Tecnologías Multimedia - Study Guide - Milestone 5: Minimal InterCom

Vicente González Ruiz - Depto Informática - UAL

September 30, 2020

1. Description

1.1. Sequential model

InterCom can be divided roughtly in 6 steps:

Algorithm 1.1: SEQUENTIAL_INTERCOM( )

procedure RECORD_IO_AND_PLAY( )

```
chunk ← record()
packed_chunk ← pack(chunk)
send(packed_chunk)
packed_chunk ← receive()
chunk ← unpack(packed_chunk)
play(chunk)
```

Where:

1. `record()` captures a chunk of frames. In `sounddevice`, this operation is carried on by the `read()` method. As it can be seen in wire4.py[1] and also in the documentation of `sounddevice`, if we read only the frames that are available in the ADC's buffer, this in non-blocking operation and the chunk size depends on the instant of time in which this method is called. Otherwise, if we especify a number of frames different to the number of available frames, the operation is blocking and I/O-bound (the calling process sleeps until the required chunk size is returned).

2. `pack(chunk)` process the chunk to create a packet (or a sequence of packets), a structure that can be transmitted through the Internet using the Datagram Model. In general, this is a CPU-bounded (CPU-intensive) operation, and therefore, it reduces the number of cycles/second that the Record_IO_and_Play's loop can reach.

3. `send(packed_chunk)`, sends the packet to the interlocutor's Inter-

[1] `curl https://raw.githubusercontent.com/Tecnologias-multimedia/intercom/mas`
`> wire4.py`

Com. When datagrams are used, this step is non-blocking neither CPU-bounding (the CPU usage is very low), as long as the number of packets/second is small and the sizes of the payloads are also small, as it is expected in the InterCom.

4. `receive()`, waits (blocking) for a incoming packet, and therefore, this operation is IO-bound. However, most socket APIs [2] offeer a non-blocking option where when a packet is not available in the kernel's buffer associanted to the corresponding socket, some kind of exception is generated and, in this case, it is resposability of the programmer to generate an "alternative" chunk (in our case, for example, a chunk filled with zeros will not produce any sound).

5. `unpack(packed_chunk)` is (as `pack(chunk)`) a CPU-intensive step that transforms a packed_chunk into a chunk of audio.

6. `play(chunk)` renders the chunk. In general, this is an I/O-bound blocking action. However, if `play()` is called at the same pace than `record()` and the record and play parameters are exactely the same (as happens in our model), the playing of the chunk will return in-

mediately because the time that the `play()` method needs to complete would exactly match the time that the `record()` method requires (see `wire4.py`).

This model works fine if the chunk size is controlled by PortAudio [3] (`sounddevice`) and also, the run-time required by `pack(chunk)` and `unpack(packet)` operations is smaller than the chunk-time. The first issue is complicates slightly the implementation because we need to work with chunks of constantly changing lenghts (that as you can see, most of the time are 0). This also complicates the control of the latency (the chunk size is variable). However, the real problem appears when our computer is not able to satisfy the second requirement. This only can be addressed through optimizing and parallelizing the code.

1.2. Timer-based model

In this model, the task dedicated to record and play the chunks of audio is called periodically (probably, using some timer provided by the sound hardware). This procedure guarantees a gliches-free audio-IO when constant

chunk sizes are used because the timer interruption coincides exactly with the instant of time in which the `record()` and the `play()` methods need to be run to guarantee that we are going to handle a chunk of audio of a known size without blocking. The following algorithm describes the new model that is basically the previous one, except that the chunk size is controlled.

Algorithm 1.2: TIMER-BASED_INTERCOM( )

procedure RECORD_IO_AND_PLAY(`chunk_size`)

```
chunk ← record(chunk_size)
packed_chunk ← pack(chunk)
send(packed_chunk)
packed_chunk ← receive()
chunk ← unpack(packed_chunk)
play(chunk)
```

2. What you have to do?

1. Implement a timer-based InterCom. Use as reference wire.py[2]. Your implementation should be able to record, transmit [1] (using UDP [4]) and play raw (this means that the `pack()`/`unpack()` operations are not required for this milestone) audio data between two computers.

[2] `curl https://raw.githubusercontent.com/Tecnologias-multimedia/intercom/mas`
`> wire.py`

3. Timming

You should reach this milestone at most in two weeks.

4. Deliverables

A Python module with the implementation of the minimal InterCom. Name it `intercom_minimal.py`. Store it at the root directory of your `intercom`'s repo.

5. Resources

[1] YAPT.

[2] Python website.

[3] PortAudio.

[4] J. Postel. *RFC 768. The User Datagram Protocol (UDP)*. USC/Information Sciences Institute, http://www.rfc-editor.org/rfc/rfc768.txt, 1980.