ATLAS

RT systems, Scheduling, Link to Control

Best Practices in integrating complex robotic systems

Gianni Borghesan
KU Leuven
Febraury 2019

Outline

Real Time Systems Definition:

RT-systems are

1) systems whose correct working depends not only by the correctiveness of output, but also by the timing of such output

2) System that emulate a physical behaviour over time

3) System that produces output with no significant delay

Real Time Systems Definition:

RT-systems are

1) systems whose correct working depends not only by the correctiveness of output, but also by the timing of such output

2) System that emulate a physical behaviour over time

~~*3) System that produces output with no significant delay*~~

Where RT-systems are needed (some examples)

- critical-safe applications
  - Traffic control (airplane and trains)
  - Stock exchange
- (critical-safe) mechatronic/control applications
  - Avionics-aereospace, automotive,
  - industrial/power plants, power grid
- Applications with quality of service
  - Communication
  - digital signal processing (e.g. audio recording)

What makes a system realtime

Determinism

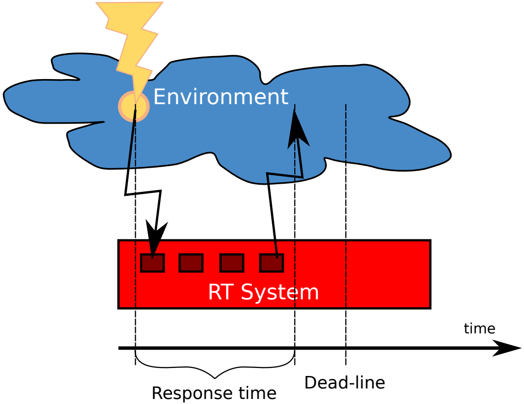
A RT-system produce an output before a given time (**dead-line**) from a trigger.

Note that a realtime system timing is not exact.

The dead-line

It must suffice that a result is achieved before the dead-line. The Response Time is composed by a

- Computation – an upper-bounded or exact time,
- Jitter – an upper-bound time, often statistically described – computation, communication latency, task switching, ...



Missing the dead-line

Hard dead-line

Missing the deadline brings to potentially catastrophic effects

Firm dead-line

Missing the deadline invalidates the result connected with such deadline

Soft dead-line

Missing the deadline does not invalidate the result, however there is a degradation of the “service”

RT Vs Non-RT systems

Guaranteed Timeliness

- Computational load hypothesis is available
- Temporal correctness can be shown analytically
- Coverage must be complete

Best effort

- Analytical argument for temporal correctness cannot be made.
- The temporal verification relies on probabilistic arguments, even within the specified load hypothesis

RT system elements

- Hardware adequate resources - the system must be dimensioned for the worst case scenario.
- Real Time OS
- Real Time communication
- Real Time -compliant programming

RT Operative System

- Allows some process/thread to be shielded from interrupts
- Allows for pre-emption
- Allows for reserving a resource to a specific task

Or no operative system !

Some RTOS for desktops:

- QNX
- RTAI
- XENOMAI
- VXWorks
- Real-time Linux (CONFIG_RT_PREEMPT)

see https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

RT communication

RT communication protocol requires

- determinism
- synchronization
- security

Some RT protocols

- CAN – Control Area Network
- Profibus – Process Field Bus
- AFDX – Avionics Full-Duplex Switched Ethernet
- TTP - Time-Triggered Protocol
- FlexRay
- Real-Time Ethernet (e.g. Powerlink, Ethercat)

Concurrent Programming

Concurrency means that more “Activities” have to run together, with limited resources.
Activities can implemented with

- Processes - different memory
- Multi-threaded process - share the same memory/address space (but each one has his stack - private memory)

Concurrent Programming

Concurrency means that more “Activities” have to run together, with limited resources.

Activities can implemented with

- Processes - different memory
- Multi-threaded process - share the same memory/address space (but each one has his stack - private memory)

Question:

Concurrency == Parallel Execution ?

Concurrent Programming - communication

Inter-Process Communication

- signals
- semaphores
- shared memory (e.g. shm_open)
- queue (e.g.mq_open)
- files
- sockets

Threads Communication

- shared memory
- mutex (pthread_mutex_*)
  sync on data access
- conditions (pthread_cond_*)
  sync on data access and value

Basic multitreading with posix

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr )      {
        char *message;
        message = (char *) ptr;
        printf("%s \n", message);
}
main(){
        pthread_t thread1, thread2;
        char *message1 = "Thread 1";
        char *message2 = "Thread 2";
        int  iret1, iret2;
        /* Create independent threads each of which will execute function */

        iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
        iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

        /* Wait till threads are complete before main continues.  */

        pthread_join( thread1, NULL);
        pthread_join( thread2, NULL);
        exit(0);
}
```

Basic atomic actions with posix

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *functionC(){
        pthread_mutex_lock( &mutex1 );
        counter++;
        printf("Counter value: %d\n",counter);
        pthread_mutex_unlock( &mutex1 );
}
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main(){
        int rc1, rc2;
        pthread_t thread1, thread2;
        /* Create independent threads each of which will execute functionC */
        rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
        rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )

        pthread_join( thread1, NULL);
        pthread_join( thread2, NULL);
        exit(0);
}
```

RT Programming

Also the program needs to adhere to some rules; each program has at least 3 steps

Initialization

Non RT, setup communication, claim memory if not statically allocated.

Running

(Real time) execution of the algorithm(s)

Clean-up

Close communication, close files in a coherent state, clean-up memory

RT Programming

Issues:

- Algorithm must have upper-bounded Worst-Case Execution Time (WCET)
- Some system call are not RT-Safe
  - memory allocation (but see also TLSF allocator, [2])
  - file access
  - writing in a console

Issues from concurrent programming:

- Concurrency
- (Dead-)Lock

RT Programming – Concurrent Programming

Concurrency means that more threads/processes has to share the same resource (CPU Time) in a non-exclusive way.

Some type of scheduling must be applied to solve the concurrency, respecting the deadlines.

Concurrency means that the threads/processes exchange

- data
- events

Typical problems:

- Concurrent access to data protected by mutex (*mutual exclusion*)
- Wait for data

Outline

Scheduling

The study of schedulability assumes 3 steps:

- task models
- Scheduling algorithm
- (Schedulability test for Worst Case Scenario (WCS))

Scheduling Algorithms

Model of tasks

- Periodic (or synchronous) task - hard deadline
  Parameters : T_i period, D_i deadline C_i WCET
- Sporadic (or Asynchronous) tasks - hard deadline
  Parameters : D_i deadline C_i WCET
- Soft or no deadline tasks.

All tasks can have priorities

Scheduling Algorithms

Types of scheduling

- best effort vs Guaranteed
- static vs dynamic
- Preemptive versus non-preemptive
- Single-processor versus multi-processor

Fixed-Priority Scheduling (FPS)

- Each task has a static priority
- Priorities of ready tasks determine the execution order of tasks
- Priorities are derived from temporal requirements

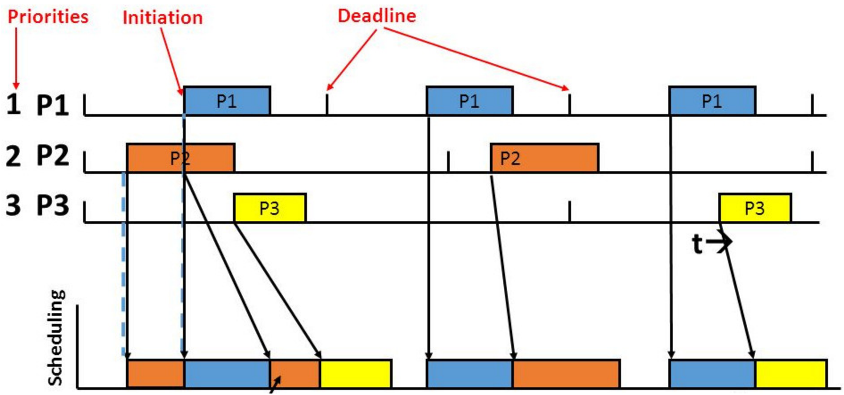
Can be done only with periodic tasks

Fixed-priority pre-emptive scheduling (FPPS)

Is a scheduling system commonly used in real-time systems.

- Each task has a static priority
- Each task in execution can be pre-empted by a higher priority task that is ready
- We can have starvation

Fixed-priority pre-emptive scheduling (FPPS)



Rate-Monotonic Scheduling (RMS)

- Fixed priority scheduling, preemptive
- Rate-monotonic priority assignment: The shorter the period (= the higher the rate) of a task, the higher its priority

if there are only periodic tasks this is a fixed scheduling.

Earliest Deadline First Scheduling (EDF)

- No priority, No Pre-emption
- Absolute deadlines determine the execution order of the tasks
- Selection function: the task with the earliest absolute deadline is selected to execute next

Unpredictable - can cause a domino effect

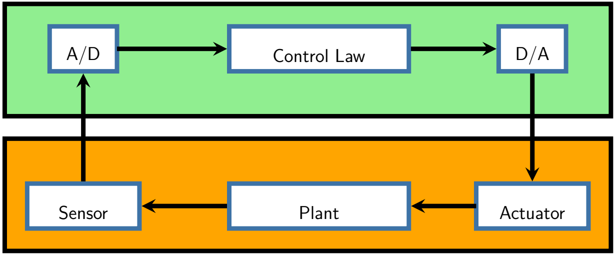
Least-Laxity First Scheduling (LLF)

- Laxity: Difference between deadline and remaining computation time
- Selection function: The task with the smallest laxity gets the highest (dynamic) priority and is therefore selected for executing next

"Optimal" in single processor, periodic task scenario.

Outline

Very simple mechatronic system



It has only one activity at time – unluckily, the one-actuator model is not very realistic. . .

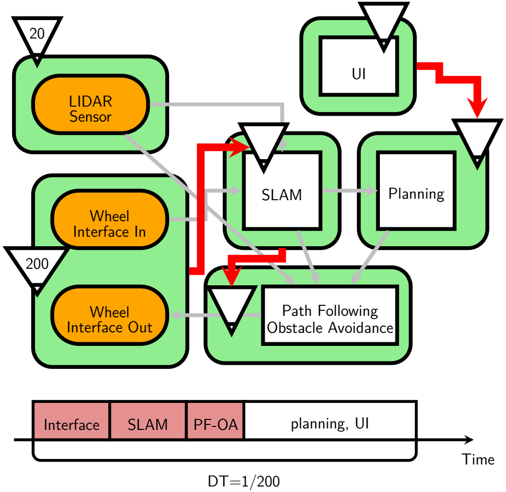
A more realistic mechatronic system

It is composed:

- one or more hardware interfaces
  - minimal loops in external harware (e.g. direct torque control), to
  - complete system providing RPC (e.g. robot with trajectory planner), via (RT) digital communication
- one or more sensors, that produce data at different time.
- critical-safe computations
- other computations (also non-deterministic)

Example – mobile robot navigation

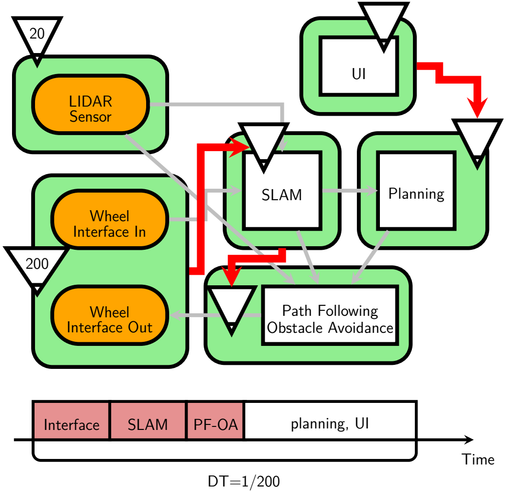
- Contexts in green
- Activities as triangle
- Async activities as red arrows
- data flow in gray arrows



Example – mobile robot navigation

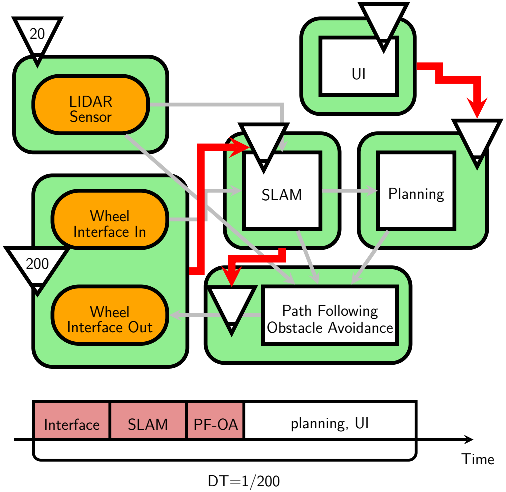
Priorities:

1. The robot interface must run at nominal frequency.
2. Control action must be ready before deadline.
3. The lidar sensor should be read as soon as possible, without interfering the control loop
4. UI should be updated asap.
5. This planning is not deterministic in time.



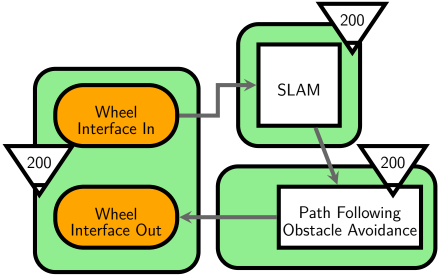
Example – mobile robot navigation

1. RT-high priority, periodic for robot interface
2. Async RT- high priority for SLAM and PF-OA
3. RT-medium priority, periodic for LIDAR
4. (RT-) low priority for the rest.



Example – lazy scheduling

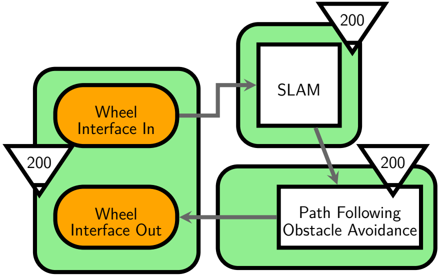
Which would be the worst case scenario timing?



Example – lazy scheduling

Which would be the worst case scenario timing?

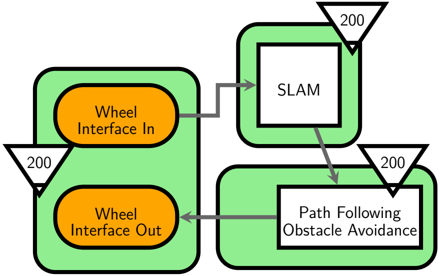
- PF-OA, SLAM, **interface** – 200 ms



Example – lazy scheduling

Which would be the worst case scenario timing?

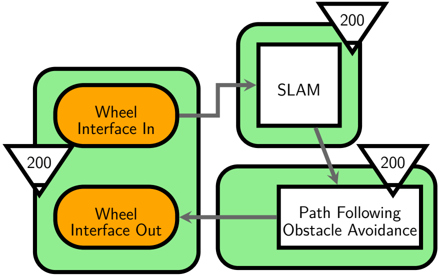
- PF-OA, SLAM, **interface** – 200 ms
- PF-OA, **SLAM**, interface – 200 ms



Example – lazy scheduling

Which would be the worst case scenario timing?

- PF-OA, SLAM, **interface** – 200 ms
- PF-OA, **SLAM**, interface – 200 ms
- **PF-OA**, SLAM, interface – 200 ms



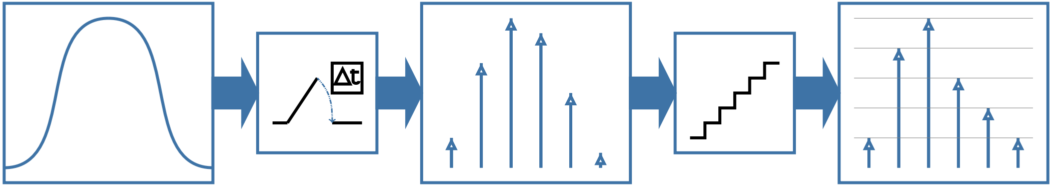
600 ms in place of 200 ms

Discretization of signals

Sensors produces two type of signals

- continuous (e.g. voltage of train gauges)
- discrete (e.g. encoder pulse)

The latter case needs a Discretization/Quantization process :



KU LEUVEN ATLAS

Communication of signals

Good transmission

More or less fixed lantency, that is a fraction of the discretization time.

$$\hat{x}[t] = x[t + \delta t], t \in \Delta T \cdot i, i \in \mathcal{N}, \delta t << \Delta T$$

Bad transmission

- High Latency
- Buffering - time warping of the data

System discretization

When discretizing a system for e.g. design of controllers, we assume

- to know the rate it runs
- that data arrives at the same rate (otherwise account for delay)

To discretize a linear dynamical system a well know approach is to use the the Tustin discretization method

$$C(s) = \frac{b_0 s^{n-1} + \cdots + b_{n-1}}{s^n + a_0 s^{n-1} + \cdots + a_{n-1}} \Rightarrow s = \frac{2}{T}\frac{z-1}{z+1} \Rightarrow C_d(Z)$$

Time is crucial !

Example: the maximum stiffness K achievable by an haptic interface with damping b interacting with a virtual wall is a function of ΔT [1]:

$$b \leq \frac{K\,\Delta T}{2}$$

If ΔT varies, the system becomes unstable.

Take-home messages

- Timing is very important whenever you have control loops
- timing issue can come from non-RT OS, not enough resources, bad programming, communication issues.
- Even if you do not make an explicit derivation of your control but tune "rule of thumb", if rates changes, the system can become unstable.
- better to consider these problems before than try to debug "strange behaviours" after.

Questions ?

Bibliography I

[1] N. Diolaiti et al. "Stability of Haptic Rendering: Discretization, Quantization, Time Delay, and Coulomb Effects". In: *Trans. Rob.* 22.2 (Apr. 2006), pp. 256–268. ISSN: 1552-3098. DOI: 10.1109/TRO.2005.862487. URL: https://doi.org/10.1109/TRO.2005.862487.

[2] M. Masmano et al. "TLSF: a new dynamic memory allocator for real-time systems". In: *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* July 2004, pp. 79–88. DOI: 10.1109/EMRTS.2004.1311009.