Tecnologías Multimedia - Study Guide - Milestone 5: Minimal InterCom

Vicente González Ruiz - Depto Informática - UAL

September 1, 2021

1. Description

1.1. Sequential algorithm

InterCom can be divided roughtly in 6 steps:

```
# Sequential algorithm
def record_IO_and_play():
    chunk = record()  # (1)
    packed_chunk = pack(chunk)  # (2)
    send(packed_chunk)  # (3)
    packed_chunk = receive()  # (4)
    chunk = unpack(packed_chunk)  # (5)
    play(chunk)  # (6)
```

Where:

1. the `record()` method 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 is a 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 executions/second that the `record_IO_and_play()` method can reach.

3. `send(packed_chunk)`, sends the packet to the InterCom's interlocutor. When datagrams are used, this step is not blocking neither CPU-bounding (the CPU usage is very low), as long as the number of

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

packets/second is small and the sizes of the payloads are also small, as it is expected in InterCom.

4. `receive()`, waits (blocking) for an 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 associated to the corresponding socket, some kind of exception is generated and, in this case, it is resposabability of the programmer to generate an "alternative" chunk (in our case, for example, a chunk filled with zeros that will not produce any sound when it is played).

5. `unpack(packed_chunk)` is (like the method `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 algorithm), the playing of the chunk will return inmediately because the time that the `play()` method needs to com-

plete would exactly match the time that the `record()` method requires (see `wire4.py`).

This algorithm works fine if the chunk size is controlled by PortAudio [1] (`sounddevice`) and also, the run-time required by `pack(chunk)` and `unpack(packet)` operations is smaller than the chunk-time. The first premise (variable chunk sizes) complicates slightly the implementation because we would work with chunks of constantly changing lenghts (that as you can see, in most of the iterations are 0). This also complicates the control of the latency because the chunk size is variable. However, the real problem appears when our computer is not able to satisfy the second requirement, i.e., when the chunk time is smaller than the time that we need to process the chunks. This only can be addressed through optimizing (for example, parallelizing) the code.

1.2. Timer-based algorithm

In this algorithm, 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 we are going to handle a chunk of audio of a known size without blocking. The following algorithm describes the new algorithm that is basically the previous one, except that the chunk size is fixed.

```
# Timer-based algorithm
def 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)
```

The current implementation of InterCom uses the Timer-based algorithm.

2. What you have to do?

1. Read the source code of `minimal.py` and identify in the code the interruption handler that implements the timer-based algorithm. This guide [3]

2. Evaluate the performance (quality of the sound) in different configurations:

   (a) When `minimal.py` runs in the same computer.

   (b) When `minimal.py` runs in different computers. In this case, try that the computers do not belong to the same local network. This guide can help. Consider in your analysis that UDP [4] is used to transport the audio data.

3. Timming

You should reach this milestone at most in one week.

4. Deliverables

The results of the experiments proposed in the previous section, that will be presented to the rest of the class.

5. Resources

[1] PortAudio.

[2] The Python Foundation. The Python Website.

[3] V. González Ruiz. YAPT.

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