Real-Time Component Software

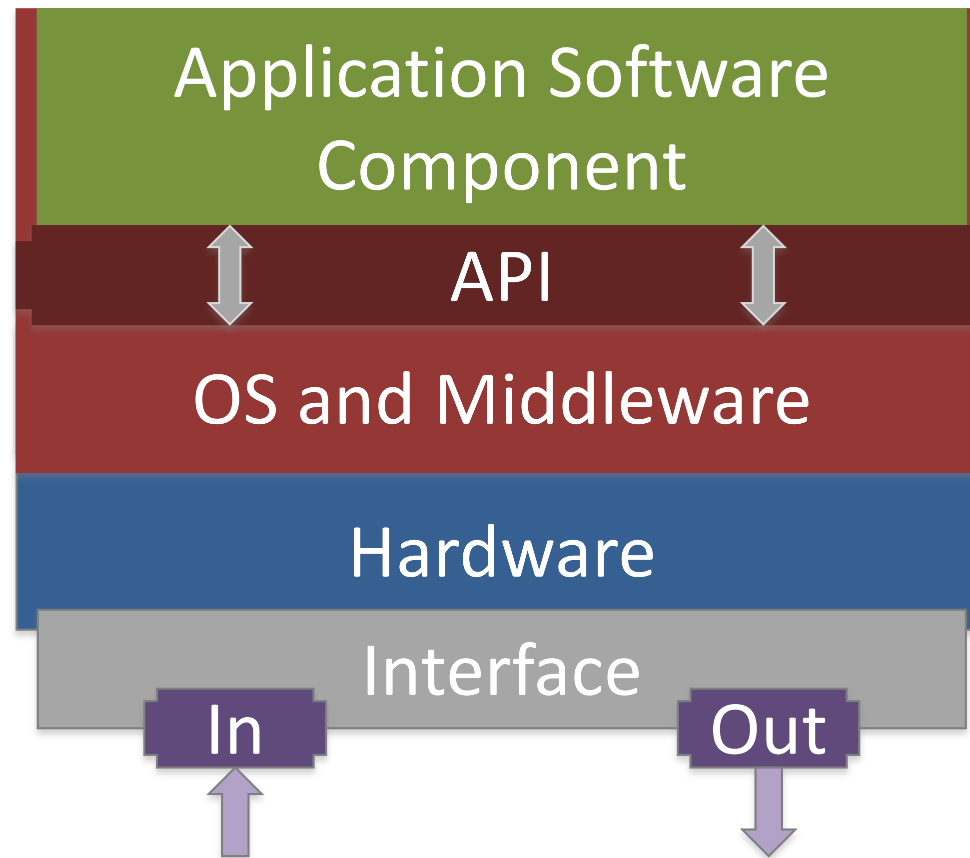
slide credits: H. Kopetz, P. Puschner

TU WIEN

Overview

- OS services
- Task Structure & Interaction
- Input/Output
- Error Detection
- TT Component

Operating System and Middleware



OS Services

- Secure boot of component software
- Reset, (re)start, reintegration of component SW
- Task management
- Task interaction
- Message communication interface
  - Linking Interface (RTS)
  - Config., control (TII), debug (TDI)
- I/O Handling
  - Discretization
  - Agreement

Time predictability, determinism

TU WIEN

Assumptions about Software

HRT Software

- Closed world assumption
  - Tasks and task timing parameters known at design time
  - Task communication, precedence known at design time
  - I/O requirements known (values, timing)

⇨ Pre-runtime preparation/analysis to provide runtime guarantees

SRT Software

- Open world assumption
  - Tasks, task timing and QoS parameters, I/O requirements

⇨ QoS assessment before runtime; at runtime: best effort

TU WIEN

Task Management

- Component software: set of tasks that run in parallel.
- OS provides the execution environment for each task.
- Temporal and spatial isolation: HRT Software versus other SW
- HRT Tasks are cooperative, not competitive.
- Component = unit of failure
  - No resource-intensive protection between HRT tasks
  - Light-weight OS
  - Stateless versus stateful tasks

Time Services

Clock synchronization, Sparse time

Timed services:

- Triggering actions, event time-stamping, duration measurement
- Message I/O
- Modeling physical second and calendar service

Role of event-occurrence time

- Time as data: timestamp of value / value change of RT entity.

  Example: timekeeping in downhill skiing

- Time as control: computer system reacts immediately to event.

  Example: Emergency stop

TU WIEN

Fault Tolerance, Redundancy

Determinism (1st attempt)

A model behaves deterministically if and only if, given a full set of initial conditions (the initial state) at time t_o, and a sequence of future timed inputs, the outputs at any future instant t are entailed.

- Definition of determinism is intuitive,
- neglects the fact that in a real (physical) distributed system clocks cannot be precisely synchronized,
- therefore a system-wide consistent representation of time (and consequently state) cannot be established.

Determinism

Let us assume

- *Q is a finite set of symbols denoting states*
- *Σ is a finite set symbols denoting the possible inputs*
- *Δ is a finite set of symbols denoting the possible outputs*
- *$q_0 \in Q$ is the initial state*
- *$t_i \in N$ is the infinite set of active sparse time intervals*

then a model (*processing, communication*) is said to behave *deterministically* **iff, given** a sequence of *active sparse real-time intervals t_i*, the initial state of the system $q_0(t_0) \in Q$ *at t_0 (now),* and a sequence of *future* inputs $in_i(t_i) \in \Sigma$ **then** the sequence of *future* outputs $out_j(t_j) \in \Delta$ and the sequence of future states $q_j(t_j) \in Q$ is *entailed.*

TU WIEN

Replica Determinism

A set of replicated RT-objects is replica determinate if all objects of this set visit the same state within a specified interval of real time and produce identical outputs.

The time interval of this definition is determined by the precision of the clock synchronization.

Replica Determinism

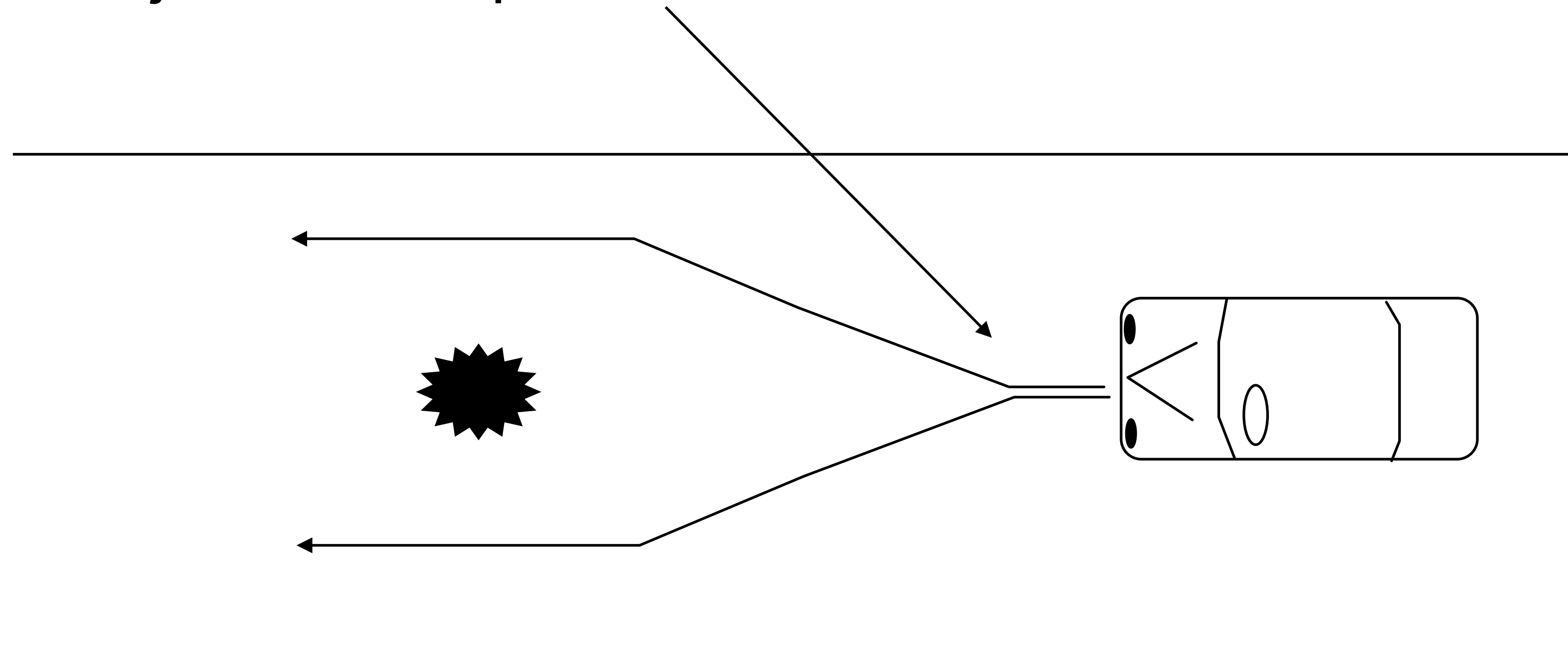
Replica Determinism is needed for the following reasons:

- To implement consistent distributed actions.
- To improve the testability of systems – tests are only reproducible if the replicas are deterministic.
- To facilitate the implementation of fault tolerance by active replication.

Replica Determinism helps to make systems more intelligible!

Major Decision Point

How can we make sure, that both replicas take the same decision at this major decision point?



TU WIEN

Replica Determinism: Destroying Factors

Replica determinism can be destroyed by:

- Differing inputs (inconsistent order or sensor variations)
- Non-deterministic program constructs
- Dynamic preemptive scheduling decisions
- Explicit synchronization statements (e.g., *Wait*)
- Uncontrolled access to the global time and timeouts
- Differing processing speeds (diff. crystal resonators, clocks)
- Consistent comparison problem (software diversity)

This list is not complete!

Support for Replica Determinism

- Sparse value/time base
- Static or non-preemptive scheduling
- exact arithmetic
- agreement on input data and order

Error Detection Mechanisms

An RTOS must provide error detection in the temporal domain and in the value domain

- Consistency checks, CRC checks
- Monitoring task execution times
- Monitoring interrupts (MINT)
- Double execution of tasks (time redundancy)
- Watchdogs – observable heart-beat signal

TU WIEN

Task Models and Control

Simple task (S-Task)

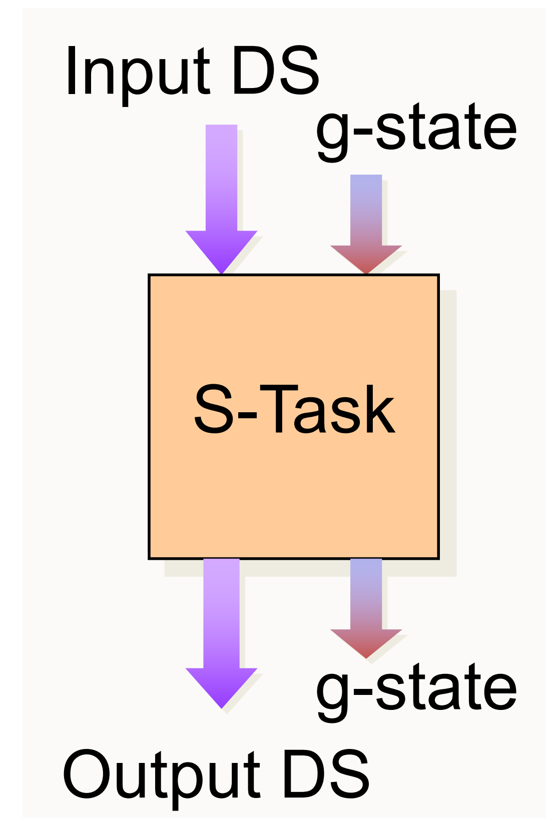
- executes from the beginning to the end without any delay, given the CPU has been allocated.

Complex task (C-Task)

- may contain one or more *WAIT* statements in the task body.

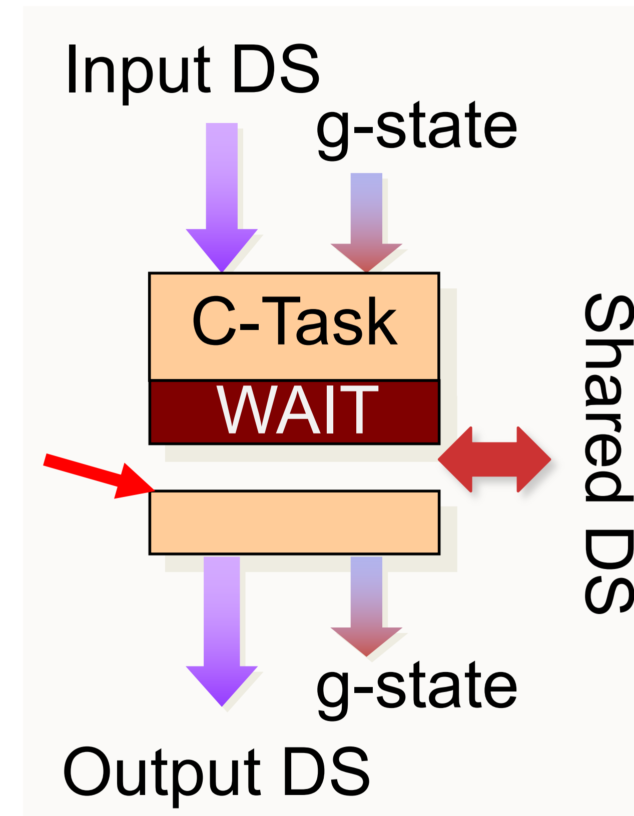
Simple Task (S-Task)

- Can execute from beginning to end without delay, given the CPU has been allocated to it
- No blocking inside (no synchronization, communication)
- Independent progress
- Inputs available in input data structure at the task start
- Outputs ready in the output data structure upon task completion
- API: input DS, output DS, g-state



Complex Task (C-Task)

- May contain one or more WAIT operations
- Possible dependencies due to synchronization, communication
- Progress dependent on other tasks in node or environment
- C-task timing is a global issue
- API: input DS, output DS, g-state, shared DS, dependencies



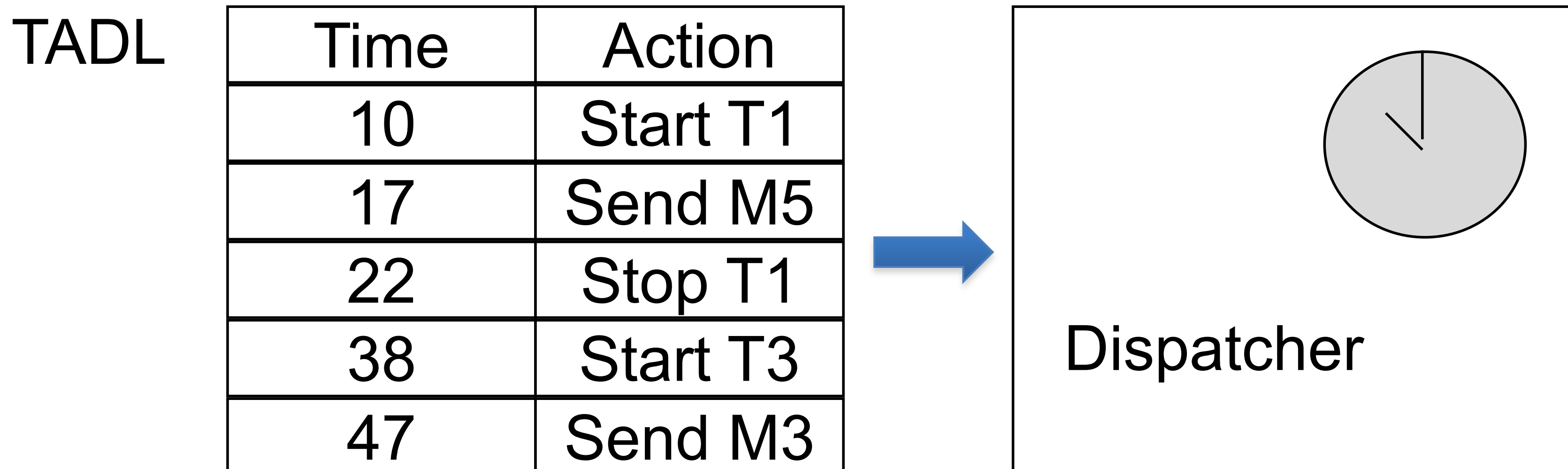
ARINC Standard WG 48-1999

HRT systems demands (taken from ARINC standard):

The Avionics Computing Resource (ACR) shall include internal hardware and software management methods as necessary to ensure that time, space and I/O allocations are deterministic and static.
"Deterministic and static" means in this context, that time, space and I/O allocations are determined at compilation, assembly or link time, remain identical at each and every initialization of a program or process, and are not dynamically altered during runtime.

Time-Triggered Task Control

In strictly TT systems, the dispatcher controls the execution of tasks, by interpreting the Task Descriptor List (TADL).



Time	Action
10	Start T1
17	Send M5
22	Stop T1
38	Start T3
47	Send M3

The TADL tables are generated and checked by a static scheduler, before runtime.

TU WIEN

TT Resource Management

In a TT OS there is hardly any dynamic resource management.

- Static CPU allocation.
- Autonomous memory management. It needs little attention from the operating system.
- Buffer management is minimal. No queues.
- Implicit, pre-planned synchronization fulfills synchronization needs and precedence constraints → S-tasks only
- No explicit synchronization (e.g., mgmt. of semaphore queues).

Operating systems become simple, can be formally analyzed.

Examples: TTOS, OSEK time

TU WIEN

TT Task Structure

Basically the task structure in a TT system is static.
Limited means for data/situation dependent adaptation:

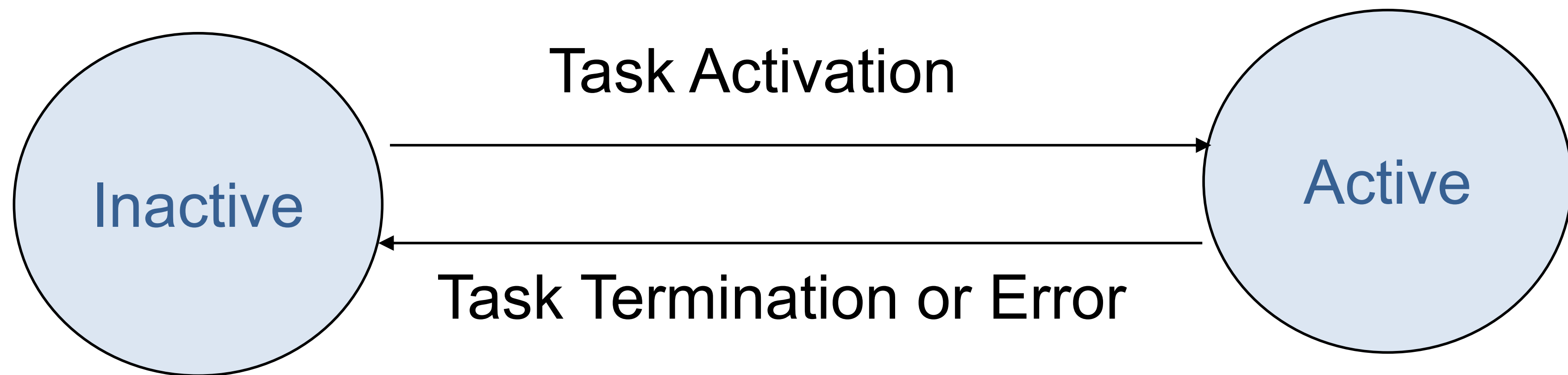
- Mode changes – navigate dynamically between statically validated operating modes.
- Sporadic server tasks: Provide a laxity in the schedule that can be consumed by a sporadic server task.
- Precedence graphs with exclusive or: dynamic selection of one of a number of mutually exclusive alternatives (not very effective!)

Advantages: low runtime complexity, predictability, guarantees

Disadvantages: low degree of flexibility, planning cost, resource reservation based on worst-case assumption

TT Task States

Non preemptive system



TU WIEN

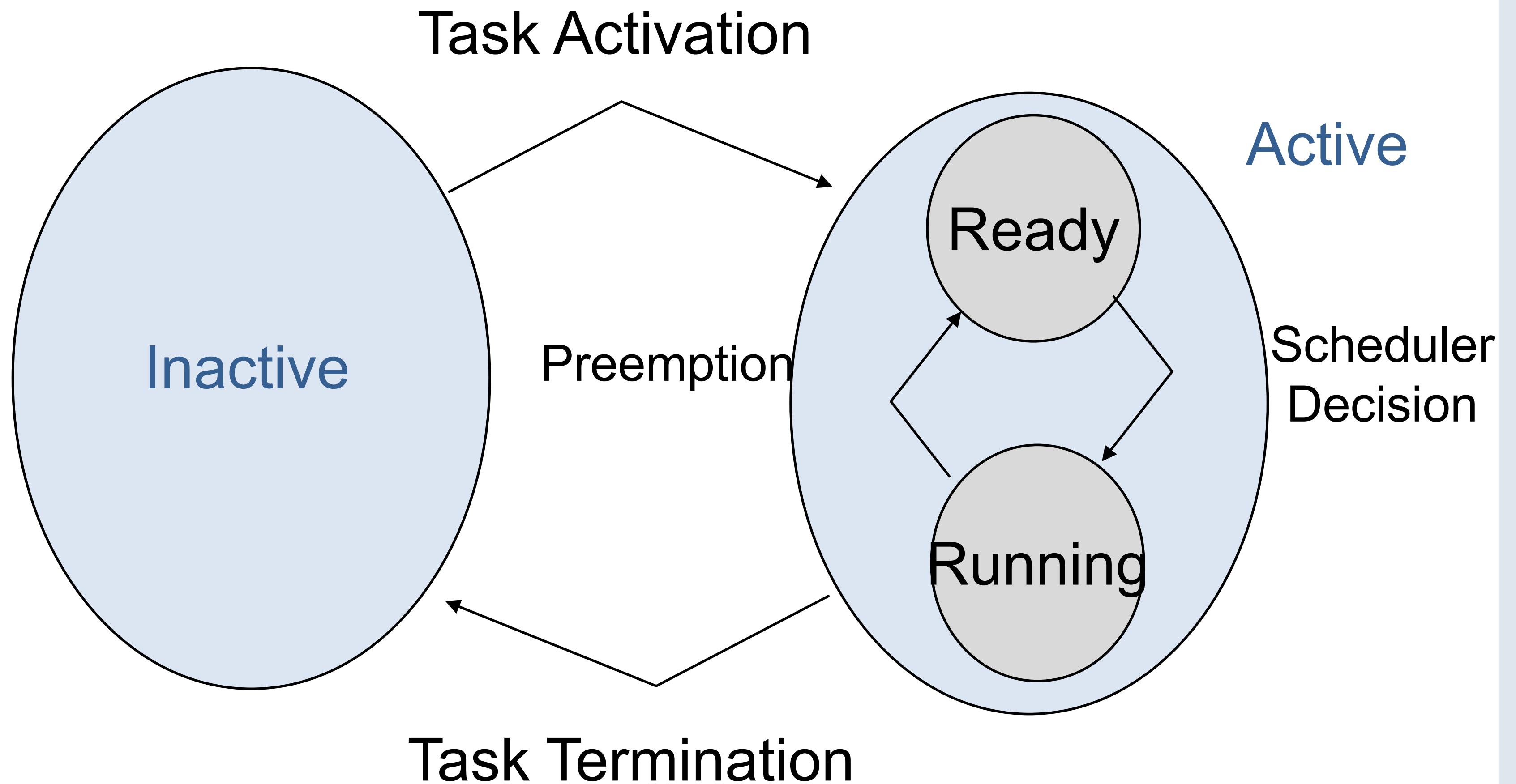
Task Control – ET with S-Tasks

In an ET system, the task control is performed by a dynamic scheduler that decides which task has to be executed next on the basis of the evolving request scenario.

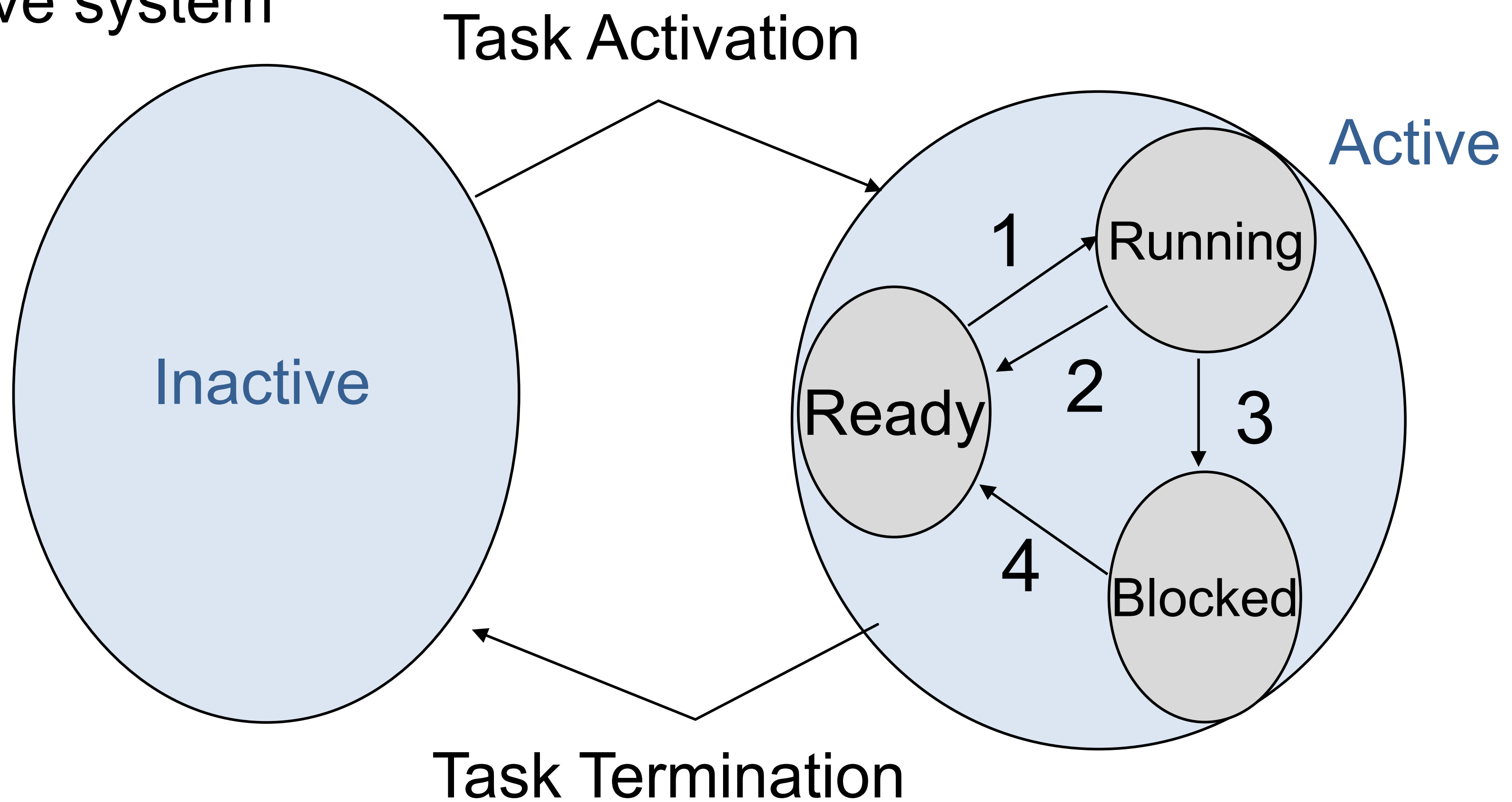
- Advantage:
  Actual (and not maximum) load and task execution times form the basis of the scheduling decisions.
- Disadvantage:
  In most realistic cases the scheduling problem that has to be solved on-line is NP hard.

ET Task States with S-Tasks

Preemptive system



ET Task States with C-Tasks



1 Scheduling Decision
2 Task Preemption
3 Task executes WAIT for Event
4 Blocking Event occurs

TU WIEN

ET Resource Management

In ET OS the dynamic resource management is extensive:

- Dynamic CPU allocation.
- Dynamic memory management.
- Dynamic Buffer allocation and ET management of communication activities
- Explicit synchronization between tasks, including semaphore queue management and deadlock detection.
- Extensive interrupt management.
- Timeout handling of blocked tasks.

A formal timing analysis of ET operating systems is beyond the state of the art (e.g., OSEK).

Task Interaction

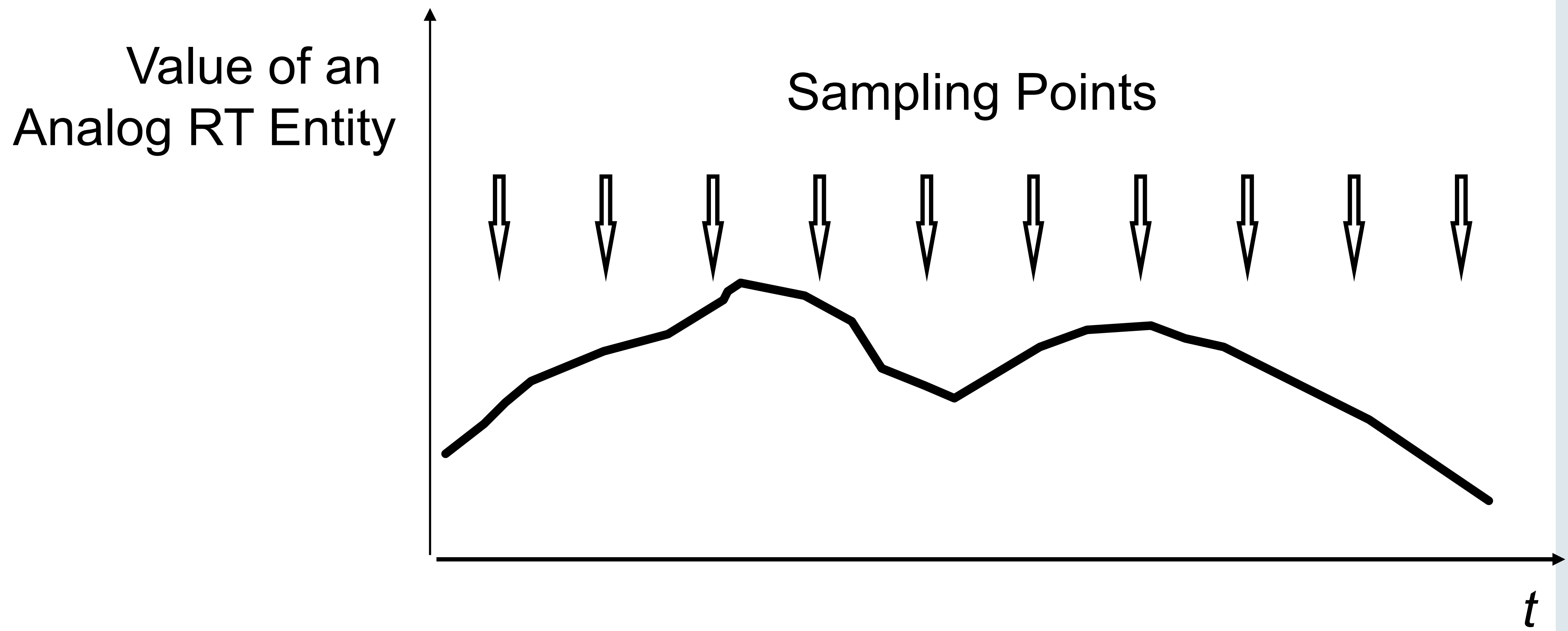
Precedence constraints: restrictions on task sequence (e.g., sequence of actions or outputs)

- TT: reflected in TT schedule (TADL)
- ET: WAIT

Exchange of data

- Messages
- Shared data structure ⇒ provision of integrity
  - TT: Coordinated task schedules
    TT schedule guarantees mutex: deterministic, min. overhead
  - ET: Semaphores

TT I/O: Sampling



TU WIEN

Sampling States vs. Events

Sampling refers to the periodic interrogation of the state of a RT entity by a computer.

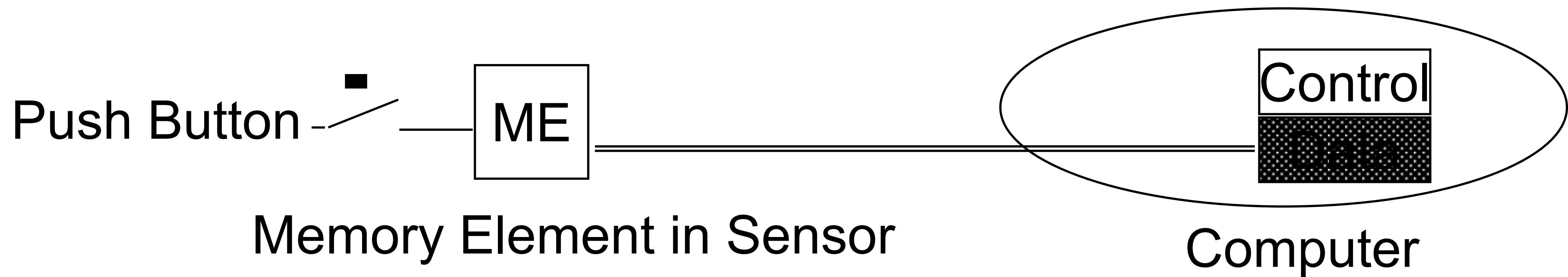
The duration between two sampling points is called the sampling interval.

The length of the sampling interval is determined by the dynamics of the real-time entity.

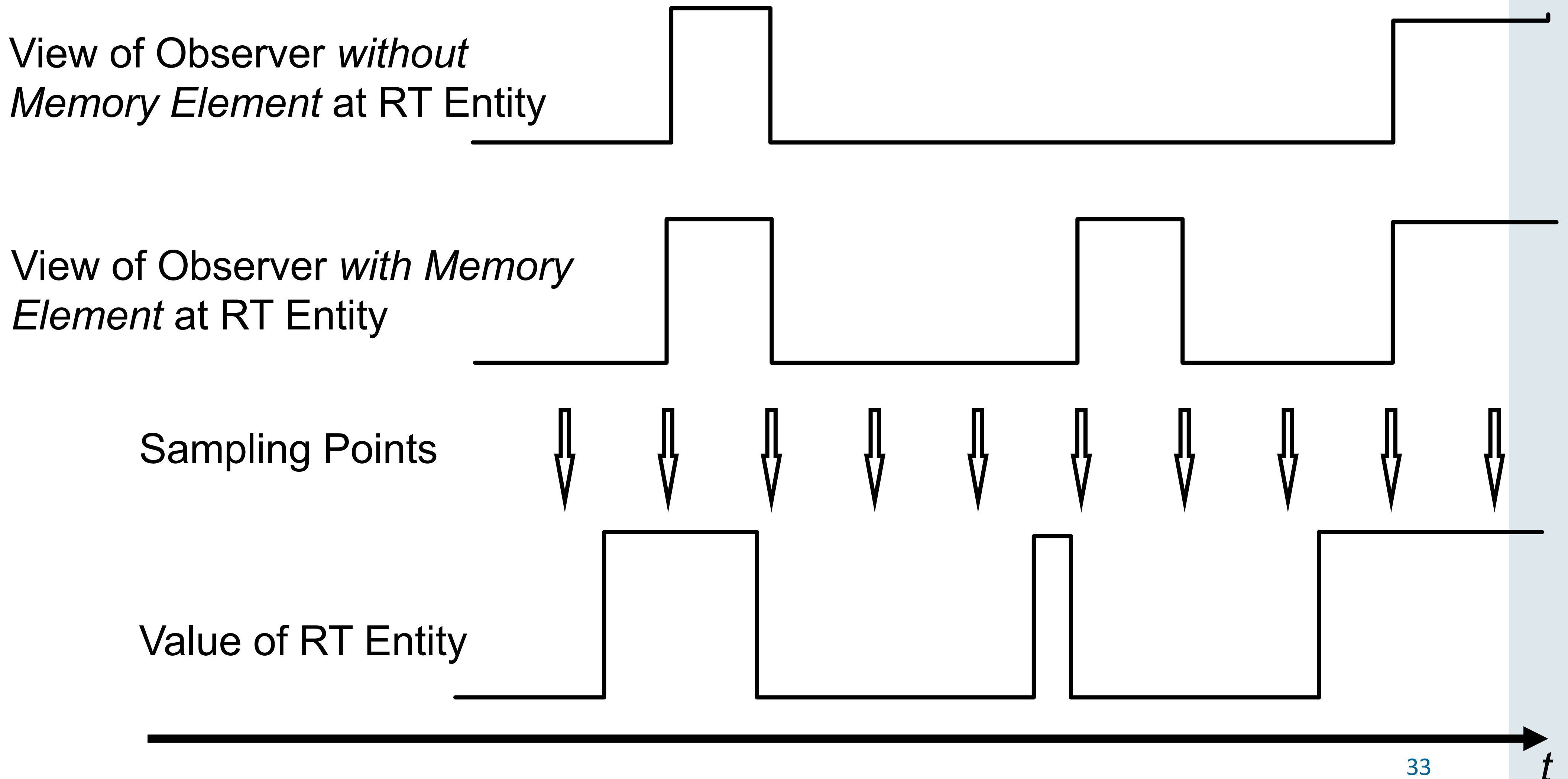
States can be observed by sampling.

Events cannot be sampled. They have to be stored in an intermediate memory element (ME).

Sampling – Position of Memory Element

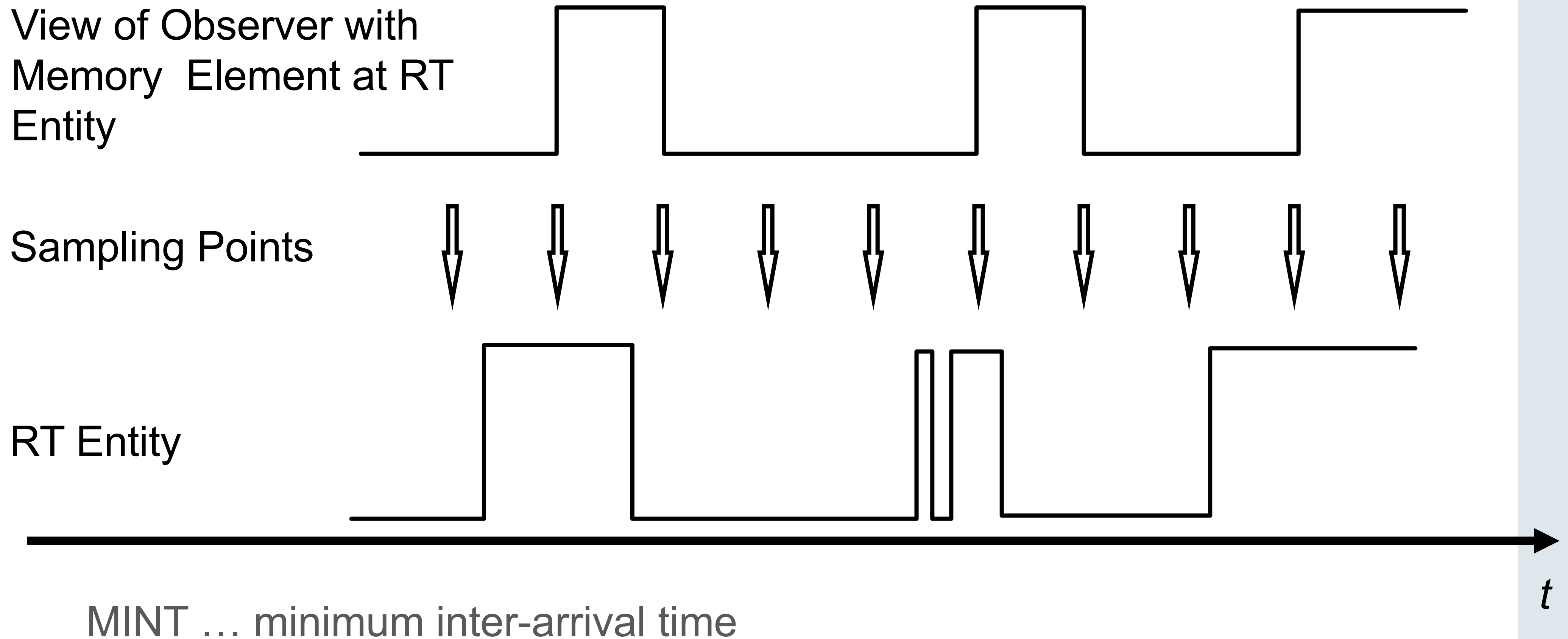


Sampling – Role of the Memory Element



TU WIEN

Sampling – Importance of MINT



MINT … minimum inter-arrival time

TU WIEN

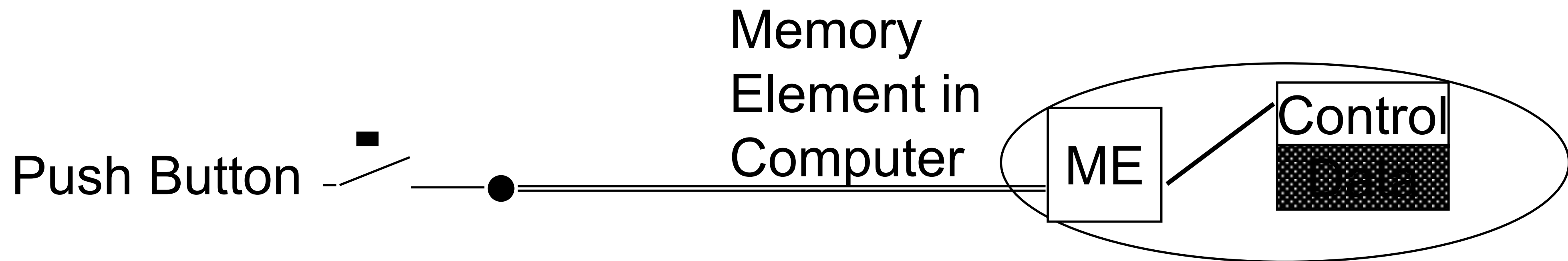
ET: Interrupt

An interrupt is a hardware mechanism that periodically monitors (after the completion of each instruction – or CPU clock cycle) the state of a specified signal line (interrupt line).

If the line is active and the interrupt is not disabled, control is transferred after completion of the currently executing instruction (from the current task) to an instruction (task) associated with the servicing of the specified interrupt.

As soon as an interrupt is recognized, the state of the local "interrupt memory" is reset.

Interrupt



External event forces computer into interrupt service state.

Need for Agreement Protocols

If an RT entity is observed by two (or more) nodes of the distributed system, the following may happen:

- The same event can be time-stamped differently by two nodes – *fundamental limit of time measurement*.
- When reading an analog sensor, a dense quantity is mapped onto a digital values – *discretization error*. Even sensors of highest quality may yield readings differing by a single-bit quantity.

⇨Whenever a dense quantity is mapped onto a discrete representation, agreement protocols are needed to get an agreed view on multiple redundant sensor readings

TU WIEN

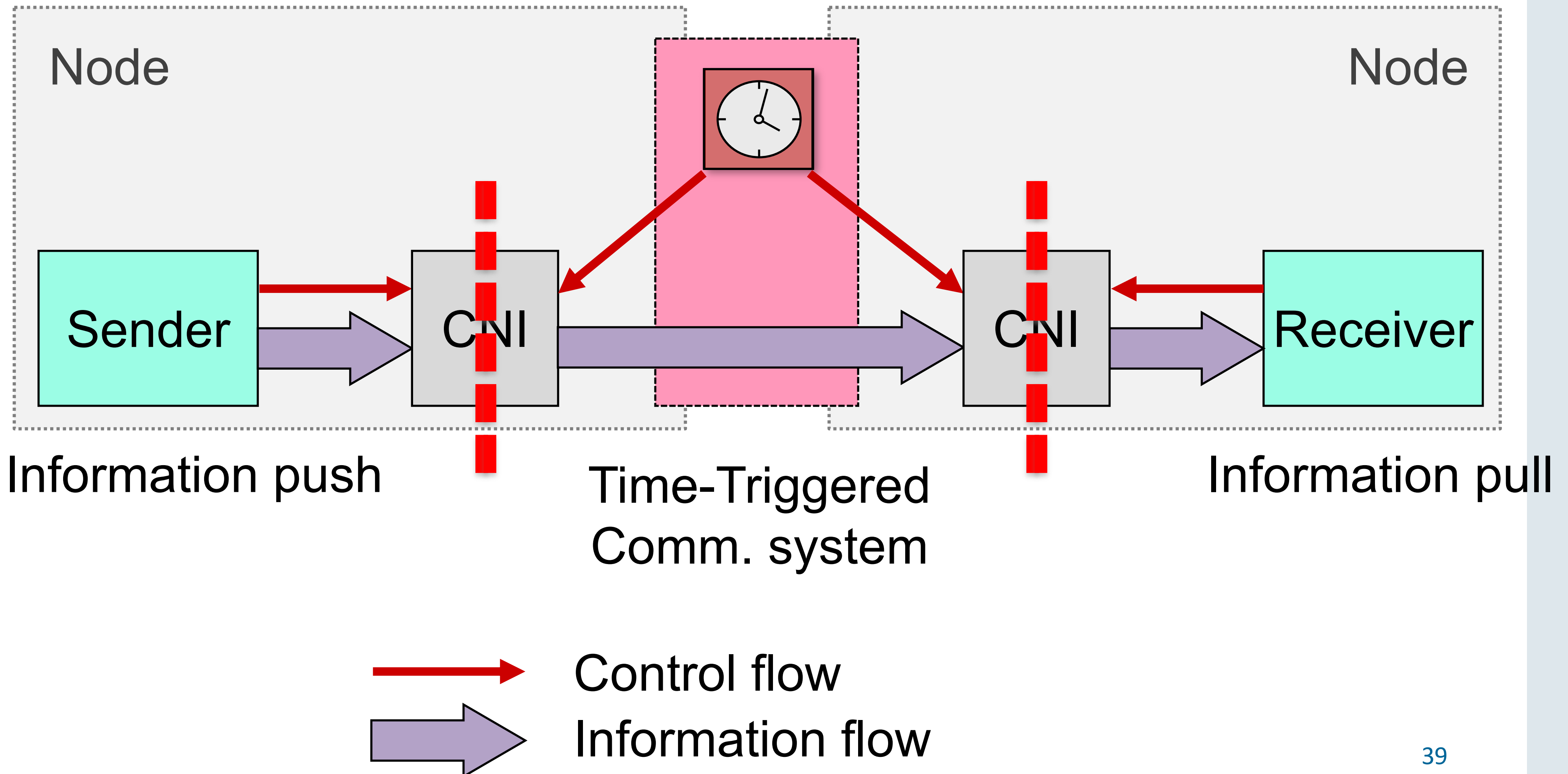
Agreement Protocol

An agreement protocol provides a *consensus* on the *value* of an observation and on the *time* when the observation occurred among a number of fault-free members of an ensemble:

- First phase: exchange of the local observations to get a globally consistent view to each of the partners
- Second phase: each partner executes the same algorithm on the collected data (e.g., averaging) to come to the same conclusion – the agreed value and time

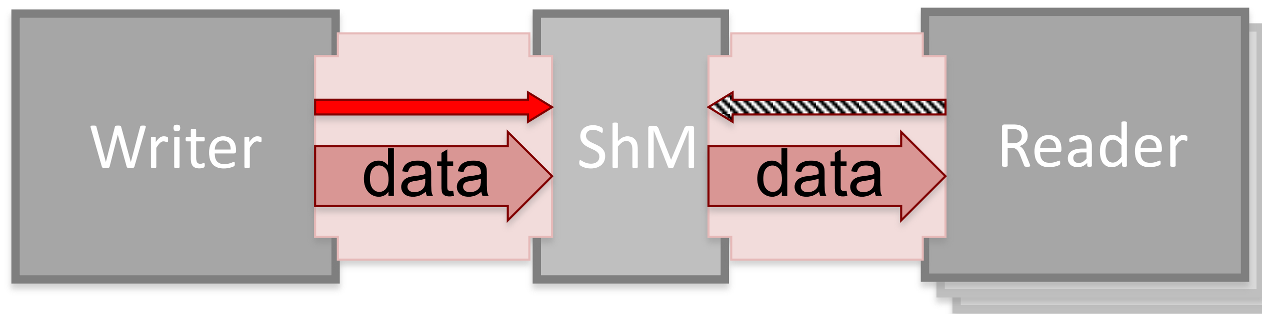
Agreement always needs an extra round of communication and thus weakens the responsiveness of a real-time system.

Component Autonomy and TT Comm.



Mutex at CNI

- One writer, one or more reader(s) with private CPUs
- Communication via shared memory
- Intervals between writes >> duration of writes



Writer: immediate (non-blocking) write

Reader:

- async. access, mask read delay jitter ➪ NBW variants, or
- synchronize node actions to TT network operation

TU WIEN

Non-Blocking Write Protocol (NBW)

Node is fully autonomous; not synchronized to TT network

Demanded Protocol Properties (NBW):

Consistency: Read operations must return consistent results.

Non-Blocking Property: Readers must not block the writer.

Timeliness: The maximum delay of a reader during a read operation must be bounded (or even constant).

Non-Blocking Write Protocol

Init:

```
CCF := 0;        /* concurrency control flag */
```

Writer:

CCF_old := CCF;
CCF := CCF_old + 1;
write to shared struct;
CCF := CCF_old + 2;

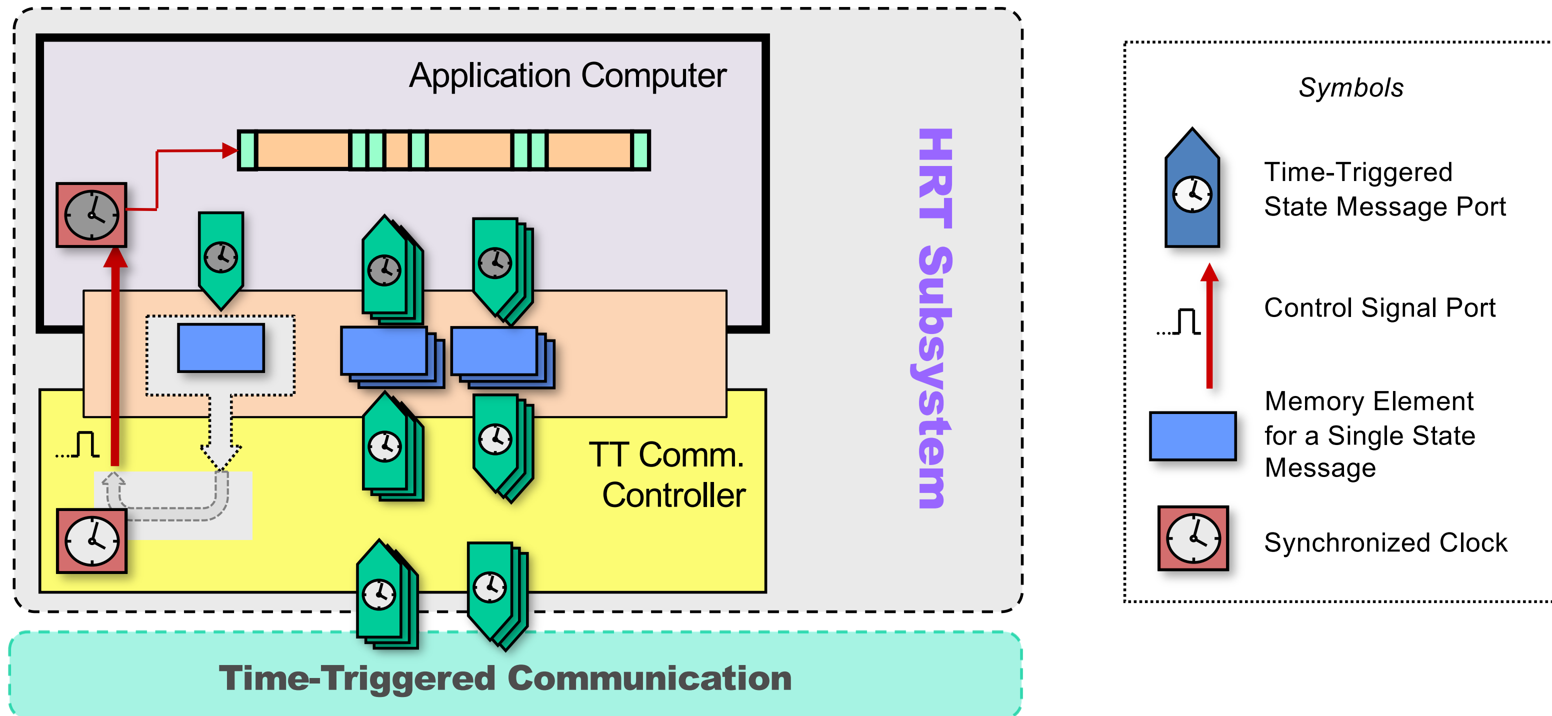
Reader:

start: CCF_begin := CCF;
if CCF_begin **mod** 2 = 1
then goto start;
read from shared struct;
CCF_end := CCF;
if CCF_end ≠ CCF_begin
then goto start;

CCF arithmetics in practice: all CCF operations mod (2 * bigN)

Synchronous TT Component

Node synchronizes CNI access to TT Comm. Controller

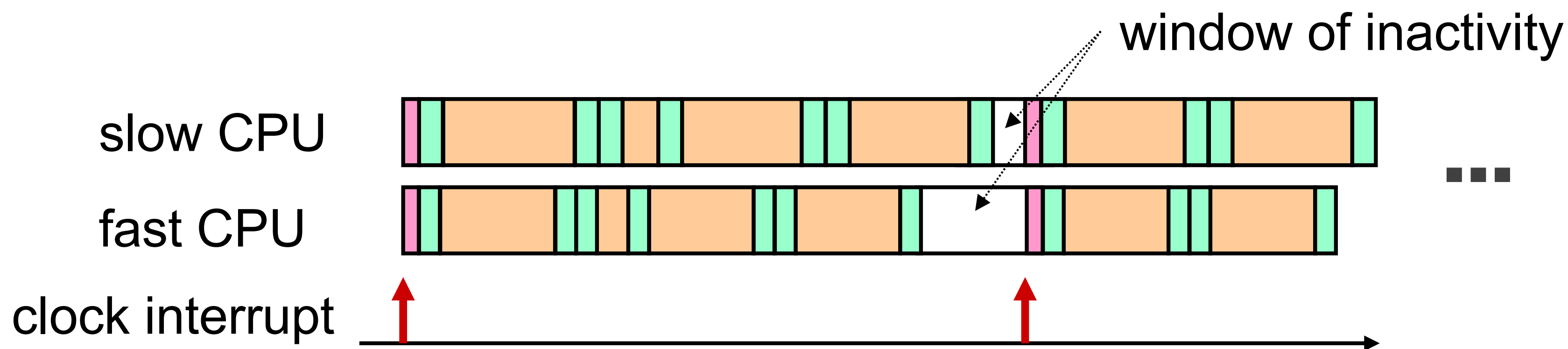


Synchronization with Real-Time Clock

Master clock synchronization

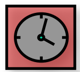
- Programmed clock interrupt from connector unit

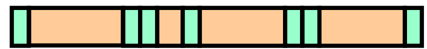
Planned window of inactivity before expected clock sync. time allows slow CPUs to complete the same workload as fast CPUs (needs bound on clock skew)

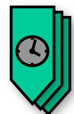


Time-Predictable Component (2)

synchronized representation of global time


instruction counter “clock”,
synchronized to the local representation of global time


Static schedule (instruction-counter interrupt for preemptions)


Data transfer triggered by progression of instruction counter clock

Data transfer triggered by progression of global-time representation

Programmable clock interrupt to synchronize the instruction-counter clock with the global-time representation

Points to Remember

- Separation HRT SW vs. other SW
- Time services and the role of time
- Pre-planned TT operation ➪ simple & verifiable SW
- Sampling I/O
  - Application-dependent timing parameters
- Input agreement at system borders
- Component Autonomy and TT Cluster Communication
  - Autonomous component with conflict masking
  - Synchronous time-predictable TT component