Building `conf`, a simple audio conference application, v4.1

In this practice, you should write an application called `conf` which allows establishing a point-to-point audio conference between two systems. In this implementation you are requested to use RTP over UDP, so you must read the corresponding specifications (RFC 3550, RFC 3551 and RFC 4961).
One objective of this assignment is to become familiar with specifications written in a standard document (RTP/RTCP in this case, which is, by the way, a very easy protocol, compared to other specifications). Note that the standard specification contains sufficient information to guide your implementation choices; it even includes example code (for your convenience, we have extracted the code in a file `rtp.h` which is available on Aula Global, along with additional material.) Read the standard carefully before you start writing code.

The main architecture of the application is as follows:

- The application could be started in two ways: In `first` mode, where the process waits indefinitely for the other `conf` entity (`second`) to send data. In `second` mode, `conf` initiates communication by sending immediately an audio packet to `first`.
- Once communication has been established between `first` and `second`, the execution code is the same for both. For now on, each system running `conf` reads data continuously from the soundcard (capturing the data recorded from the microphone) and sends these data to the remote `conf` system. Besides, the data received from a remote `conf` system is stored in a *circular buffer*. This buffer allows accumulating some amount of data to counteract for possible variations in the transmission delay. Once enough data has accumulated, it is transferred to the sound card and playout begins.

Command line interface

The command-line interface the program shall offer is the following:

```
conf first [-pLOCAL_RTP_PORT] [-vVOL] [-c] [-kACCUMULATED_TIME] [-
mMULTICAST_ADDR]

conf second addressOfFirst [-pLOCAL_RTP_PORT] [-vVOL] [-c] [-lPACKET_DURATION]
[-kACCUMULATED_TIME] [-yPAYLOAD]
```

Verbose mode, `-c`

This parameter exists for both `first` and `second`.
With `-c` (*verbose* mode), a dot ("`.`") shall be printed each time a packet is sent and a "+" each time a packet is inserted in the circular buffer to be played. Besides (each of these cases is explained below in the specification),

- every time packet loss is detected while checking the RTP sequence number, an '`s`' (for *sequence*) is printed. If packets are played-out due to this event, an '`x`' is printed
- every time the `select` timer expires, a '`t`' (for *timer*) is printed
- every time a packet arrives after the timer (programmed for the case this packet would not arrive) expires, a '`d`' (for *delayed*) is printed.

Examples that may occur are the following (note that '`.`' are not relevant here):

- Packet 11 arrives, 12 is lost, and the timer for 12 expires before 13 arrives: '`+.t+`'.
  Each t displayed indicates a packet has been generated to substitute the packet expected but not received
- Packet 11 arrives, 12 is lost, 14 arrives before expiring 12's timer: '`+..sx.x+`'.
  `s` indicates there are missing packets before number 14 was received; `x` stands for packets number 12, other `x` for packet 13, then + indicates that 14 is appropriately played
- Packet 11 arrives, the timer for 12 expires, then 12 arrives, 13 arrives on time: '`+.t.d+`' (`d` is generated by 12 arriving when `conf` has already 'created' a packet with sequence 12)

When the execution in verbose mode finishes, it must show the theoretical time required to playout the content, as well as the actual time required to perform the playout: time elapsed since the playout of the first packet to the playout of the last one (or a good approximation possible to that time); the comparison between the theoretical and actual playout times can be used to evaluate the quality of an implementation.

`-c` can be independently activated in `first` or `second`.

Local RTP port, `-p`
This parameter exists for both `first` and `second`.
The ports to use can be configured using `-p`, with a default value of 5004.
Port numbers must be the same for both `first` and `second` modes.

Volume, `-v`
This parameter exists for both `first` and `second`.
It allows setting the local volume for both recording and playout, which can have different values at `first` and at `second`.

Accumulated time, `-k`
This parameter exists for both `first` and `second`.
`-kACCUMULATED_TIME` is used to indicate that `ACCUMULATED_TIME` milliseconds of data must be stored in the circular buffer before starting the playout. This time is internally rounded down by `conf` to a multiple of the actual playout time of a packet, so that an integer number of packets are buffered. For example, if `-k90` is specified, and a packet has an actual duration of 16.00 ms, the accumulated data will correspond to 80 ms (5 16-ms packets).
100 ms is the default value for this parameter.
Note that when allocating memory, the circular buffer must be sized to store 200 milliseconds in addition to the buffering time specified in this command, to cope with possible delay variations caused by the network. For example, for `-k50`, the buffer will be sized to be able to store the data corresponding to 250 ms of playout (rounded down to the multiple of the actual playout time of a packet).
`-k` may have different values for `first` and `second`.

Multicast operation
The application running as `first` must be able to use a multicast address to receive the data, as an alternative to receiving it from its unicast address. This is achieved by means of the `-mMulticastAddr` optional parameter for `first` mode (for example, `-m226.0.0.100`), and the `addressOfFirst` for `second` mode configured to the same multicast address. In the following example, `first` listens at multicast address `226.0.0.100` (using `-m`), which is the same address to which `second` sends its data.

```
Host1:~> ./conf first -m226.0.0.100

Host2:~> ./conf second 226.0.0.100
```

Remember, that the range of valid multicast addresses is 224.0.0.0 to 239.255.255.255 (we recommend you to use an address in the range 226.0.x.x). You can verify if the association is correct by executing a `ping` to the address.
`-m` can only be used with `first`.
If `-m` is not used, then `second` is started with the IP unicast address of `first`, as in the following example.

```
Host1:~> ./conf first

Host2:~> ./conf second 163.117.144.7
```

Packet duration, `-l`
This parameter is only available with `second`.
The option `-l` allows specifying the duration in milliseconds of the play-out of the data being carried by a packet. By default, this value is 20 ms.
The value of `-l` is just a hint, since the actual length used by first for the audio blocks will be rounded down to the immediately inferior power of 2 number of bytes, to comply with the fragment size restrictions imposed by the soundcard operation.

`first` learns the audio block length when it receives the first audio packet from `second`, by processing the byte length of the UDP packet received (`recvfrom` system call).

Payload, `-y`
This parameter is only available with `second`.
Two values are allowed: `-y100` indicates L8 format, and `-y11` indicates L16 format. The default value is L8.

L8 refers to a linear coding of 8 bit per sample, single channel, sampling frequency of 8000 Hz; L16 to linear coding of 16 bit per sample, single channel at 44100 Hz. They are partially defined in *RTP profiles for audio and video* (RFC3551), sections 4.5.10, 4.5.11 y 6.
Values of 100 and 11 are the payload type values to be carried in the corresponding RTP packet.

In order to configure the format of soundcard playing and recording, by means of the `SNDCTL_DSP_SETFMT` command with the `ioctl` call), you should use format value `8` for L8, and format value `16` for L16. (Note that the format value refers to the value used to configure the soundcard).

`first` learns the payload, and therefore the audio format to be used in the communication, when it receives the first audio packet from `second`, by reading it from the RTP header.

Suggested structure for your code

You should write a single program (named `conf`) composed of three sections:
The first section is responsible of initiating the communication between the two modes, `first` and `second`. It consists of an `if` statement which performs different operations for `first` and for `second`. At the end of this section,

- `second` has configured the soundcard according to the `-y` and `-l` values. It has created the circular buffer sized according to the information obtained from `-y`, `-l` and `-k`, recorded an audio block, inserted it in an RTP packet, and sent this (one) packet to first using the IP address introduced through the command line. Note that this first packet contains audio information.
- `first` has received the first packet from `second`, learnt `second`'s address through the `recvfrom` system call, it has obtained the payload and length from the received packet, has configured the soundcard with this information, created its circular buffer (using its own `-k` value), and inserted the first block of audio data on it.

When this section ends, either activated as `first` or `second`, `conf` knows the IP address of the remote node, the payload to use, the audio block length, has the soundcard configured, and the circular buffer created.
From now on, the code is common for both `first` and `second`.

The second section is responsible for accumulating the buffering data. In this section, both `conf` entities are recording and sending to the other endpoint, and receiving data from the network and storing it in the buffer, but are not sending data to the soundcard (there is no playout). To implement this application we recommend a model in which a single process takes care of different events[1]. Then, the application uses the `select` system call to wait for each one of the following events:

- Once data is available (recorded) at the soundcard, it is immediately sent to the remote system[2]. No buffering is performed when sending to the other side.
- Once data is received from the remote node (`recvfrom` system call), the audio data is copied into the circular buffer.

The steady regime of the program is achieved at the third section, in which the `select` system call attends to the following operations:

- Once data is available (recorded) at the soundcard, it is immediately sent to the remote system. No buffering is performed when sending to the other side.
- Once data is received from the remote node (`recvfrom` system call), the audio data is copied into the circular buffer.
- Writing data to the soundcard. We suggest you to try to write audio blocks the soundcard when there is any in the circular buffer. In other words, if there are audio blocks in the circular buffer, the

[1] A not recommended alternative is to choose an architecture comprised of several processes or threads, each in charge of a given task.
[2] Note that the generation of a fixed amount of audio data with at constant rate format requires constant time, so this activity is periodic.

`select` system call must be programmed to request a `write` operation in the soundcard. The rationale for this is that we prefer the data to be in the soundcard than in the circular buffer, so that any short operating system timing variation will not impact in the playout quality.
If the amount of data written exceeds the maximum available space in the soundcard memory, the operating system will block the writing operation, and it will awake the process later when there is enough space.

RTP considerations.
For UDP data sending, you must consider the recommendation made in RFC 4961, so that source and destination ports must be the same for all UDP packets generated for RTP.
For timestamp encoding, take into account the recommendations of RFC3551: The timestamp of the RTP header of a packet is the timestamp corresponding to the first sample included in the packet, and the frequency used to mark the packets is the same as the one used for sampling. So if a packet contains 128 samples and the packet has a timestamp of 3300, the following packet must have a timestamp of 3428.

To ease debugging, we recommend you to start sequence numbers and timestamp values by 0. Be careful with the coding of RTP headers to make them compliant to the standard: control fields must honor the byte order convention for sending data over a network, the *network byte order* or *Most Significant Byte*, in contrast to what it is used by our Intel systems, so that you should use the functions `htonl`, `htons` and their inverses `ntohl` and `ntohs` appropriately. Moreover, the audio data should be transported using an appropriate byte order (so that they can be received by architectures with a different byte order and can be reconstructed in the correct way). Therefore, you should use conversion functions if data are recorded in a 16 bits format.
Use also 0 and 1 for the SSRC of `first` and `second`, respectively.
For the other packet header fields, use the value 0.

Sequence number check
For each RTP packet received from the remote `conf` system, the application is required to check for the appropriate sequence number. This check (and the operations described in this paragraph) is performed when the packet is received.
We assume that packets can be lost in the network, but never reordered (this is a very simplified model of a real network, in which reordering may definitely occur). This assumption is raised to allow simplifying the code. With this 'ordered delivery' assumption, if we receive a packet with sequence number [X], and then a packet with sequence number [X+2], we must assume that only possible reason for this is that packet [X+1] has been lost (because reordering is impossible due to our initial assumption).
In this case, the behavior specified for `conf` is that the data corresponding to [X] will be copied to the position in the circular buffer corresponding to [X+1], with the data corresponding to [X+2] (the same data) inserted right after.
If packet [X+1] arrives later, it must be discarded.

If verbose mode (`-c`) is active, this part of code is responsible for generating `d` (*delay*) and `s` (*sequence*) outputs.

Late packets

If verbose mode (`-c`) is active, this part of code is responsible for generating `t` (*timer*) outputs.

Optional improvements

1. Processing RTCP.

The application sends RTCP information to the remote system in a periodic fashion (this is a simplification of the standard specification). The packets to be sent are SR packets, SDES with CNAME and TOOL information, and a BYE packet when the user decides to interrupt the transmission by pressing CTRL-C (the other communicating peer must stop its execution in a proper way). For an SR packet, it is not necessary to send the NTP information – you can set this value to 0. To fill in the CNAME field, you could use the operating system call `gethostname`. The BYE message, when generated, does not carry an indication of the reason why the session has been terminated. Remember that an SR and a SDES are always packed in the same UDP packet, and if necessary, a BYE message is appended to those two. The user information (SDES packet) could be hard-coded into the program or could be obtained from a configuration file (which is a more flexible option).
In the receiver side, the application processes received RTCP information at any time. If a BYE packet is received, the application stops. When receiving the CNAME or TOOL message the first time, or each time you detect a change, you must show the value of this field on the screen.
For the sake of simplicity, we are not requesting you to implement the RTCP soft-state approach: the participants of the audio conference are not going to deal with the fact that their communication partner has aborted the session by monitoring the time when they received information from the partner the last time. Therefore, the only way to terminate a session is to receive a BYE message or press CTRL-C.
Take into account for your code that some of the implemented functions in the standard (files rfc3559.h and .c) can give hints on how to parse an SDES message.
If verbose mode is active, and a RTCP message arrives, a "<" shall be printed, whereas a ">" shall be printed on departure of a RTCP packet.
Use `rtpdump` to test that you RTCP code is compliant with the requirements stated above in this specification. Take into account the comments made before about the byte order of the data: If you do not use the appropriate format, `rtpdump` will not interpret the arriving data correctly.
Proposed implementation order for RTCP:

1. Implement the data transmission part (including the RTP headers) and verify if this part works correctly.
2. Implement the generation and reception of RTCP packets.
3. Start with the SDES packets, which are very easy to construct and to interpret. Test whether they are generated and received correctly and if they are written on the screen when required.
4. Thereafter, incorporate the SR packets. Verify that they are sent and received and that the transmitted data are reasonable.
5. Implement the BYE functionality.
6. Test with `rtpdump` that the format of the sent packets is correct (even if your code understands what it receives, this does not mean that it conforms to the standard. It must be understood by any RTP/RTCP application).

1.1. A simplified enhancement is to implement only the BYE functionality. In this case, only BYE messages are generated, although these messages must be in the proper format (so that `rtpdump` could parse the information exchanged).

2. Support of out-of-order packets
You can improve the practice by supporting out-of-order packets. In order to do so, we propose you that modify the circular buffer that we provide you in the following manner: Introduce for each memory block a variable that allows you to store the sequence number (transmitted by RTP) corresponding to the data block contained in that sequence. If N is the number or blocks that contain the circular buffer, always put the block of data with sequence number S in the position of the buffer 'S module N', and update the value of the number of sequence associated to that block. In that way each data block has a fixed place, and it is possible to place appropriately data blocks that arrive later than other that come after them in the playback logic sequence. When you playback a data block, check that it corresponds to the expected sequence number (on the contrary, playback the block coming immediately before it).

3. Silence detector
Implement a 'silence detector' heuristic and avoid sending packets identified as silent.
When data packets are sent after a silence period, timestamps are configured to indicate the time at which this data burst must start.