University of Trento, dept. of Computer Science and Engineering

An RTP chunkiser for PeerStreamer

Davide Kirchner

October 17, 2014

Contents

1 Introduction

PeerStreamer (`http://peerstreamer.org`) is an open source software that provides a framework for peer-to-peer media streaming. It includes a streaming engine for the efficient distribution of media streams, which takes care of building and management of the P2P overlay topology. It also comes with a source application for the creation of channels and a player application to visualize the streams.

The distribution of the media content is based on so-called *chunks*. A chunk is the smallest data unit that can be transmitted via the P2P network, requested by a peer or re-transmitted due to transmission errors. The transmission of a media stream is thus managed by first splitting the stream in multiple chunks: each chunk is then sent to the P2P network, which will handle the distribution of the information to all the nodes.

In this architecture, it is crucial to correctly balance the chunk size: whereas too small chunks will result in additional overhead in the chunk negotiation phase, bigger chunks will cause a larger delay in the transmission of the data, because frames and audio samples will be cashed at transmitter for a longer time. If this might not be a problem when transmitting a recorded file, for real-time streaming applications it is crucial to keep delay under control.

The management of the input stream for PeerStreamer and the creation of the chunks is handled by the *chunkiser* module of the GRAPES (*Generic Resource-Aware P2P Environment for Streaming*) library (`http://peerstreamer.org/GRAPES/`). The library is modular and provides the possibility to configure at run-time which of the available chunkisers should be used.

1.1 Motivations

PeerStreamer currently has the possibility to get an input stream from a particular satellite receiver, but lacks the capability of grabbing and distributing the stream from a webcam or other capturing devices. This feature would be needed in order to run a conference call over the P2P network, but the effort of supporting multiple and non-homogeneous hardware is not feasible.

However, in the free and open source software landscape many solutions exist that solve this problem and if PeerStreamer could take full advantage of these solutions it could gain an important added value. One way of taking advantage of external software is by exploiting the capability of such software to produce and handle RTP streams: this approach was chosen because it provides a frontend software-agnostic and encoding-agnostic way of managing the stream in the P2P network, and delegates to external software the issues of managing the source and destination streams, allowing any streaming application to be run on top of the P2P topology provided by PeerStreamer. A sample use case is represented in figure 1.1

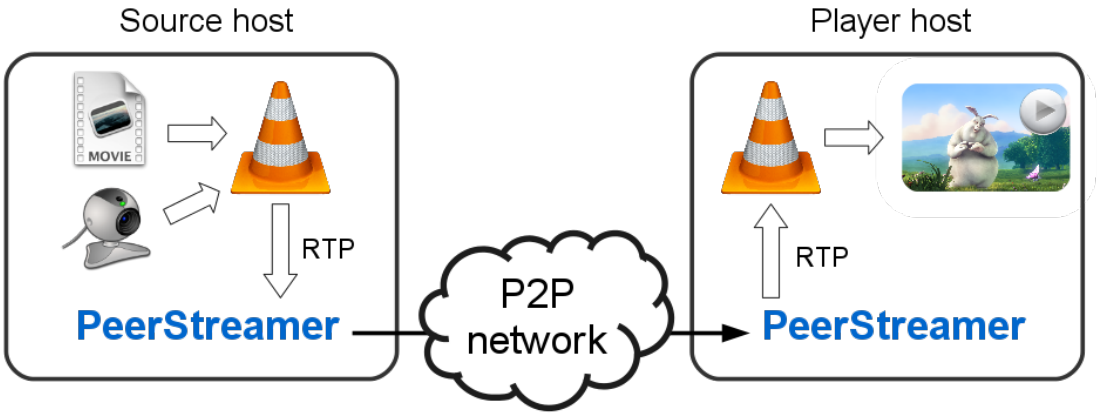

Figure 1.1: Sample use case for the RTP chunkiser.

1.2 The RTP protocol

RTP, the *Real-Time Transport Protocol*, was initially standardized in 1996 with the RFC 1889[3], later obsoleted by RFC 3550[1] in 2003, which is the current standard and was thus used as a reference in this work. Recently, RTP has gained popularity thanks to the growth of Voice-over-IP systems, which often use RTP for media transfer. RTP provides a general framework for real-time data streaming, providing the basic features for implementing full-featured application-level protocols on top of it. This section describes the basic structure of the protocol, focusing on the restricted part of the standard that is relevant to the development of the chunkiser.

An RTP data flow, also called *session* in the RFCs and throughout this document, is composed of 2 interdependent packet streams: one is used for the actual data, while the other is reserved for control messages following the RTP Control Protocol (RTCP).

1.2.1 Transport protocol

Even if RTP itself does not mandate a particular underlying transport protocol, the most common way of using it is on top of UDP protocol. Because this is by far the most common scenario, the chunkiser is built on the assumption that UDP is used: the same will be assumed in the rest of this document.

A single RTP session will thus use two coupled UDP ports: one is for RTP data, the other for RTCP control messages. The standard suggests that two subsequent UDP ports, with the first being even, are to be used for the RTP data stream and its corresponding RTCP control stream[1, p. 68]. Moreover, the standard states that multimedia communications should use a separate RTP session for each medium[1, p. 16]: thus, a common audio/video stream will use 2 RTP sessions, each with data and control streams, resulting in the need of 4 UDP ports.

RTCP messages are periodically sent by each sender to share some information about itself and about the stream; receivers, instead, can optionally use RTCP to report about

the reception quality and jitter statistics.

1.2.2 Timing

The part of the protocol that is more relevant for the chunkiser is the management of timing in the real-time stream. Each RTP data packet contains a 32-bit time-stamp "derived from a clock that increments monotonically and linearly in time"[1, p. 14]; note that this time-stamp is not required to be coherent across multiple RTP sessions, nor to be aligned to an absolute clock. However, a coherent time scale is required for allowing the receivers to align parallel sessions and (if reproducing in real-time) to playout the frames at the correct time: for this purpose, each RTCP packet sent by the source must contain a Sender Report (SR) sub-packet. Among other data, the SR holds a mapping between a standard NTP time-stamp and the corresponding value in RTP time-stamp units[1, p. 37].

1.2.3 Chunkiser and dechunkiser in the RTP flow

In this context, PeerStreamer will act as a virtual network layer, carrying UDP datagrams over the distribution network. However, the RTP-aware chunkiser will inspect the stream in order to take better decisions when dealing with the aggregation of multiple datagrams in the same chunk.

Note that the PeerStreamer distribution system does not currently provide the possibility to transfer data backwards across the P2P network towards the stream source. For this reason, there is no way to carry or generate RTCP Reception Report (RR) packets to inform the source about the reception quality.

2 Features and implementation

A chunkiser for PeerStreamer is built of three main callbacks: an **`open`** function called before any operation which has access to command-line configuration, a **`chunkise`** function which is periodically polled for new chunks to be sent, and a **`close`** function for cleanup and graceful closing. All these functions have access to a single shared instance of a **`struct chunkiser_ctx`**, which is to be defined by the chunkiser itself. The dechunkiser works in an analogous way, except that the **`dechunkise`** function is only called when a chunk is received.

The next sections define the main features of the RTP chunkiser and dechunkiser, and provide high-level descriptions of the respective implementations.

2.1 Stream [De]Multiplexing

Because a standard RTP audio/video stream is made of 4 UDP streams (see section 1.2.1), the chunkiser should be able to multiplex multiple packets of multiple streams on top of one chunk stream.

This is achieved with the simple chunk structure described in Fragment 2.1, which is similar to the one implemented in the plain UDP chunkiser: an arbitrary number of UDP datagrams can be carried into one chunk, each tagged with its length and an identifier of the UDP stream it belongs to. Note that it is not needed to store the number of embedded packets, because the total chunk length will be carried by the underlying chunk distribution system, resulting in a dechunkising procedure outlined in fragment 2.2.

From the user's perspective, at run-time the proper configuration for the multiplexing must be given at both chunkiser and dechunkiser sides; for convenience, the command line syntax is identical at each side, the only difference is that the chunkiser will *listen* for the streams on the given ports, while the dechunkiser will generate traffic *towards* the given ports. Note that, if using the same port specification command-line for invoking both the chunkiser and the dechunkiser, they are guaranteed to correctly associate input and output ports.

The most direct and powerful way to provide input ports is with command line arguments **`stream0`** through **`stream<N-1>`** (with N configurable at compile time and currently set to 5). Each of these arguments accepts as valid input either a colon-separated pair of UDP port numbers, or a single one. In the former case, the first is intended as the RTP data port and the second its corresponding RTCP port. In the latter case, the port should be *even* and is interpreted as the RTP data port number, with the subsequent port number being used as the corresponding RTCP port. In compliance with the standard's recommendations[1, p. 68], in case an *odd* port number is supplied, it is instead interpreted as the RTCP port, and the previous one will be used for RTP data.

In case a pair of audio/video RTP sessions is to be carried, it is convenient to use the **`audio`** and **`video`** command-line arguments: this is the most intuitive way to provide inputs when reading from an SDP stream specification. This is also the preferred way to specify ports when using the **`rfc5331`** feature (see section 2.4), because the video stream is explicitly marked as such.

The last way to specify input or output ports is with the **`base`** command-line argument: it takes as value one single port number **`N`**, and uses ports **`N:N+1`**, **`N+2:N+3`**, pair-wise interpreted as port for the same RTP session. This format is convenient because it resembles the input format of VLC video player when streaming RTP.

Note that the three ways of providing input or output ports are mutually exclusive, and in case of conflicts **`streamN`** options will have priority over **`audio`** and **`video`**, which have priority over the **`base`** option.

```
  0    1    2    3
  +----+----+----+-------------
  |  size   |s_id| data (UDP payload, with RTP header)
  +---------+----+-------------
  |  ...
  +---------+------------------
  |  size   |s_id| data ...
  +---------+----+-------------
  size: size of the following `data` field (16 bits)
  s_id: stream identifier (8 bits)
```

Fragment 2.1: Chunk payload format for carrying RTP packets.

```
void rtp_dechunkise(struct dechunkiser_ctx *ctx, ...,
                    uint8_t *data, int size) {
  int stream, psize;
  uint8_t* data_end = data + size;
  while (data < data_end) {
    rtp_header_parse(data, &psize, &stream);
    data += RTP_HEADER_SIZE;
    send_udp(ctx->outfd, ctx->ip, ctx->ports[stream], data, psize);
    data += psize;
  }
}
```

Fragment 2.2: Pseudo-C code for the dechunkising procedure.

Concerning the implementation at **`open`** time, as the command-line arguments are parsed, the corresponding context variables get populated. The dechunkiser will simply save the port numbers in the **`ports`** array, and its length in **`ports_len`**. The chunkiser, instead, needs to keep track of the RTP sessions and start listening, while it does not need to store the port numbers after starting to listen: thus, file descriptors for open ports are stored.

2.2 Maximum chunk size

The basic feature of the chunkiser, and the one that is expected to significantly lower the overhead of running RTP over the peer-to-peer network, is the possibility to multiplex many consecutive RTP packets in a single chunk.

One of the options for achieving this is by specifying a chunk size. Note that, in order to avoid to waste resources in dynamic memory re-alloccation or movement of data across buffers, a maximum chunk size is needed by the chunkiser to determine the size of the memory to be alloccated: once the target chunk size C_{target} is set, the maximum chunk size is computed as $C_{max} = C_{target} + \text{max\_UDP\_size}$. Then, the chunk will be sent as soon as its size exceeds C_{target}: this way, every time a UDP datagrams arrives it is guaranteed to fit the remaining space in the chunk.

Using the command-line option **`chunk_size`**, the value (in bytes) of the target chunk size can be set.

2.3 Maximum delay

The chunkiser is able to take the decision about the number of packets that are to be carried in one chunk based on their timing: when all time-stamps can be determined, a chunk will be sent when $t_{max} - t_{min} >$ max_delay. The maximum delay can be set using the command line parameters **max_delay_ms** or **max_delay_s**, both accepting integer values.

2.3.1 Packet time-stamp retrieval

Note that retrieving the time-stamp from an RTP packet is non-trivial: each RTP packet only holds a 32-bit time-stamp which, according to the standard, is *only* required to be "derived from a clock that increments monotonically and linearly in time"[1, p. 14]: there is no notion of seconds in that time-stamp and the audio and video streams of the same file are allowed to use different time scales. Thus, for the purposes of determining the wall-clock time, the RTP time-stamp must be converted to an NTP-formatted one, using the information carried in the associated RTCP stream (see section 1.2.2).

```
   a         b     c <- incoming RTP pkt
---|---------|-----*--> RTP timeline
---|---------|-----?--> NTP timeline
```

Thus, assuming the two most recent RTCP packets provide the association between a_r and a_n and between b_r and b_n, the NTP time-stamp c_n of the incoming RTP packet with time-stamp c_r can be computed as: $c_n = \frac{(c_r - a_r) * (b_n - a_n)}{(b_r - a_r)} + a_n$.

Note that, for this reason, two RTCP packets are needed in order for this feature to work properly: In my test cases, this usually happens more than ten of seconds after the stream has started.

2.4 Non-split of packets referring to the same frame

The chunkiser also implements the possibility to take advantage of the semantic of the *marker* bit in the RTP header, in case the video stream conforms to RFC 3551 [2, p. 30]. This feature can be enabled using **rfc3551=1** in the chunkiser command line: when enabled, the chunkiser will ignore the maximum delay option if a video frame is received with the marker bit set to 0. The 0 marker bit, in fact, indicates that more parts of the previous frame are to come. Then, as soon as a packet with marker bit set is received, the chunk will be sent.

Note that, at this stage, the "video stream" is assumed to be the value of **stream0** or **video**, as no other RTP metadata is read.

The assumption behind the implementation of this feature is that each part of the same frame is useless if considered by itself: thus, splitting it upon two chunks may result in only one chunk to be lost, which will make the effort of transmitting the other part of the frame become useless. Moreover, it is expected that the remaining parts of a single frame will come shortly after the first, without waiting for one sampling interval: thus, no significant additional delay will be introduced.

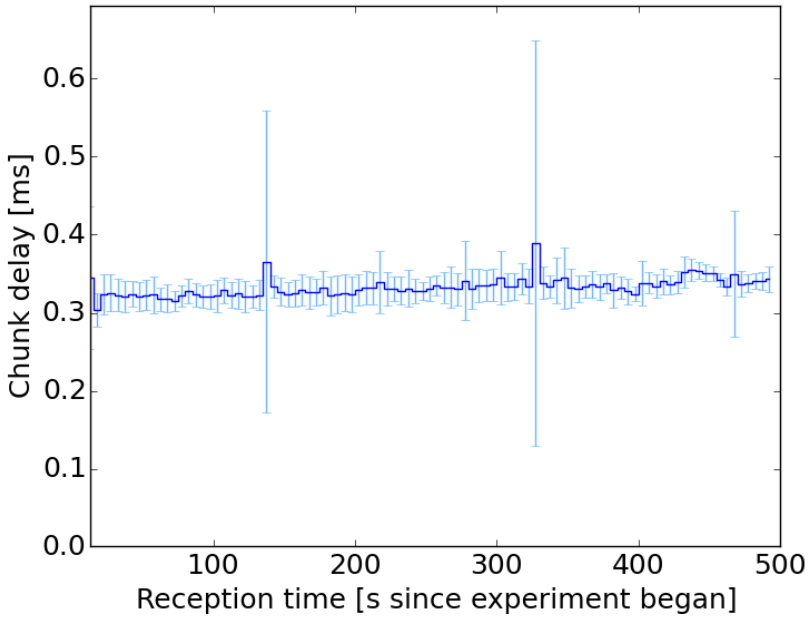

Figure 3.1: Chunks delay, computed as the difference between the creation and the reception of each chunk. The values are aggregated at 5-seconds intervals, and for each interval the average delay and an indication of the standard deviation among packets received in the interval are shown.

3 Proof-of-concept testing

The RTP [de]chunkiser module was tested on two physical hosts located on the same ethernet-based LAN network, with one PC running a stream source and the other PC running a player. The source node was running an instance of VLC player generating a RTP stream on the loopback network interface. On the same host, an instance of PeerStreamer was configured to listen on the appropriate ports and broadcast the stream to the P2P network. On the second host, a PeerStreamer instance was connected to the source node with its dechunkiser configured to output the RTP stream to a set of ports on the loopback interface, where an instance of VLC was listening for RTP packets and reproducing the stream, resulting in a setup similar to the one described in figure 1.1. Note that, in order to measure the delays by comparing time-stamps recorded at the two ends, the system clocks were synchronized using the NTP protocol.

This set-up is, of course, a best-case testbed for the software, but was considered enough for a basic proof-of-concept for the new chunkiser and dechunkiser modules, as they work end-to-end relying on the underlying P2P infrastructure.

Figure 3.1 shows the chunks transmission delay over time: due to the simple setup, the delays due to the chunks negotiation and transmission are in the order of tenths of milliseconds. Figure 3.2, instead, shows the delays of single RTP packets highlighting the

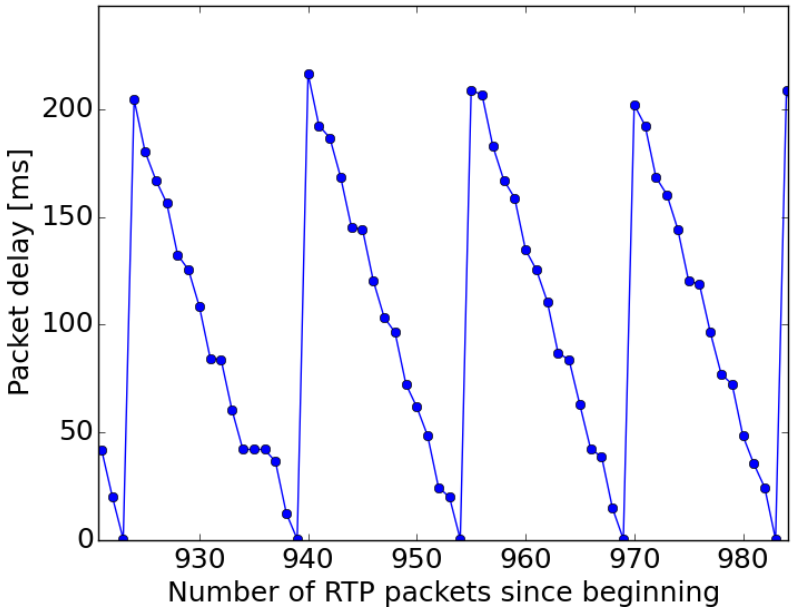

Figure 3.2: Big-timescale plot of the delay obtained by individual RTP packets, with the chunkiser maximum delay set to 200 ms. Delays are computed as the time span between the converted RTP timestamp written in the packet and the reception time of chunk that contained that packet.

pattern generated by the chunkiser's caching behaviour. Note that, because the chunk delays are up to 4 orders of magnitude smaller, they can not be seen at this scale. In the plot, aligned consecutive points correspond to packets aggregated in the same chunk, while almost-vertical lines identify the points when a chunk was sent and the next one was created.

4 Future work

This section describes the features that would be interesting but were not [yet] implemented because of time limitations.

4.1 Input description from SDP

An interesting feature would be the possibility to provide an SDP file, instead of direct port numbers, in order to describe the stream. This would be especially useful because SDP can be generated and parsed by many RTP streamers and players, including VLC: thus, this feature would prevent the need of manually specify port numbers in many cases.

4.2 Early packets time-stamping

As discussed in section 2.3.1, the **`max_delay_*`** options require two RTCP packets to have arrived in order to work properly. Thus, even when this option is enabled, during the first seconds of operation the behavior falls back to be controlled by the maximum chunk size: in practice, this results in the first tens of seconds of the RTP stream to be received with a delay which is much grater than the configured bound.

To overcome this problem, it would be possible to store alongside with each RTP packet its reception time-stamp and use it as an approximation of the yet unknown packet generation time: the chunkiser could then use this value in place of the correct one while the latter is not available, switching back to the current behavior after receiving the two required RTCP packets.

References

[1] Schulzrinne, Casner, Frederick. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. IETF. July 2003.
`http://www.ietf.org/rfc/rfc3550`

[2] Schulzrinne, Casner. *RTP Profile for Audio and Video Conferences with Minimal Control*. RFC 3551. IETF. July 2003.
`http://www.ietf.org/rfc/rfc3551`

[3] Schulzrinne, Casner, Frederick, Jacobson *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889. IETF. January 1996.
`http://www.ietf.org/rfc/rfc1889`

5 Appendix: user documentation

5.1 Command-line options

The following command-line options apply to both chunkiser and dechunkiser.

Note that "**ports pair**"s can be specified in three ways: as a colon-separated pair of port numbers like "<RTP port>:<RTCP port>"; as a single even port number N as a shortcut for "N:N+1"; as a single odd number M as a shortcut for "M-1:M".

When specifying ports to the chunkiser, it will listen on the given ports (on all available interfaces) for the UDP-carried RTP/RTCP traffic, while when specifying ports at the dechunkiser side, traffic will be generated towards the given UDP ports. Specifying at least a ports pair is required (no default port will be used). Always use the same syntax for specifying ports at the chunkiser and dechunkiser sides.

- **`stream0`** through **`stream9`** ("ports pair"s): specify UDP ports for the RTP/RTCP streams;
- **`video`** and **`audio`** ("ports pair"s): specify UDP ports for the RTP/RTCP streams; these setting have no effect if any of the **`streamN`** option is specified;
- **`base`** (integer): shortcut for **`stream0=<base>,stream1=<base+2>`**; it has no effect if any of the **`streamN`**, **`video`** or **`audio`** option is specified;
- **`verbosity`** (integer in 0–2, default 1): 0 only print unrecoverable errors, 1 also print warnings, 2 also print debug messages.

The following parameters apply only to the chunkiser:

- **`chunk_size`** (integer, default 65536): target chunk size in bytes. Whenever a chunk reaches this size, it will be immediately sent (see section 2.2 for details). Default value can be tuned at compile-time with the **`RTP_DEFAULT_CHUNK_SIZE`** constant;
- **`max_delay_ms`** or **`max_delay_s`** (integer, default 250 ms): use either option (but not both) to set the target maximum delay to accumulate in a chunk (see section 2.3 for details). Default value can be tuned at compile-time with the **`RTP_DEFAULT_MAX_DELAY`** constant;
- **`rfc3551`** (0 or 1, default 0): boolean setting whether the input stream can be assumed to be rfc3551-compliant; when using this option, always specify ports using the **`video`** and **`audio`** parameters. See section 2.4 for details;
- **`rtp_log`** (0 or 1, default 0): if 1, the chunkiser will print to its standard error an informative logging line for each received RTP packet.

Finally, the following parameter applies only to the dechunkiser:

- **`addr`** (string, default "127.0.0.1"): dotted decimal representation of the IPv4 address where the unwrapped rtp packets will be sent.

5.2 Examples

5.2.1 Compilation and offline testing (no P2P network involved)

```
$ cd THIRDPARTY-LIBS/GRAPES/src
$ export PJDIR=/xxx/xxx/THIRDPARTY-LIBS/pjproject-install/
$ make
$ make tests
$ ./Tests/chunkiser_test
  -I chunkiser=rtp,verbosity=2,stream0=3002,chunk_size=65536,
max_delay_ms=50,rfc3551=1 \
  -O dechunkiser=rtp,verbosity=2,stream0=5000 \
  null null
```

5.2.2 Usage with VLC and SDP files

```
SRC-HOST $ cvlc video_file.ts \
  --sout "#rtp{dst=127.0.0.1,port=5004,sdp=file://src.sdp}" &
SRC-HOST $ streamer-udp-grapes-static -P 6000 -m 1 \
   -f null,chunkiser=rtp,max_delay_ms=200,audio=5006,video=5004

DST-HOST $ # download 'src.sdp' from SRC-HOST
DST-HOST $ sed -r -e 's/5004/7004/g' -e 's/5006/7006/g' \
  <src.sdp >dst.sdp
DST-HOST $ streamer-udp-grapes-static \
  -i SRC-HOST -p 6000 \
  -F null,dechunkiser=rtp,verbosity=2,audio=7006,video=7004 &
DST-HOST $ cvlc dst.sdp --sout '#display'

# For webcam streaming, you can try the following vlc invocation.
# Note that you may need to configure VLC to limit the buffering.
SRC-HOST $ cvlc v4l:///dev/video0  # may be different on other machines

$ ./Tests/chunkiser_test
  -I chunkiser=rtp,verbosity=2,stream0=3002,chunk_size=65536,
max_delay_ms=50,rfc3551=1 \
  -O dechunkiser=rtp,verbosity=2,stream0=5000 \
  null null
```

6 Appendix: developer documentation

Despite this document should have given a general overview over the module, this section adds some extra implementation details that may hopefully be useful when some maintenance will be required.

6.1 [De]Chunkiser context variables

The *context* for the chunkiser (or dechunkiser) is a singleton instance of **`struct chunkiser_ctx`** (or **`struct dechunkiser_ctx`**, respectively) which is accessible by all (de)chunkising-related functions, and represents the internal state of the (de)chunkiser.

The chunkiser context stores the following variables:

- **`int max_size`** alloccation size for **`buff`**. Set at initialization time to one maximum-UDP-packet-size more than the target chunk size;
- **`uint64_t max_delay`** maximum delay to accumulate in a chunk (in ntp format, with the most significant 32 bits for seconds and the least significant ones for fractions of second);
- **`int video_stream_id`** index in the **`streams`** array to indicate which one is the video (useful to be known if **`rfc3551`** is set)
- **`int rfc3551`** boolean, set at initialization time according to the respective input option;
- **`int verbosity`** set at initialization time according to the respective input option;
- **`int fds[RTP_UDP_PORTS_NUM_MAX + 1]`** "-1"-terminated array of file descriptors for the UDP ports where the chunkiser is listening. Even indices are for RTP open ports, while each subsequent odd index listens to the corresponding RTCP port;
- **`int fds_len`** even if "-1"-terminated, save its length to make things easier
- **`struct rtp_stream streams[RTP_STREAMS_NUM_MAX]`** sequence of descriptors for each RTP/RTCP stream. Each entry stores data structures needed by the underlying RTP library and the two most recent NTP-RTP time-stamps associations. Its length is **`fds_len / 2`**;
- **`uint8_t *buff`** pointer to a buffer containing the current chunk being constructed;
- **`int size`** the portion of the buffer being currently occupied. Note that there should always be room enough for one maximum-sized UDP packet, so whenever the extra room is not enough the buffer is immediately sent.

- **int next_fd** index in the **fsd** array, representing which of the file descriptor (i.e. port) is to be polled next. Note that ports are checked sequentially, but if the maximum chunk size is reached the subsequent round will start from the point it had stopped (so as to avoid port starvation);
- **int counter** counts the chunks sent so far;
- **uint64_t min_ntp_ts** NTP timestamp of the oldest packet in the current chunk;
- **uint64_t max_ntp_ts** NTP timestamp of the most recent packet in the current chunk;
- **int ntp_ts_status** represents the validity of the previous 2 values: 1 means they are valid, 0 means they are junk (i.e. no RTP packet has been put in the current chunk yet), while -1 means the chunk contains packets with unknown timestamp, so the time-stamping option can not be used for this chunk.

The dechunkiser context is instead much simpler:

- **int verbosity** set at initialization time according to the respective input option;
- **int outfd** The file descriptor for the UDP socket to be used for sending (corresponds to an ephemeral port assigned by the operating system);
- **char ip[IP_ADDR_LEN]** the target ip address, set at initialization time according to the respective input option;
- **int ports[RTP_UDP_PORTS_NUM_MAX]** set of ports that will be used for demultiplexing the packets from the incoming chunks. It is ordered in the same way as the chunkiser counterpart, so that the stream identifier in the chunks' headers can be interpreted coherently;
- **int ports_len** length of **ports**.

6.2 Affected files

Except for the MAKEFILEs needed to include the external RTP library, the additions were limited to the **src/Chunkiser** directory. Files named hereafter are contained into this folder.

These files needed some additions:

- **payload.h**: constants and inline functions for generating and parsing the chunk format;
- **input-stream.c** and **output-stream.c**: the possibility to use the RTP chunkiser and dechunkiser was added here, linking to the appropriate newly-written functions.

While the following files were added:

- **`stream-rtp.h`**: holds constants and inline functions that are shared among the RTP chunkiser and the dechunkiser;
- **`input-stream-rtp.c`**: holds the core implementation for the chunkiser;
- **`output-stream-rtp.c`**: holds the core implementation for the dechunkiser.