I-frame

This is done fully server-driven, so the client only recognizes stream switching by receiving a header frame with the new decoder configuration. MuViPlayer resets the decoder and, if the new stream is coded in a different resolution, it does the necessary up-scaling from eg. QCIF to CIF, so the user is distracted as little as possible.

As depicted in Figure 1.8, the B-frame directly after the new I-frame would need a second reference frame, which would be only available from the new stream, but there is only the old P-frame at hand, instead. Fortunately, this is not so problematic as long as both streams use the same frame pattern and the P-frames are therefore at the same positions and are coding the same visual information. Only the quantized quality differs, so in case of downswitching we get even better results when using motion vector references to higher quality macro blocks. Note, that if the P-frames are on different positions in the streams, this might lead to remarkable drift on the wrongly coded motion vectors in this B-frame. For all used *switch sets*, the coded frame patterns where the same, only quantization was changed, so no drift problems could occur.

APPENDIX A

Implementation Details

A.1 Recursive Generation of Timely Uniform Distributed Priority Values

The following algorithm is used for timely uniform distribution (see Section ??) and sets up a table with frame priorities. Basically, it implements a recursive depth-search, which is limited to a certain depth. At each depth, left and right traversals are started, which sets ascending priority numbers to the alternating tree halves.

```
int main() {
 int prioTable[patternSize], actpos=patternSize;
 int maxdepth=(int)((log((double)patternSize)/log(2.0)) + 1);
  for (int i=0; i<maxdepth; i++)
    patSplit(prioTable,patternSize,i,&actpos);
}

void patSplit(int *pat, int len, int maxdepth, int *actpos) {
 int i=1, j=1;

  if (*actpos == len) { // init table
    for (int k=0; k<len;k++)
      pat[k]=-1;
    patSplitRec(pat, len, 0, 0, 0, actpos);
  }
  while ((i!=0) || (j!=0))  {
    i = patSplitRec(pat, len, 0, maxdepth, 1, actpos);   // next node RIGHT
    j = patSplitRec(pat, len, 0, maxdepth, 0, actpos);   // next node LEFT
  }
}
```

```
int patSplitRec(int *pat, int len, int actdepth,
                int maxdepth, int gowhere, int *actpos) {
 int half = len/2, *middle = &pat[half], ret;

if (actdepth < maxdepth) {
  if (gowhere) {
    ret = patSplitRec(pat, half, actdepth+1, maxdepth, gowhere, actpos);
    if (ret == 0)
      ret = patSplitRec(pat+half, len-half, actdepth+1,
                        maxdepth, gowhere, actpos);
      return ret;
    } else {
      ret = patSplitRec(pat+half, len-half, actdepth+1,
                        maxdepth, gowhere, actpos);
      if (ret == 0)
        ret = patSplitRec(pat, half, actdepth+1, maxdepth, gowhere, actpos);
      return ret;
    }
  } else {
    if (*middle == -1) {
      *middle = (*actpos)--;
      return 1;
    } else {
      printf("FULL\n");
      return 0;
    }
  }
}
```

A.2 Extensions for RTCP-based Feedback

The discussed extension for RTCP-based feedback was implemented according to the Internet draft [12] and its performance gains were measured in Section ??. In the following, detailed information is given how this implementation was done and how it can be used by other applications, including the necessary code fragments. We have extended the *UCL Common Multimedia Library* [11] of the University College London to support RTCP Feedback. For simplicity issues, only *Immediate Feedback* and *Regular RTCP Mode* are implemented. Possible feedback messages are restricted to the *Transport Layer*, so there only might be *ACK* and/or *NACK* messages on RTP packets but not on whole frames. This might increase the amount of feedback packets, but will reduce the amount of retransmission packets, since only the needed frame fragment will be resent. Note that data packets are more likely larger than feedback packets.

A.2.1 Receiver Side

To enable and hereby automatically include and send feedback on the receiver side with every RTCP packet, the UCL idea of settable RTP behaviour options is extended:

```
rtp_set_option(session, RTCP_OPT_FB_ACK, TRUE);
rtp_set_option(session, RTCP_OPT_FB_NACK, TRUE);
```

It is the application's task to decide if *ACK* or even both *ACK & NACK* is needed and really feasible with respect to network constraints.

When at least one feedback message type is enabled, the RTP library keeps track of incoming packets and stores bitmasks of missing and arrived packets. Whenever `rtcp_send_ctrl(...)` is called, all necessary feedback packets are generated and added to one large RTCP compound packet with *RTCP Sender and Receiver Reports* and *SDES* information.

To choose between *immediate* and *regular* RTCP feedback intervals, we have extended the `rtcp_send_ctrl(...)` function:

```
rtp_send_ctrl(session, rtp_ts, NULL, RTCP_NORMAL);
rtp_send_ctrl(session, rtp_ts, NULL, RTCP_IMMEDIATE);
```

If the option `RTCP_NORMAL` is chosen, the call might even return without sending anything, since the maximum RTCP bandwidth might be reached or the minimum time interval is not yet over (here, the five seconds of minimum interval and five percent of overall bandwidth apply). When set to `RTCP_IMMEDIATE`, all gathered information is sent out immediately, regardless of any restricting RTCP sending statistics and rules.

Explicit NACKs With the above described behaviour, the UCL library only sends NACKs (and ACKs) once, whenever `rtcp_send_ctrl(...)` is called. Since RTP is normally used over UDP, also NACKs could be lost. To allow an application to selectively re-request already NACKed packets, we introduced the following function:

```
rtp_send_explicit_nack(session, ssrc, rtp_seq, rtp_ts, NULL);
```

This immediately generates an RTCP receiver report and adds one NACK request for the given RTP sequence number, but does not send it yet. So again, a call of

```
rtp_send_ctrl(session, rtp_ts, NULL, RTCP_IMMEDIATE);
```

is needed, to send the RTCP receiver report with the explicit NACK.

A.2.2 Sender Side

If the receiver sends extended feedback messages (ACK and/or NACK), the sender has to somehow parse them to react accordingly.

Since the RTCP feedback is included in the compound RTCP packets, we only have to extend our applications' `RtpCallback` routine, which is called by the library, whenever new packets arrive. The library already splits the compound packet and calls the `RtpCallback` for every message part from the packet. We only had to introduce a new message type: `RX_FB`. Table A.1 shows an example implementation. Note that ACK and NACK packets are two consecutive calls to our `RtpCallback` routine! In eg. the ACK case, if one packet is not ACKed (if the bit is not set), we cannot tell right there, if this really means that a packet was lost. We have to compare it to the NACK bit of the according RTCP feedback packet.

The following example code (Table A.2) shows how to parse the feedback message and is called from the above `RtpCallback` routine. Basically, in the NACK case, the `for` loop runs through the bitmask. If there was a set bit (1), it writes out the character `D` for *Dropped*. In the ACK case, we have to first check the *Range* bit. Accordingly, we write out the following number of ACKed packets or the bit mask again.

```
void Rtp::RtpCallback(struct rtp *session, rtp_event * e) {
  rtcp_fb         *fb;

  switch (e->type) {
  case RX_RTP:          /* RTP data packet */
    printf("RX_RTP SSRC = 0x%08x\n", e->ssrc);
    /* do something */
    break;
  case RX_SR:             /* sender report */
    printf("RX_SR SSRC = 0x%08x\n", e->ssrc);
    /* do something */
    break;
  case RX_RR:           /* receiver report */
    printf("RX_RR SSRC = 0x%08x\n", e->ssrc);
    /* do something */
    break;
  case RX_FB:           /* feedback report */
    printf("RX_FB SSRC = 0x%08x\n", e->ssrc);
    fb = (rtcp_fb*)e->data;
    Rtp::fb_print(session,e->ssrc, fb);
    free(e->data);
    break;
  default:
    break;
  }
}
```

Table A.1: Example of `RtpCallback` routine with RTCP feedback support

```
void Rtp::fb_print(struct rtp *session, uint32_t ssrc, rtcp_fb *fb) {
  int i;

  printf("RX_FB SSRC = 0x%08x  ", ssrc);
  switch (fb->subtype) {
  case RTCP_FB_FMT_NACK:
    printf("NACK BLP from %6i: ",fb->fci.fb_ack.pid);
    for (i=0; i < 15; i++) {
      if ((fb->fci.fb_nack.blp & ((uint64_t)1 << i)) != 0) {
        printf("D");                              /* Dropped */
      } else {
        printf(".");          /* unknown (might be ACKed) */
      }
    }
    break;
  case RTCP_FB_FMT_ACK:
    printf("ACK  BLP from %6i: ",fb->fci.fb_ack.pid);
    if (fb->fci.fb_ack.r == 1) {                    /* range */
      printf(" to %6i",fb->fci.fb_ack.blp+fb->fci.fb_ack.pid);
    } else                                         /* bitmap */
      for (i=0; i < 15; i++) {
        if ((fb->fci.fb_ack.blp & ((uint64_t)1 << i)) != 0) {
          printf("r");                           /* received */
        } else {
          printf(".");        /* unknown (might be NACKed) */
        }
      }
    break;
  default:
    printf("FATAL: unknown Feedback Format!");
    ::exit(1);
  }
  printf("\n");
}
```

Table A.2: Example of RTCP feedback message parsing

APPENDIX

B ViTooKi Convenience Functionality

The following section will describe some ViTooKi convenience functions which can be used for better analysis of adaptation methods or received quality.

B.1 Statistics Dumps

When the *MuViServer* and *MuViPlayer* are used in the debug build, they create some output files for each instantiated *Statistics* object. The statistics class distinguishes between

- *streamout seconds*, where data comes in at any IO-Input before it is buffered or processed, and
- *playout seconds*, where the data is played out (displayed) to the client, sorted and ordered by their associated timestamps.

This allows a distinct analysis of streamed bandwidth or frame rate, and the finally stored or displayed frames.

Streamout statistics are stored in the files `rtp_stat-streamout-server-`N and `rtp_stat-streamout-client-`N resp., where N is an ascending number, reflecting the actual counter of the instantiated `Statistics` class objects. *Playout* statistics are stored in the files `rtp_stat-playout-client-`N and `rtp_stat-playout-server-`N, again with N as the ascending instance counter.

The following lists give an overview of how to interpret the file contents. All bandwidth or buffer values are in Kilobytes.

- `Statistics::writeStreamoutSecServerStats`
  writes server-side streamout statistics as tab-seperated values to `rtp_stat-streamout-server-`N, and is called every second.

  - the according streamout second,
  - *netBW* (what we can stream out in this second)
  - data bandwidth, which was sent out in one single attempt
  - retransmitted data bandwidth
  - NACKed bandwidth
  - accumulated real *Ssec* bandwidth (dataBW+RtxBW-NackedBW), will slightly differ with *netBW* because of coarse-grained packet sizes
  - packet loss in percent
  - estimated actual PlayoutSec, which is always behind the streamoutSec
  - estimated amount of *Vsecs* in client buffer
  - rate of adaptation in percent
  - size of adaptation window in *Vsec* seconds (normally 1.0)
  - estimated client buffer fill level in Kilobyte

- `Statistics::writeStreamoutSecClientStats`
  writes client-side streamout statistics to `rtp_stat-streamout-client-`*N*.

  - the according streamout second,
  - data bandwidth, which was received at first attempt
  - received retransmission bandwidth
  - accumulated real *Ssec* bandwidth (dataBW+RtxBW)
  - actual PlayoutSec, which is always behind the streamoutSec
  - amount of *Vsecs* in client buffer
  - estimated client buffer fill level in Kilobyte
  - real client buffer fill level in Kilobyte

  Note, that client streamout statistics do not have any NACKed bandwidth sizes, since the client only recognizes missing packets, but not their sizes.

- `Statistics::writePlayoutSecStats`
  is used for both, server-side (`rtp_stat-playout-server-`*N*) and client-side playout statistics (`rtp_stat-playout-client-`*N*) and writes the following values:

  - the according playout second,

- the undecoded FPS
- the really decoded FPS, eg. after ESSync dropping
- data bandwidth, which arrived at first attempt
- retransmitted data bandwidth
- NACKed bandwidth
- accumulated real bandwidth (dataBW+RtxBW-NackedBW) of this $Vsec$
- average PSNR of this second

Note again, that client playout statistics dont have any NACKed bandwidth sizes, since the client only recognizes missing packets, but not their sizes.

B.2 Priority Files

As described earlier, the adjustment of the *streamBW* to available *netBW* has to be done by adaptation. To drop B-frames by various prioritization algorithms, ViTooKi allows the storage of *priority files*.

When an `.mp4`-file is opened with the *MuViServer* for demuxing and streaming, all demultiplexed elementary streams are stored to the `demux` directory with the following file names, where the trailing numbers are the elementary stream Ids:

```
bsboth.mp4
   demux/bsboth.mp4.conf
   demux/localfile_bsboth.mp4.video.1
   demux/localfile_bsboth.mp4.video.1.hint
   demux/localfile_bsboth.mp4.audio.4
   demux/localfile_bsboth.mp4.audio.4.hint
```

To generate the `.hint`-files, the `ffMP4IO` and `isoMP4IO` classes store timestamps, frame types, but also frame priorities with each issued method-call to `MP4IO::getFrame()`. For predefined priorities, it is searched for a file in the same directory as eg. `bsboth.mp4`, which has to be named similarily like the demuxed elementary stream: `bsboth.mp4.video.1.prio`. The file format is defined as follows:

```
Prio 0  Frame 0 Type I Bytes 3842
Prio 0  Frame 1 Type P Bytes 2129
Prio 13 Frame 2 Type B Bytes 907
Prio 21 Frame 3 Type B Bytes 1039
Prio 8  Frame 4 Type B Bytes 1003
Prio 22 Frame 5 Type B Bytes 912
Prio 0  Frame 6 Type P Bytes 1914
Prio 16 Frame 7 Type B Bytes 835
...
```

Note that reference frames (I- and P-frames) have a priority value of zero, which denotes their high importance.

If no `.prio`-file is found, priority values are calculated on-the-fly in `MP4IO::getFrame()` according to the proposed timely uniform distribution scheme, where each I- and P-frame is assigned a zero-priority, whereas B-frames are prioritized by fast table lookups according to the proposed algorithm for timely uniform distribution priority values.

B.3 PSNR Calculation

For various measurements and quality-wise comparisons of different adaptation algorithms, it is important to compare the displayed video with the very original video, which was used at creation (encoding) time.

Whenever *MuViPlayer* is used for video playback of eg. `bsboth.mp4`, it looks for a locally stored file called `localfile_bsboth.mp4.video.1.yuv`, which is (or is pointing to) the very original YUV video[1].

If such a file is found, each really displayed and decoded frame is further compared to the original frame and its PSNR value is calculated. This also works in the special case, where frames were dropped or were undecodable/unavailable, because the previously available frame is simply "replayed" to the PSNR comparison adaptor.

As mentioned above, the *Statistics* class is storing average PSNR values for *playout* seconds and they are dumped to the output files. This PSNR calculation is implemented as an `UncompressedVideoAdaptor`, where the original YUV-stream is passed to the constructor as a `YUVStreamIO` object.

B.4 YUVDump Adaptor

Another `UncompressedVideoAdaptor`, which is integrated into the *MuViPlayer*, is the `YUVDump` adaptor. Whenever a video is chosen for playback and PSNR calculation is invoked, each by-passing YUV-frame is stored to a file called `localfile_bsboth.mp4.video.1.dump.yuv`.

[1]The very original is different from the already encoded and then decoded video without applied adaptation! The already encoded and then decoded video will give infinite values, since this compares the very same streams (at least for the non-dropped frames).

Again, missing frames are compensated by "replaying" and doubly storing the previous frame multiple times, so the dumped YUV-stream has the same length and framerate as a decoded and unadapted stream would have.

Bibliography

[1] Michael Kropfberger and Peter Schojer, “ViTooKi – The Video ToolKit”, http://ViTooKi.sourceforge.net.

[2] “FFmpeg”, http://ffmpeg.sourceforge.net/.

[3] “XviD”, http://www.xvid.org/.

[4] “MPEG4IP”, http://mpeg4ip.sourceforge.net/.

[5] “Helix Community”, https://helixcommunity.org/.

[6] “VideoLAN”, http://www.videolan.org/.

[7] Haifeng Xu, Joe Diamand, and Ajay Luthra, “Client architecture for MPEG-4 streaming”, IEEE Multimedia, vol. 11, pp. 16–23, April-June 2004.

[8] Peter Schojer, *QBIX-G - A Quality Based Intelligent Proxy Gateway*, Ph.D. thesis, University of Klagenfurt, 2004.

[9] Peter Schojer, Laszlo Böszörmenyi, and Hermann Hellwagner, “QBIX-G - a transcoding multimedia proxy”, Tech. Rep. TR/ITEC/04/2.16, University of Klagenfurt, August 2004.

[10] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RFC 1889: RTP: a transport protocol for real-time applications”, January 1996.

[11] University College London, “UCL common multimedia library”, http://www-mice.cs.ucl.ac.uk/multimedia/software/common/.

[12] Joerg Ott, Stephan Wenger, Noriyuki Sato, Carsten Burmeister, and Jose Rey, “Extended RTP profile for RTCP-based feedback (RTP/AVPF)”, draft-ietf-avt-rtcp-feedback-04.txt, October 2002, expires April 2003.

[13] Jose Rey, David Leon, Akihiro Miyazaki, Viktor Varsa, and Rolf Hakenberg, “RTP retransmission payload format”, draft-ietf-avt-rtp-retransmission-04.txt, December 2002, expires May 2003.

[14] Y. Kikuchi, T. Nomra, and S. Fukungaga, “RFC 3016: RTP payload format for MPEG-4 audio/visual streams”, November 2000.

[15] Mike Piecuch, Ken French, George Oprica, and Mark Claypool, “A selective retransmission protocol for multimedia on the Internet”, Proceedings of SPIE International Symposium on Multimedia Systems and Applications, Boston, November 2000.

[16] Christian Leicher, “Hierarchical encoding of MPEG sequences using priority encoding transmission (PET)”, Tech. Rep. TR-94-058, International Computer Science Institute, Berkeley, CA, November 1994.

[17] Andres Albanese and Giancarlo Fortino, “Robust transmission of MPEG video streams over lossy packet-switching networks by using PET”, Tech. Rep. TR-99-014, International Computer Science Institute, Berkeley, CA, June 1999.

[18] Rob Koenen, “Profiles and levels in MPEG-4: Approach and overview”, Image Communication Journal. Tutorial Issue on the MPEG-4 Standard, vol. 15, no. 1-2, January 2000.

[19] Sally Floyd and Kevin Fall, “Promoting the use of end-to-end congestion control in the Internet”, IEEE/ACM Transactions on Networking, August 1999.

[20] Shanwei Cen, Calton Pu, and Jonathan Walpole, “Flow and congestion control for Internet streaming applications”, Tech. Rep. CS-97-03, Oregon Graduate Institute of Science and Technology, 1998.

[21] Reza Rejaie, Mark Handley, and Deborah Estrin, “RAP: An end to end rate-based congestion control mechanism for realtime streams in the Internet”, Proceedings of IEEE Infocom, New York, March 1999.

[22] YoungGook Kim, JongWon Kim, , and C.-C. Jay Kuo, “Smooth and fast rate adaptation mechanism (SFRAM) for TCP-friendly Internet video”, Proceedings of the International Packet Video Workshop, Sardinia, Italy, May 2000.

[23] Dorgham Sisalem and Henning Schulzrinne, “The loss-delay based adjustment algorithm: A TCP-friendly adaptation scheme”, in *Proceedings of NOSSDAV*, Cambridge, UK., 1998.

[24] Dorgham Sisalem and Adam Wolisz, “LDA+: A TCP-friendly adaptation scheme for multimedia communication”, in *IEEE International Conference on Multimedia and Expo (III)*, 2000, pp. 1619–1622.

[25] Michael Welzl, *Scalable Performance Signalling and Congestion Avoidance*, Ph.D. thesis, Technische Universität Darmstadt, 2002.

[26] Jörg Widmer, Robert Denda, and Martin Mauve, "A survey on TCP-friendly congestion control", IEEE Network, vol. 15, no. 3, pp. 28–37, 2001.

[27] Ingo Buchbauer, "Flusskontrolle und QoS-Feedback für Multimedia-Streaming-Protokolle", M.S. thesis, University of Klagenfurt, May 2003.

[28] I. Busse, B. Deffner, and H. Schulzrinne, "Dynamic QoS control of multimedia applications based on RTP", First International Workshop on High Speed Networks and Open Distributed Platforms, St. Petersburg, Russia, June 1995.

[29] Yeali S. Sun, Fu-Ming Tsou, and Meng Cheng Chen, "Predictive flow control for TCP-friendly end-to-end real-time video on the Internet", Computer Communications, pp. 1230–1242, August 2002.

[30] Sally Floyd and Van Jacobson, "Random early detection gateways for congestion avoidance", IEEE Transactions on Networking, vol. 1, no. 4, pp. 397–413, August 1993.

[31] Reza Rejaie, *An End-To-End Architecture for Quality Adaptive Streaming Applications in the Internet*, Ph.D. thesis, University of Southern California, December 1999.