NTNU

Innovation and Creativity

# Design Patterns for Communicating Systems with Deadline Propagation

Martin Korsgaard and Sverre Hendseth

Department of Engineering Cybernetics

# Contents

I Explain Toc: occam with TIME-construct for real-time programming.

II Show how certain types of communication can distort timing.

III Introduce design patterns that helps to avoid this.

IV (Demonstrate schedulability analysis on Toc programs using these patterns.)

NTNU
Innovation and Creativity

# I Traditional Real-time Programming

*Given a specification of tasks with deadlines, how to implement?*

Traditional approach:

1. Each task is converted into a thread.
2. Periods are implemented using sleeps or delays.
3. Deadlines are converted into relative priorities.
4. Communication uses monitors/mutexes/etc., which are unaware of timing and leads to unbounded inversion problems.
5. Priority inheritance or ceiling protocols needed to fix these.

NTNU
Innovation and Creativity

# Traditional Real-time Programming (2)

The tradition is not all right:

1. Bad Abstraction: Threads, priorities and delays are low-level primitives. Correct use is difficult, and the flexibility allowed by these primitives is not needed.
2. No Reflection: The transformation from tasks and deadlines to threads and priorities is irreversible: Information is lost and the implementation does not reflect the timing requirements of the specification.
3. Complex Synchronization: Priority inheritance or ceilings lead to complex scheduling rules, making it difficult to predict scheduling behaviour in unexpected situations such as an overloaded system.

NTNU
Innovation and Creativity

# Bad abstraction

The concurrency primitives cannot be used to specify timing requirements directly.

```
void thread()
{
  set_priority(5);
  next := clock();
  while(1) {
    do_something();
    next := next + 20;
    delay_until(next - clock());
  }
}
```

Where is the timing requirement?

NTNU
Innovation and Creativity

# Introduction to Toc

The Toc Approach:

1. Timing requirements are specified as deadlines directly in code.
2. Scheduling uses EDF $\Longrightarrow$ no priorities.
3. Synchronous communication using channels with deadline propagation. Inheritance protocol is implicit $\Longrightarrow$ no need for extra rules.
4. Toc is lazy and does not execute primitive processes without a deadline. Considering timing requirements is not optional.

NTNU
Innovation and Creativity

# Toc `TIME` Construct

The construct

```
TIME x
  P()
```

creates a process which is

1. scheduled with relative deadline `x`
2. *and* which is not allowed to terminate before its deadline.

(The scheduler cannot force programs to complete within their deadlines but it will enforce 2.)

NTNU
Innovation and Creativity

# Tasks in Toc

The following therefore creates a periodic task with a period and relative deadline of 10 ms:

```
WHILE TRUE
  TIME 10 MS
    Task.body()
```

Periodic task with period 100 ms and relative deadline of 10 ms.

```
WHILE TRUE
  TIME 100 MS
    TIME 10 MS
      Task.body()
```

NTNU
Innovation and Creativity

# Lazy Scheduling

## Definition (Lazy scheduling)

Lazy scheduling means that no statements are executed without an associated deadline.

## Hypotheses

- Every task in a real-time system can be given a deadline.
- Background tasks with undefined timing requirements are never needed.
- Programmers should be forced to consider the timing requirements of all functionality in the system.

NTNU
Innovation and Creativity

# Laziness in Toc

Toc is lazy and does not execute primitive processes unless driven by a deadline.

```
PROC Main()
  SEQ
    a := 10
    P(a)
:
```

No deadline; will never be run.
Entire program equal to STOP.

(In fact, every occam program, when interpreted as a Toc program, is semantically equal to STOP.)

NTNU
Innovation and Creativity

# Deadline Propagation

Scheduling of dependent processes is handled through deadline propagation.

## Definition (Deadline Propagation)

A process blocked by a channel that is not ready will transfer its deadline through the channel to the process blocking on the other end.

- In effect, if an early deadline task is blocked, code to unblock it is executed with the early deadline.
- When an early deadline task is ready, processes are only executed that help that deadline being reached.
- The result is an implicit priority inheritance protocol over channels.

NTNU
Innovation and Creativity

# Order of Execution: Passive Server

```
PAR
1.  TIME 10 MS
      SEQ
3.      request ! 0
        P1()
5.      reply ? x
        P3()
2.  ALT
      request ? x
4.      SEQ
          P2()
          reply ! 0
          P.lazy()
```

# II Distorted timing

Programs may run into several issues that may distort the intended timing.

1. Tasks may stall, waiting for the minimum-execution time property of another task.
2. Processes may be driven by another task than intended, undermining the given deadline, making the system harder to schedule.
3. The systems or a subsystem may deadlock.

NTNU
Innovation and Creativity

# Direct communication distorts timing

In general, two tasks must not communicate directly.

```
                        PAR

WHILE TRUE                      WHILE TRUE
  TIME 10 MS                      TIME 50 MS
    SEQ                             SEQ
      ...                             ...
      ch ! 0                          ch ? x
      ...                             ...
```

Will stall

50 ms deadline effectively ignored

- The left-hand task will stall and (nearly) always miss its deadline.
- The right-hand task will partly execute with the deadline of the left, undermining the given deadline specification.

NTNU
Innovation and Creativity

# Deadlock in Deadline-driven systems

A deadlock in a lazy, deadline-driven system like Toc is slightly different from a deadlock in a strict RR system (like occam):

— In Toc, A task that requires communication over a channel is never blocked; it simply defers its execution to the task that it is waiting for.

This can only fail if a process — through others — passes a deadline onto itself, so a deadlock equals circular propagation of a deadline.

— A possible circular wait cannot deadlock unless it comes with a circular deadline propagation: i.e. code may be too lazy to deadlock

NTNU
Innovation and Creativity

# Too lazy to deadlock

The following process never deadlocks, because of laziness.

```
PAR
  STOP
  WHILE TRUE
    TIME 10 MS
      P()
```

"STOP" representing a situation that deadlocks

# III Design Patterns

Communication between tasks must be restricted to avoid deadlocks or distorted timing. Using design patterns can aid in this.

The design patterns presented here are designed to

— Be flexible and useful for programming most real-time applications.

— Allow analysis of programs w.r.t. deadlocks.

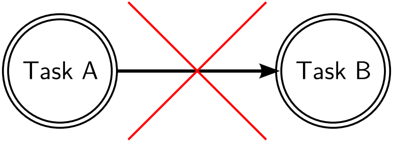
— Allow analysis of programs w.r.t. schedulability.

NTNU
Innovation and Creativity

# Tasks

The first pattern is the task.

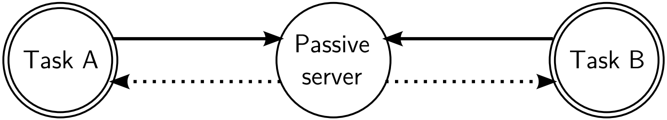
- A task is a process with one `TIME` construct, or two nested `TIME` constructs.
- It may be periodic or sporadic.
- A task is depicted as a double circle.
- A task may not communicate directly with other tasks:



# Passive Server

A passive server has no `TIME` construct, and executes only when driven (possibly indirectly) by a task.

- A task may accept requests from one or more clients. The protocol with the client may a reply.
- Sharing a passive server between clients implies synchronization between those clients and may inevitably lead to deadline inversion.



# Deadlocks in Passive Server Networks

The client-server paradigm for deadlock-freedom in occam applies to networks of tasks and passive servers in Toc.

1. No client communicates with other processes between a request to a server and the corresponding reply.
2. No server accepts new requests between a request and a reply, but may send requests to other servers acting as a client.
3. The client-server relation graph must be acyclic.

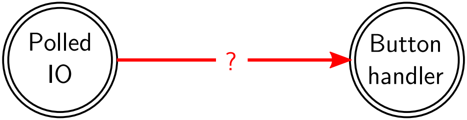
To avoid one client stalling another, it is also required that no servers may be held across multiple task instances.

NTNU
Innovation and Creativity

# Sporadic tasks

Many types of tasks are sporadic rather than periodic, in that they are not started until triggered by some other event.

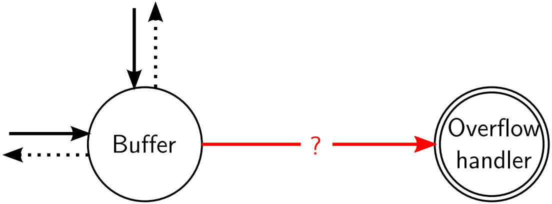
## Example (Button Polling)



# Sporadic tasks (2)

Sporadic tasks should be allowed to be triggered from passive servers as well.

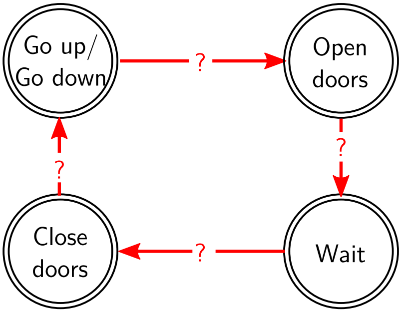
## Example (Error Handling)



# Sporadic tasks (3)

Sporadic tasks should be able to trigger each others cyclically.

## Example (Simple Elevator Model)



# Implementing triggers

A straight-forward channel trigger is insufficient:

```
PAR
```

```
WHILE TRUE
  TIME 10 MS
    SEQ
      ...
      IF
        condition
          trigger ! 0
```

Will stall if 1/2 or more instances cause trigger

```
WHILE TRUE
  trigger ? x
    TIME 20 MS
      Sporadic()
```

NTNU
Innovation and Creativity

# Events

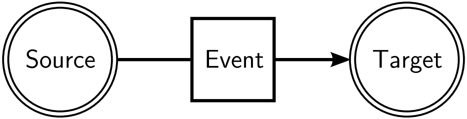
A trigger of a sporadic task should have event-like behaviour.

- An event is a message from one task (the source) that may trigger another task (the target).
- An event must be able to drive an idle sporadic task to start (synchronous message)
- The source of an event must never stall when outputting an event.
- The timing of the target sporadic task should not be affected by an incoming event, and the target can never drive the source to send an event. (asynchronous message)

NTNU
Innovation and Creativity

# Event process

Because two tasks are not allowed to communicate directly, an intermediate event process between the source and the target is needed.



# Event rules

Two rules lead to desired properties:

1. The target must notify the event process that is ready, after which it cannot communicate with other processes before triggered by a new event.
2. A trigger from the source should never drive the target task. This implies that outputting a trigger will never stall the source.

No deadline from the source can propagate beyond the target $\implies$ an event can never be involved in a deadlock.

NTNU
Innovation and Creativity

# Implementing Events

## Example (Wake-up event)

PAR

```
BOOL busy:
WHILE TRUE
  ALT
    source ?? x
      IF
        NOT busy
          event.trigger ! 0
      busy := TRUE
    target.ready ? x
      busy := FALSE
```

```
WHILE TRUE
  SEQ
    target.trigger ? x
    TIME 20 MS
      SEQ
        Sporadic()
        target.ready ! rdy
```

NTNU
Innovation and Creativity

# IV Execution Time Analysis

Worst-case execution time analysis is required for some safety-critical real-time systems.

- Execution time analysis of synchronously communicating systems is not well developed.
- Analysis usually assume threads/locks.
- Ada Ravenscar (safety-critical profile) prohibits synchronous communication because of this.
- However, we have developed a method for WCET analysis of Toc programs, given that the presented design patterns are being used.

NTNU
Innovation and Creativity

# Execution Time Analysis (2)

Our current analysis is based on the traditional model for schedulability of EDF systems. The traditional model requires

1. Fixed set of periodic tasks with $D = T$.
2. Tasks are independent.
3. System is schedulable iff

$$\sum_i C_i/T_i \leq 1$$

$C_i$ and $D_i$ is the computation and deadline/period of task $i$, respectively.

NTNU
Innovation and Creativity

# Execution Time Analysis (3)

Procedure:

1. Enforce $D = T$. (No nested `TIME` constructs) (unfortunate)
2. Assume that all sporadic tasks are fully periodic. (pessimistic)
3. Treat event processes as servers to both source and target.
4. Augment $C_i$ to include the worst-case execution time of dependent processes due to deadline-propagation.
5. System is schedulable iff

$$\sum_i C_i / T_i \leq 1$$

*Equations are in the proceedings.*

NTNU
Innovation and Creativity

# Summary

1. Toc is a real-time programming language building on occam, where specification of timing requirements is treated as an fundamental part of the language
2. Careless communication between tasks in Toc can distort timing.
3. This can be avoided by using a small set of design patterns.
4. Schedulability analysis is possible for Toc when using only these design patterns.

## Some Future Work

1. Formal analysis of lazy, deadline-driven systems.
2. Check if schedulability analysis can be extended to other types of synchronously communicating processes.

NTNU
Innovation and Creativity

# Questions

NTNU
Innovation and Creativity

Innovation and Creativity

# Timing in Toc

The start-time of a `TIME` construct is the time of the event that caused its evaluation.

```
WHILE TRUE
  TIME 10 MS
    P()
```

Cause of execution of TIME is end of previous instance => no drift

```
WHILE TRUE
  SEQ
    ch ? x
    TIME 10 MS
      P(x)
```

Cause of execution of TIME is communication

NTNU
Innovation and Creativity

# Laziness and Extended Rendezvous

```
PROC id(CHAN FOO left?, right!)
  FOO x:
  WHILE TRUE
    SEQ
      left ? x
      right ! x
:
```

Deadline on input will not drive output

```
PROC id(CHAN FOO left?, right!)
  FOO x:
  WHILE TRUE
    left ?? x
      right ! x
:
```

NTNU
Innovation and Creativity

# Circular Propagation is Deadlock

```
PAR
  SEQ
    TIME d
      ch1 ! 0
    chx ? x
  SEQ
    ch2 ! 0
    ...
    ch1 ? x
  SEQ
    ch3 ! 0
    ...
    ch2 ? x
  SEQ
    ch4 ! 0
    ...
    ch3 ? x
  ...
                    ?
```

— The deadline propagates to the right.

— If any of the parallel processes writes to `chx`, the left task will be driven indirectly by its own deadline (circular propagation) and the system deadlocks.

# Preemptions

An earlier task that becomes ready will preempt (take over execution from) a later deadline task.

For two tasks $A$ and $B$ where $D_A < D_B$:

1. $A$ can preempt $B$ at most $\lfloor D_B / D_A \rfloor$ times.
2. A single instance of $A$ can preempt $B$ at most once.

Depending on the system state at time of preemption, there may be an execution time penalty for both being preempted and for preempting another task.

NTNU
Innovation and Creativity

# Timing of Passive Server

*Client*

```
TIME
  SEQ
    ...
    request ! ...
    Work()
    reply ? ...
    ...
```

$C_{client}$

*Server*

```
WHILE TRUE
  SEQ
    Prepare()
    ALT i = 0 FOR N
      request[i] ? ...
        SEQ
          Process()
          reply[i] ! ...
          Cleanup()
```

Phase 1

$C_{request}$

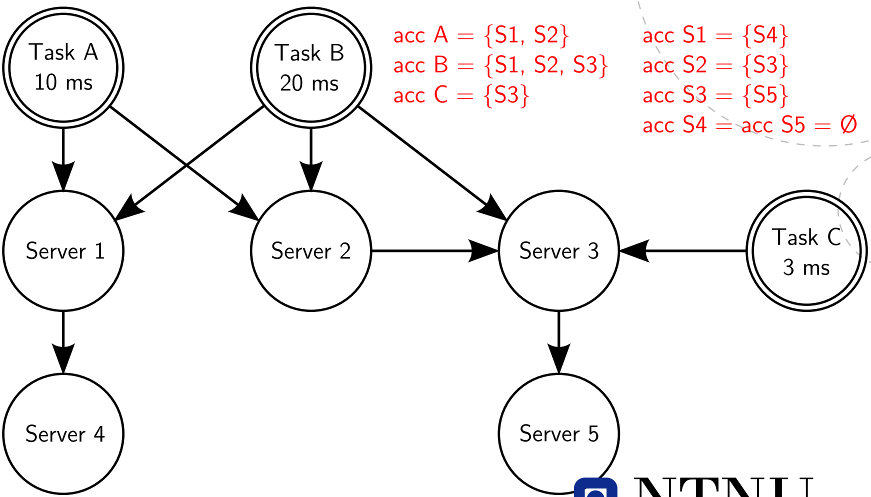
Phase 2

$C_{reply}$

NTNU

Innovation and Creativity

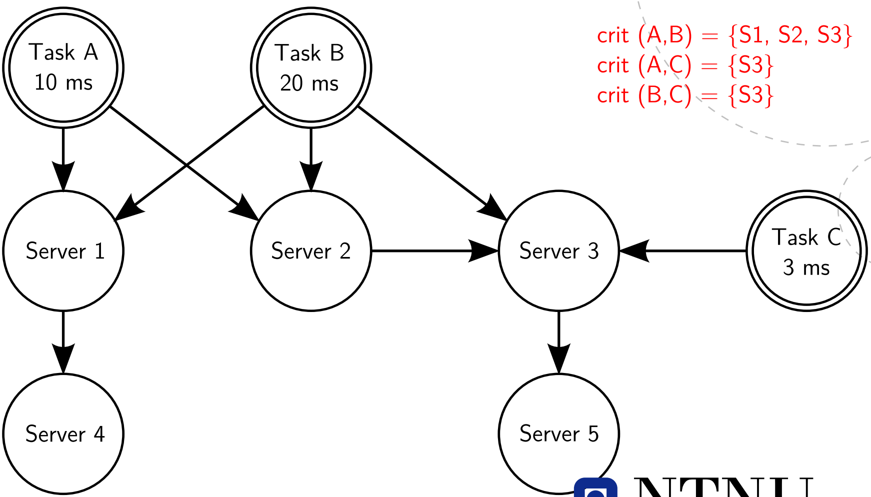
# Access Set for a Process




acc A = {S1, S2}
acc B = {S1, S2, S3}
acc C = {S3}

acc S1 = {S4}
acc S2 = {S3}
acc S3 = {S5}
acc S4 = acc S5 = ∅

# Critical Processes for a Task Pair




crit (A,B) = {S1, S2, S3}
crit (A,C) = {S3}
crit (B,C) = {S3}

NTNU
Innovation and Creativity

# Timing of Passive Server (expression)

Server $s$ may access other servers, so $C_{s,reply}$ can therefore be given as

$$C_{s,reply} = \widehat{C}_{s,reply} + \sum_{s' \in \text{acc } s} \left( C_{s',request} + C_{s',reply} \right)$$

where $\widehat{C}_{s,reply}$ is the part of the execution local to $s$.

This is a recursive formula over the set of servers which will always terminate if the client-server graph is acyclic.

NTNU
Innovation and Creativity

# Big ET Equation

$$
\begin{aligned}
C_A = \widehat{C}_A &+ \sum_{s \in \text{acc}\, A} \left( C_{s,request} + C_{s,reply} \right) \\
&+ \sum_{X \in T, D_X < D_A} \left\lfloor \frac{D_A}{D_X} \right\rfloor \max_{s \in \text{crit}\,(A,X)} C_{s,request} \\
&+ \sum_{X \in T, D_X > D_A} \max_{s \in \text{crit}\,(A,X)} \left( C_{s,client} + C_{s,reply} \right)
\end{aligned}
$$

NTNU
Innovation and Creativity