Multi-Threading/Multi-Processing and its Inter-Process Communication

Herman Bruyninckx
Eindhoven University of Technology
KU Leuven
http://people.mech.kuleuven.be/~bruyninc/

May 18, 2016

Overview of this lecture

- **objectives**: to explain
  - **when** (not) to use multiple processes
  - **how** to deal with their **data flows** and **event flows**
  - some **best practices**
- **use cases** in project:
  - get data from laser scanner
  - process it for (i) world model update, (ii) motion planning, and (iii) motion control
  - feed GUI with status information about execution
  - get commands from GUI
  - log data for later inspection
- **key reference**: *"Multithreading with ZeroMQ"*. `http://zguide.zeromq.org/page:all#toc45` Also contains code examples in many programming languages.

Multiple processes: why?

Good reasons:

- some activities **must run asynchronously**: sensors & actuators with their own computers; activities share a resource that has its own computer;...
- deployment **must be on physically separated** machines. (= special case of previous one)

Bad reasons:

- because you think it is more *modular*
- because you think *publish-subscribe* is the only way to get data from one algorithm into the next.
- because you do not (want to) know how to make your *system behave deterministically*; i.e., at each moment of its activity, it is clear *what* it is doing, *why*, and *how well*.

Inter-Process Communication: how & why?

How?

- **message passing** interface: `send(message,channel`, `read(message,channel`
- **streams** interface: like message passing, but caller is ready for the case there is no data yet, no data anymore, or multiple answers at the same time.

Why?

- **data streams** between **identified** data processors
- **coordination events** between **all** processes
- while **making sure** that
  - no data is lost (unless application wants it!)
  - no data is corrupted
  - access to data is efficient

  $\Rightarrow$ *all* of these reasons have a performance that (should) depend on the application!

Multiple processes: bad practices

- avoid **mutexes or locks**

$\rightarrow$ these are *OS-facing* programming primitives, that can be misused in way too many ways...

$\rightarrow$ use *library* that (i) provides abstraction of **access to shared resources**, and (ii) with **configurable application-facing performance**.
Example: ZeroMQ.

- avoid **priorities** to determine "which process goes first"

$\rightarrow$ use events-with-meaning!

- avoid **share state**, where "state" is all the data that determines the behaviour of this process

$\rightarrow$ use *information about other processes' state* instead!

Multiple processes: best practices

- **one event queue** in each process, hence:
  - **one Coordinator**:
    - takes decisions on *what* to do
    - based on interpretation of *incoming* events
    - decisions are provided in the form of *outgoing events*
  - **one Scheduler**: defines *in what order* to call functionalities
  - **one Configurator**: defines all *"magic numbers"*

  Note: **time is just another event**
- **multiple data queues**
  - configured via events
  - available as fully accessible "streams"
- **multiple Computations**
  - configured via events
  - take data/event flows in, put data/event flows out
  - "control", "monitoring", "world modelling", "trajectory selection",...

Multiple processes: best practices (2)

Pattern: event loop in C

```
when triggered      % by OS
do {
    communicate()   // get latest events & data
    coordinate()    // handle the events
    configure()     // possibly requiring reconfiguration
    schedule()      // run functions on new data
    coordinate()    // functions could trigger
                    // new events or data
    communicate()   // that others might have to know about
    log()           // memory of what happened when
}
```


Pattern: event loop for "control"

when `scheduled` **do** { `act(); prepare();` }
with

```
act() {
  sense();          // get sensing data out of "process message"
  control();        // get continuous part in "process message"
  communicate();    // to get control out as fast as possible
}
prepare() {
   world-model-update(); // process sense results further
   plan(); // compute feedforward for next loop
   ...
   if monitor() then {
       coordinate();
       configure();
   }
}
```

Specific design force: get control signal out as fast as possible

Suggestion: discovery & communication in ZeroMQ

Streams with **ZeroMQ**:

> *"ZeroMQ sockets provide an abstraction of asynchronous message streams, multiple messaging patterns, message filtering, seamless access to multiple transport protocols and more."*

`http://www.zeromq.org/`

Discovery & **group** communication with **Zyre**:

`https://github.com/zeromq/zyre`

- *whisper* to identified peer in a group
- *shout* to all peers in a group
- one peer can take part in several group communications.