Software Aspects in Control Systems

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it

Robotic Systems
The implementation of control systems is based on an algorithm that is characterised by the execution of a **timed loop** of a set of activities:

- Starting of the activities on the basis of a **sampling time**, $\Delta T$
- Reading of input variables
- Execution of one step of the control algorithm
- Writing of the results to the outputs

**Algorithm**

```python
while True do
    On each $\Delta T$;
    $in \leftarrow read\_input()$;
    $out \leftarrow compute\_control()$;
    $write\_output(out)$;
end
```
Implementation of a Control System

When the whole system is made of several sub-systems, at first sight, each of them must be implemented in a **different (computational) loop**, and all the loops should be executed **concurrently**.

Since the sub-systems are interconnected, a certain **form of communication** between the “loops” must be implemented: in the case in figure, variable *out1* of S1 is also the input of *in2* of S2.
However, if the periods are the same $\Delta T_1 = \Delta T_2 = \Delta T$, we can “fuse” both the loops:

```
while True do
    On each $\Delta T$;
    in1 ← read_input1();
    out1 ← compute_S1();
    in2 ← out1;
    out2 ← compute_S2();
    write_output2(out2);
end
```
At the same time, when one of the intervals is an **integer multiple** of the other, $\Delta T_1 = n\Delta T_2$, we can implement the system as:

**System S1+S2**

```
while True do
    On each $\Delta T_2$;
    count ← count + 1;
    if count = n then
        count ← 0;
        in1 ← read_input1();
        out1 ← compute_S1();
    end
    in2 ← out1;
    out2 ← compute_S2();
    write_output2(out2);
end
```
Choosing the Sampling Time for Sensors

The Fourier Series

Any (periodic) signal \( s(t) \) can be represented as a linear combination of sinusoids and cosinusoids with coefficients \( a_i \) and \( b_i \):

\[
s(t) = a_0 + \sum_{n=1}^{N} \left[ a_n \cos\left(\frac{2\pi}{T} nt\right) + b_n \sin\left(\frac{2\pi}{T} nt\right) \right]
\]

In other words, the original signal can be constructed using a linear combination of sinusoids at different frequencies.

A signal coming from a sensor can be considered a signal of the type indicated.
Choosing the Sampling Time for Sensors

Nyquist–Shannon Sampling Theorem

Any (periodic) signal $s(t)$ can be **reconstructed** when it is **sampled** at a frequency $f_{sample}$ that is **at least two times** the frequency of the sinusoid with maximum frequency.
Choosing the Control Period

- The **Control Period** is used in the discretization of a system, in which the state matrix $A$ becomes:

$$
A' = A \Delta T + I
$$

- Since the stability and the behaviour of the discretized system depend on the **eigenvalues** of $A'$, the choice of $\Delta T$ plays a fundamental role.

- Theoretically $\Delta T$ must be chosen by meeting the following (give $x$ the state vector):

$$
\frac{\Delta x}{\Delta T} \approx 0
$$
Task Subdivision

Task Subdivision
Software organisation of a control system

- A set of **tasks**, each implementing a single sub-system
- A **scheduling environment** able to execute such tasks each with its **own period**
- A **communication environment** among tasks, able to support data interchange and synchronisation among tasks
Let us consider the system in figure and suppose that $S_2$ has a period $\Delta T$, and $S_1$ has a period $4\Delta T$.

**Task_S1()**

\[
\begin{align*}
\text{in1} &\leftarrow \text{read\_input1}(); \\
\text{out1} &\leftarrow \text{compute\_S1}(); \\
\text{write\_output1(out1)};
\end{align*}
\]

**Task_S2()**

\[
\begin{align*}
\text{in2} &\leftarrow \text{read\_input2}(); \\
\text{out2} &\leftarrow \text{compute\_S2}(); \\
\text{write\_output2(out2)};
\end{align*}
\]
Let us consider the system in figure and suppose that $S_2$ has a period $\Delta T$, and $S_1$ has a period $4\Delta T$. 

**System S1+S2**

```python
while True do
    On each $\Delta T$;
    count ← count + 1;
    if count = 4 then
        count ← 0;
        Task_S1();
        in2 ← out1;
    end
    Task_S2();
end
```

**Components**

- **Red**: Tasks
- **Green**: Scheduler
- **Blue**: Communication
Structured Models

- A function (task body) that implements the behaviour of the single system
- A data structure that embeds the data about the state of the task/system

Object Model

- A class that represents the sub-system with...
- a main method (e.g. `run()`) that implements the behaviour of the single system
- a set of attributes that embeds the data about the state of the task/system
Managing Periods

A **timer hardware**, configured using the period $\Delta T$, with a procedure that is activated when the timer elapsed:

**Adopted Solutions**

- A shared global variable, in polling
- A *callback* procedure associated to the timer
- Invocation of blocking procedure that *waits for* the timer event

In any case, a library framework, or an operating system, is needed, able to offer the management of the timer.
bool timer_elapsed ← false;

TimerISR () begin
    timer_elapsed ← true;
end

Main () begin
    while True do
        if timer_elapsed then
            timer_elapsed ← false;
            // Do the control tasks
        end
    end
end
Callback Procedure associated to the Timer

```
TimerCallback() begin
  // Do the control tasks
end

Main() begin
  SetTimerCallback(\Delta T, TimerCallback);
  // ... do other things
end
```
Blocking Procedure waiting for the Timer Event

Main () begin
    SetupTimer(ΔT);
    while True do
        WaitTimerEvent();
        // Do the control tasks
    end
end
Adopted Solutions

- Shared Variables, with critical sections when the access is done by concurrent tasks
- References among objects (when the implementation is object-oriented), with critical sections if needed
- Use of a communication middleware
Communication Middleware for Control Systems

Publisher/Subscriber Model (Data Distribution Model)

- Exchanged data are modelled by means of **structured types** and identified by a **topic**
- Tasks interested in a certain **topic** call a **subscribe** function by specifying the topic itself

![Diagram](image-url)
The tasks producing a data with a certain *topic* calls a `publish` function, specifying topic and *data*.

Tasks which have a subscription receive `notify` with published data.
Some Communication Middlewares

- **uORB**: communication middleware for embedded systems (centralised)
- **CORBA-DDS (Data Distribution Service)**: communication middleware based on a standard by the Object Management Group (centralised and distributed)
- **ROS (Robot Operating System)**: communication middleware specifically designed for robotic systems (centralised and distributed)
- **MQTT (Message Queue Telemetry Transport)**: a lightweight standard communication protocol (publisher/subscriber) specifically designed for IoT systems (centralised and distributed)
Control System using the Publisher/Subscriber Model

Task S1

while True do
  On each $\Delta T_1$;
  $in1 \leftarrow \text{read\_input1}()$;
  $out1 \leftarrow \text{compute\_S1}()$;
  $\text{publish}(\text{"mydata"}, out1)$;
end

Task S2

subscribe(\text{"mydata"});
while True do
  $in2 \leftarrow \text{wait\_data}(\text{"mydata"})$);
  $out2 \leftarrow \text{compute\_S2}()$;
  $\text{write\_output2}(out2)$;
end
Real-Time Operating Systems
The execution of control tasks requires **guaranteed times**, otherwise the consequences may be dangerous (above all when the system is safety-critical).

If $\Delta T_c$ is the “worst-case” execution times of a task and $\Delta T$ its period, then the following must be met: $\Delta T_c < \Delta T$

However, the remaining time $\Delta T - \Delta T_c$ must be such that the system can perform other activities.
Feasibility Condition

Feasibility condition of $N$ periodic tasks:

$$\sum_{i=1}^{N} \frac{\Delta T_{ci}}{\Delta T_i} < 1$$

where $\Delta T_{ci}$ is worst-case execution time of the task $i$ and $\Delta T_i$ is the periodo of the task $i$
Characteristics of Schedulers in Multi-tasking RTOSs

- Task Scheduling in RTOS is performed by means of **task priorities**
- Priorities are **fixed** and does not change as it happens instead in “general-purpose” operating systems
- The **priority** is assigned (statically or dynamically) on the basis of the **time characteristics** of the task (i.e. the more “urgent” the task the higher its priority)
Scheduling Policies in RTOS

- **Round-Robin (RR) with Priority**: the scheduler selects the READY task with the highest priority and executes it, preempting it at the next “scheduling tick” (or if the task goes autonomously in “sleep” due to a blocking system call).

- **FIFO (with Priority)**: the scheduler selects the READY task with the highest priority and executes it, preempting it only due to a blocking system call.
Some RTOSs

- **FreeRTOS.** Real-time Kernel for embedded systems (open-source)
- **NuttX.** Real-time Kernel for embedded systems (open-source, used in autopilots of drones)
- **RTAI.** Real-time Linux Kernel (open-source)
- **QNX.** Real-time Kernel Unix-like (proprietary)
- **VxWorks.** Real-time Kernel Unix-like (proprietary)
Linux System Call to control Scheduling

Set of the Scheduling Policy

```c
int sched_setscheduler(pid_t pid, int policy, 
                        struct sched_param * param);
```

- **pid**, the process identifier of which we want to change the scheduling policy (0 = “this process”)
- **policy**, the scheduling policy: SCHED_OTHER, SCHED_RR, SCHED_FIFO
- **param**, additional parameters, among them the **priority** (from 1 to 99)

Priority

```c
struct sched_param {
    ... 
    int sched_priority;
    ... 
};
```
Custom Scheduling Example

A main that runs 3 children

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

int child_process(int id)
{
    ...
}

int main(int argc, char **argv)
{
    int i, status;
    printf("Starting 3 children...
");
    for (i = 0; i < 3; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            child_process(i);
            exit(0);
        }
    }
    printf("Waiting...
");
    while (wait(&status) > 0) {};
    printf("End...
");
}
```
int child_process(int id)
{
    struct sched_param params;
    int i, k;

    #ifdef FIFO
    params.sched_priority = sched_get_priority_max(SCHED_FIFO);
    if (sched_setscheduler(0, SCHED_FIFO, &params) < 0) {
        perror("cannot set the scheduler");
    }
    printf("Setting priority %d\n", params.sched_priority);
    #endif

    #ifdef RR
    params.sched_priority = sched_get_priority_max(SCHED_RR);
    if (sched_setscheduler(0, SCHED_RR, &params) < 0) {
        perror("cannot set the scheduler");
    }
    printf("Setting priority %d\n", params.sched_priority);
    #endif

    usleep(500000);
    printf("Child %d started\n", id);
    for (i = 0; i < 10; i++) {
        /* 10 iterations */
        printf("Child %d, iteration %d\n", id, i);
        for (k = 0; k < 100000000; k++) {}
    } /* losing time ... */
    return 0;
}
Test with SCHED_OTHER

$ taskset --cpu-list 1 sudo ./sched_test
Starting 3 children...
Waiting...
Child 2 started
Child 2, iteration 0
Child 1 started
Child 1, iteration 0
Child 0 started
Child 0, iteration 0
Child 1, iteration 1
Child 2, iteration 1
Child 0, iteration 1
Child 2, iteration 2
Child 0, iteration 2
Child 1, iteration 2
Child 2, iteration 3
Child 0, iteration 3
Child 1, iteration 3
Child 2, iteration 4
....
End...
Test with SCHED_FIFO

$ taskset --cpu-list 1 sudo ./sched_test
Starting 3 children...
Waiting...
Setting priority 99
Setting priority 99
Setting priority 99
Child 2 started
Child 2, iteration 0
Child 2, iteration 1
Child 2, iteration 2
...
Child 2, iteration 9
Child 1 started
Child 1, iteration 0
Child 1, iteration 1
Child 1, iteration 2
...
Child 1, iteration 9
Child 0 started
Child 0, iteration 0
Child 0, iteration 1
Child 0, iteration 2
...
Child 0, iteration 9
End...
The scheduling policies allow a developer the implementation of soft real-time tasks in Linux.

But some parts of the Linux kernel are not pre-emptible thus uncontrollable latencies can occur.

For example, the management of virtual memory (swapping) introduces unpredictable latencies.

**RTLinux** overcomes such limits by patching some parts of Linux kernel thus allowing the execution of hard real-time tasks.
RTAI

- RTAI is widely used to support real-time processes on Linux.
- It is a kernel patch that uses the “virtualisation” model.
- The Linux kernel is replaced by a micro-kernel upon which both the “classic” Linux kernel and the real-time kernel (added by RTAI) run.
From the release 3.0, the Linux kernel has been made preemptible through a patch provided by the Linux Foundation(*).

This patch, with the preemption, offers a series of systems calls to support real-time tasks:

- Support for timer and timer-based task
- Scheduling policies
- Stack control
- Memory control

(*) = https://wiki.linuxfoundation.org/realtime/start
Software Aspects in Control Systems

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it

Robotic Systems