# Tecnologías Multimedia - Study Guide - Milestone 6: Buffering chunks

Vicente González Ruiz - Depto Informática - UAL

September 24, 2020

# 1. Description

## 1.1. Sequential model

InterCom can be divided roughtky in 6 steps:

**Algorithm 1.1:** SEQUENTIAL_INTERCOM( )

```
recorded_chunk ← record()
outgoing_packet ← pack(recorded_chunk)
send(outgoing_packet)
incoming_packet ← receive()
chunk_to_play ← unpack(incoming_packet)
play(chunk_to_play)
```

Where:

1. `record()` captures a chunk of frames. In general, this is a I/O-bound blocking operation, depending on the availability of frames in the ADC's buffer. This can be checked in wire4.py,

where the `read()` method always returns inmediately because only the available frames are returned in each call (the chunk-size depends on the instant of time in which this method is called).

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, reduces the number of cycles/second that the main thread can reach.

3. `send(packet)`, sends the packet to the interlocutor's InterCom. When datagrams are used, this step is non-blocking neither CPU-bounding (the CPU usage is very low), if the number of packets/second is small and the size of the payloads is 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 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 resposabability of the programmer to generate a "alternative" chunk (in our case, for example, a chunk filled with zeros will not produce any sound).

5. `unpack(packet)` is (as `pack(chunk)`) a CPU-intensive step that transforms a packet (or several packets) 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 inmediately because the time that the `play()` method needs to complete would exactly match the time that the `record()` method requires.

This model works fine if the chunk size is controlled by PortAudio 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.

**Algorithm 1.2:** TIMER-BASED_INTERCOM( )

**procedure** RECORD_IO_AND_PLAY(chunk_size)

```
recorded_chunk ← record(chunk_size)
send(outgoing_packet) ← pack(recorded_chunk)
send(outgoing_packet)
incoming_packet ← receive()
chunk_to_play ← unpack(incoming_packet)
play(chunk_to_play)
```

where:

1. `indata` points to the last chunk of audio recorded.
2. `outdata` points to the chunk of audio that is going to be played.
3. `chunk_size` is the number of frames processed in each iteration of the callback.

The advantage of this alternative compared to the Concurrent Model is that we can control the size of the buffers, which in the end controls the latency.

---

After analyzing these steps, we can see that only `pack(chunk)` and `unpack(packet)` deserve to be parallelized.

Now is a good moment to think about the computing resources that can be available: in general, more than one PU (Processing Unit) or core is available.

Using non-blocking I/O

Notice that the execution of these steps, in this order, in two different InterCom instances has at least two problems:

1. If a packet is lost, a deadlock is generated, because the `receive()` never returns.

2. If the time required by `pack(chunk)` and `unpack(packet)` is largest than the chunk-time, a glich will be generated.

The Sequential Model is straightforward. However, we cannot control

the chunk size without generating a blocking in the record and play steps. This can produce gliches.

## 1.3. Concurrent model

The parallel model supposes that there is at least two Processing Units (or PUs, typically cores), that are able to work in parallel. Thus, the sequential

model is divided into two concurrent tasks that should run in parallel.

**Algorithm 1.3:** CONCURRENT_INTERCOM( )

**procedure** RECORD_PACK_AND_SEND( )

```
chunk ← record()
packet ← pack(chunk)
send(packet)
```

**procedure** RECEIVE_UNPACK_AND_PLAY( )

```
packet ← receive()
chunk ← unpack(packet)
play(chunk)
```

In this case, the loss of a chunk does not generates a deadlock between the concurrent tasks, because even if the Receive_Unpack_Play is blocked waiting for a block in the `receive()`, the `Record_Pack_and_Send` task will continue sendind chunks.

Unfortunatelly, gliches can still happen in this model if the packets are lost or delayed, or if `pack(chunk)` or if `unpack(packet)` needs more than a chunk-time to be completed.

Most of the socket APIs offeer a version of

# 2. What you have to do?

1. Implement a timer-based InterCom. Use as reference wire.py.

# 3. Timming

You should reach this milestone at most in one week.

# 4. Deliverables

# 5. Resources