Distributed Systems and Time

1. Distributed Solution
2. Real-Time Environment
3. Global Time
4. Modelling Real-Time Systems
5. Real-Time Entities and Images

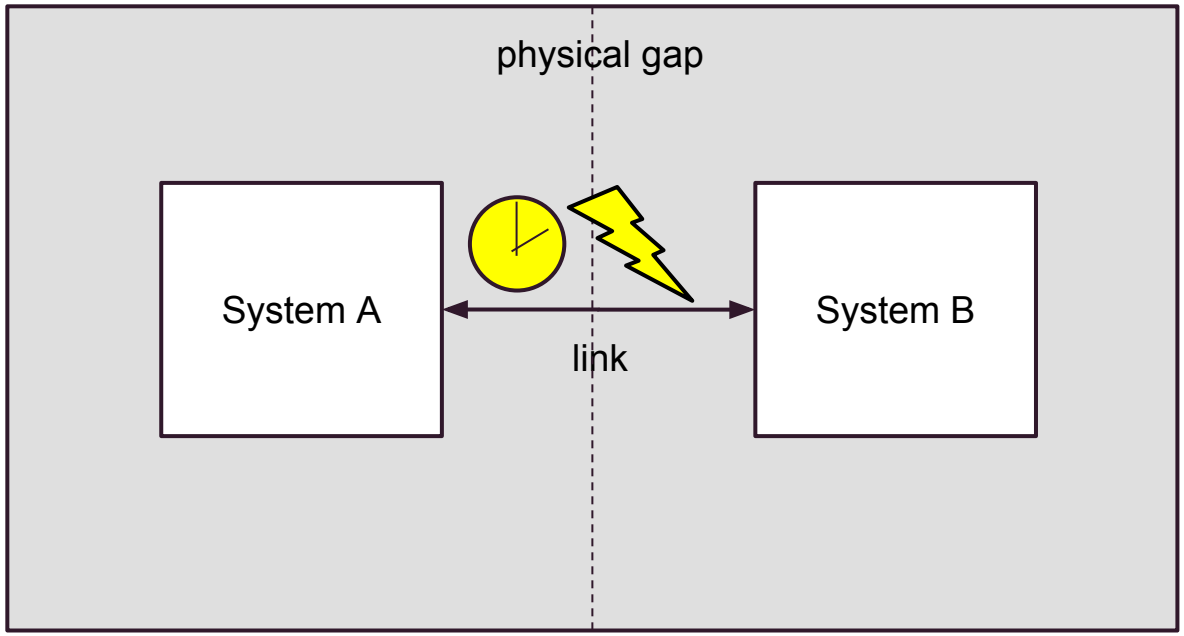
Distributed Solution

1. System Architecture
2. Communication-Network Interface
3. Composability
4. Scalability
5. Dependability

What is a Distributed System?

Two connected systems which are separated by a physical gap such that an explicit communication link is required, can be regarded as a distributed system. It is likely that the communication via the link has

- significant **delay** in relation to communication inside the system
- a higher **failure rate** than communication inside the system



Reasons for Distribution

From a functional point of view, it makes little difference whether a given specification is implemented using a centralized architecture or a distributed architecture.

Arguments to realize hard real-time systems as distributed solutions are

1. composability
2. scalability, and
3. dependability.

System Architecture

In a distributed system, it is feasible to encapsulate a logical function and the associated computer hardware into a single unit, a node.

A **node** can be replaced by an **understandable abstraction** that captures the essential functions and temporal properties of the node and hides the irrelevant details of the implementation behind a simple and stable external node interface.

It is a major advantage of a distributed solution that the extracted abstractions hold also in the case of failures. In a centralized computer system, the problems of fault diagnosis and the analysis of the effects of a subsystem failure can be complex.

Composability

Large systems are often built by integrating a set of well- specified and tested subsystems.

The challenge is to link the subsystems such that their established properties at the subsystem level do hold on the system level.

An architecture is said to be composable with respect to a specified property if the system integration will not invalidate this property once the property has been established at the subsystem level. Examples of such properties are **timeliness** or **testability**. In a composable architecture, the system properties follow from the subsystem properties.

The **communication system** has a central role in determining the composability of a distributed system.

Scalability

A scalable architecture is open to changes, and does not limit the extensibility of a system by some upper limit.

Distributed Systems are usually more scalable than central systems.

A scalable architecture must not have any central bottleneck, neither in processing nor in communication capacity. The advantages of distributed architectures:

- Nodes can be added within the given capacity of the communication channel to introduce additional processing power to the system
- If the communication capacity within a cluster is fully utilized, a node can be transformed into a gateway node to open a way to a new cluster.

Complexity

Large systems can only be built if the effort required to understand the system operation, remains under control as the system grows.

The complexity of a large system can be reduced, if the inner behavior of the subsystem can be encapsulated behind stable and simple interfaces. Only those aspects that are relevant to the function under consideration must be examined to understand the particular function.

The partitioning of a system into subsystems, the encapsulation of the subsystem, the preservation of the abstractions in case of faults, and most importantly, a strict control over the interaction patterns among the subsystems, are thus the key mechanisms for controlling the complexity of a large system.

In a TT architecture, the structure encapsulates the inner operation of one cluster from that of other clusters via data- sharing gateway interfaces.

The complexity of reasoning about the correctness of any system function is only determined by the cluster under investigation, and not by the other clusters.

Gateways

The purpose of a gateway is to exchange information between two interacting clusters.

In most cases, only a small subset of the information of one cluster is relevant to the other cluster. The gateway host must transform the data formats of one cluster to those expected by the other cluster.

A gateway encapsulates and hides the internal features of a cluster (e.g., a legacy system) and provides a clean and flexible interface.

Cost

A centralized system might provide short-term cost advantage, but can lead to a long-term cost disadvantage, as system size grows (scalability).

Dependability

A fault-tolerant system must be structured into partitions that act as **error-containment regions** in such a way that the consequences of faults that occur in one of these partitions can be detected and corrected or masked before these consequences corrupt the rest of the system.

Error containment can usually be easier established in distributed systems than in central systems.

The error-containment coverage is the probability that an error that occurs within an error-containment region is detected at one of the interfaces of this region.

Replication

For fault-tolerance, units of a system need to be redundant. While this is complex in centralized systems, a distributed system offers a clear solution.

In a distributed system, a **node** can represent a **unit of failure** (error containment region), preferably with a simple failure mode (e.g., fail-silence). All inner failure modes of a node are mapped into a single external failure mode.

If this failure hypothesis holds, node failures can be masked by providing actively replicated (redundant) nodes. However, the issue of replica determinism, i.e., replicated nodes visit the same states in about the same time, requires careful hardware / software design.

Certification Support

Frequently the design of a safety critical system must be approved by an independent certification agency. The agency bases its assessment on the analysis of the **safety case** presented by the designer.

A safety case is the accumulation of credible analytic and experimental evidence that convinces the agency that the system is fit for its purpose, i.e., it is safe to deploy.

A system with less and well defined interations via interfaces is easier to certify than a centralized system with many and complex interactions.

Critical vs. Non-critical Functions

In usual commercial systems it is required to operate critical and non-critical system functions in one system.

It is of utmost importance to present an architecture that rules out by design any unintended interactions among subsystems of differing criticality.

A distributed architecture provides a clear strategy to separate non-critical functions from critical functions by establishing error containment and by using composability arguments.

Real-Time Environment

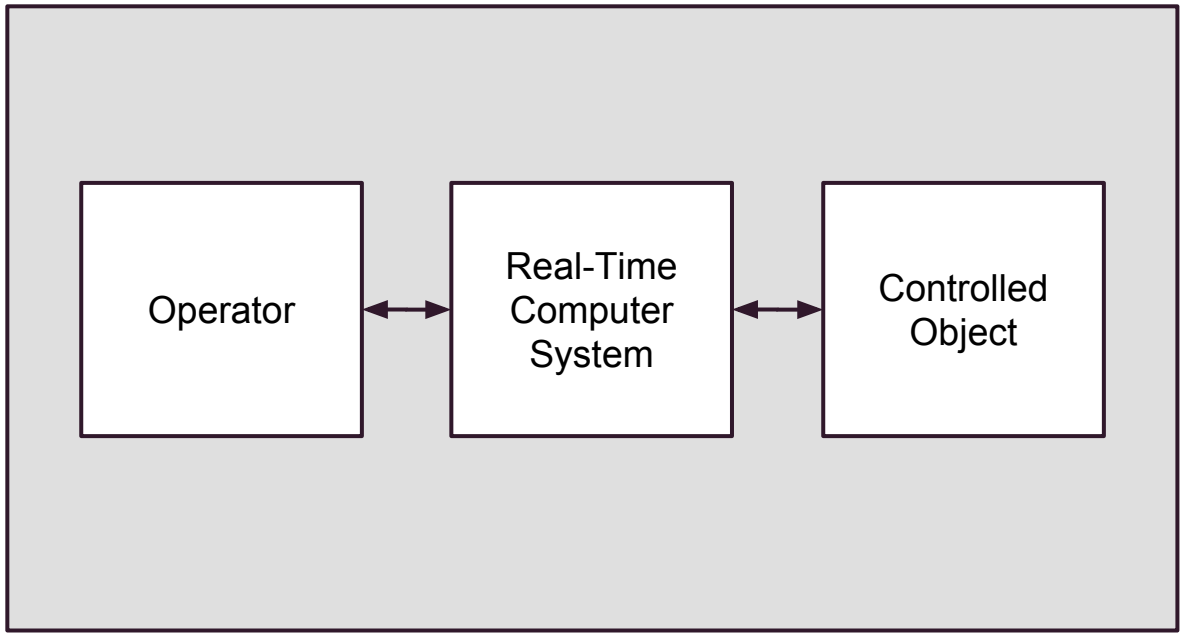


Figure: Real-Time System (Source: [1] p.2)

Real-Time Computer System

“In a real-time computer system the correctness of the system behavior depends not only on the correct outputs and inputs of the real-time computer in the data domain, but also on the points in time at which these values are produced and consumed.”

Classification

Deadline – an RT computer system must react to a stimuli from the controlled object within time intervals dictated by its environment. If a result has utility even after the deadline has passed — **soft deadline**. Otherwise — **firm deadline.** If a catastrophe could result, after a firm deadline is missed — **hard deadline.**

Hard real-time systems must sustain a guaranteed temporal behavior (hard deadline) under all specified load and fault conditions

Soft real-time systems tolerate to miss a deadline occasionally

Real-Time Entity and Image

A controlled object (e.g., a car) changes its state as a function of time — state variables are e.g., speed, … A significant state variable is called a real-time entity (**RT entity**).

Every RT entity is in sphere of control of a subsystem. Only the subsystem has the authority to change it. Outside of the subsystem, the value of the entity can be observed, but not changed.

An observation of an RT entity is represented by a real-time image (**RT image**) in the computer system. A given RT image is only temporally accurate for a limited time interval. If the state of a controlled object changes very quickly, the corresponding RT image has a very short accuracy interval.

Acquiring Real-Time Images

Time-Triggered (TT) observation. The updates of RT images from RT entities are performed periodically, triggered by the progression of a real-time clock with a fixed period.

Event-Triggered (ET) observations. The updates are performed immediately after a change of the state in the RT Entity occurs

A physical sensor produces a raw data element. The raw data element needs to be transformed before it can be used as RT Image, e.g.

- Averaging algorithm to reduce measurement errors
- Calibration
- Transformation to the standard measurement units
- Judged for plausibility

Classification of RT Systems

1. hard RT vs. soft RT
2. fail-safe vs. fail-operational
3. guaranteed response vs. best effort
4. resource adequate vs. resource inadequate
5. event-triggered vs. time-triggered

1, 2: depends on application

3-5: depends on design

Hard-RT vs Soft-RT

Characteristic	Hard RT	Soft RT
Response Time	must be guaranteed	best effort
Peak-load performance	predictable	degraded
Control of pace	controlled object	computer
Size of data	small	large
Redundancy	active	standby, checkpoint recovery
Data integrity	short-term	long-term
Error detection	autonomous	non-autonomous

Table: Hard RT vs Soft RT (Source: [1] p.12)

Failures

If a safe state can be identified and quickly reached upon the occurrence of a failure, then the system is called **fail-safe**.

In applications, where a safe state cannot be identified, minimal level of service to avoid a catastrophe even in a case of failure must be provided. These applications are called **fail-operational**.

Response

A guaranteed-response system is designed based on well- specified fault- and load-hypothesis. It works even in the **case of peak load and fault scenarios**.

If an analytic response guarantee cannot be given, it is called a **best-effort** design. The design is based on the principle “best possible effort taken”. Most non safety-critical real-time systems are designed according to the best-effort paradigm.

Resources

Guaranteed response systems have sufficient pre-allocated computing resources available to handle the specified peak load and the fault scenarios.

For many non safety-critical real-time system designs a dynamic resource allocation strategy based on resource sharing and probabilistic arguments about the expected fault and load scenarios is acceptable.

Computing resources are e.g.

- random access memory (heap and stack)
- CPU processing capacity
- communication bandwidth
- I/O bandwidth

Trigger of Action

Any occurrence that happens at a cut of the time line is called an **event**. A **trigger** is an event that causes the start of some action, e.g., the execution of a task. In the **event-triggered** (ET) approach, all communication and processing activities are initiated whenever a significant change of state is observed. In the **time-triggered** (TT) approach, all activities are initiated at predetermined points in time.

Event-Triggered Systems

- signaling is realized by an interrupt mechanism
- dynamic scheduling to activate appropriate tasks

Time-Triggered Systems

- all activities are initiated by the progression of time
- only one interrupt period (the periodic clock int.)
- clocks of all nodes are synchronized

Zeichnung action

Global Time

1. Time and Order
2. Temporal and Causal Order / Delivery Order Clocks
3. Global Time
4. Time Base
5. Clock Synchronization

Time and Order

In a typical real-time application, different functions are normally executed at different nodes.

To guarantee the consistent behavior of the entire distributed system, it must be ensured that all nodes process all events in the same consistent order in which the events occurred.

A global time base helps to establish such a consistent temporal order on the basis of the timestamps of the events.

Zeichnung
3 state
machine

Temporal Order

The continuum of the real time can be modeled by a direct timeline consisting of an infinite set {T} of instants. We call the order of instances on the timeline the **temporal order**.

A section of the timeline is called a **duration**. An **event** takes place at an instant of time and does not have a duration. If two events occur at an identical instant, then the two events occur simultaneously.

Zeichnung
event
duration

Causal Order

In many real-time applications, the causal dependencies (cause and effect) among events are of interest.

A change of an RT entity might cause many others to react and generate alarms. Such a set of alarms is called an **alarm shower**. The event that triggers the alarm shower is called the **primary event**.

Temporal order of two events is necessary, but not sufficient, for their causal order. Causal order is more than temporal order.

Delivery Order

A weaker order relation is a consistent **delivery order**.

E.g. a communication system may guarantee that all host computers in the nodes see the sequence of events in the same delivery order.

This delivery order is not necessarily related to the temporal order of event occurrences or the causal relationship between events.

Physical Clock

A physical clock is a device for measuring time. It contains a counter and a physical oscillation mechanism that periodically generates an event to increase the counter.

The periodic event is called the **microtick** of the clock.

The duration between two consecutive microticks is the **granularity** of the clock (digitization error in time measurements).

Zeichnung granularity

Reference Clock

Assume an omniscient external observer who can observe all events that are of interest in a given context. (Note that this may violate the laws of relativity)

This observer possesses a **unique reference clock z** with frequency f which is in perfect agreement with the international standard time.

Because z is the single reference clock in the system, each event can be stamped with an **absolute time stamp**.

Clock Drift

The drift of a physical clock k between two microticks i and i+1 is the frequency ratio between this clock and the reference clock.

Drift rate $\rho_i^k = | (z(microtick_{i+1}^k) - z(microtick_i^k))/n^k - 1|$

A perfect clock will have a drift rate of 0. Typical maximum drift rates are in the range of 10^{-2} to 10^{-7} sec/sec depending on the quality of the resonator.

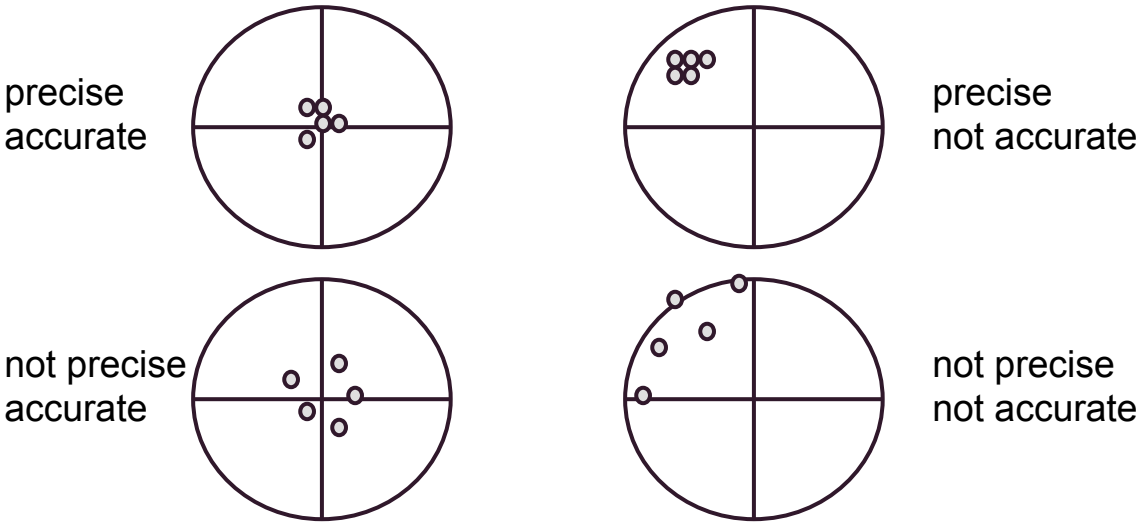
Zeichnung drift

Offset Precision Accuracy

An **offset** denotes the time difference between the respective microticks of the two clocks, measured in the number of microticks of the reference clock.

The **precision** denotes the maximum offset of respective microticks of any two clocks of the ensemble during the period of interest.

The **accuracy** denotes the maximum offset of a given clock from the external time reference during the time interval of interest.



Synchronization

Because of the drift rate of any physical clock, the clocks of an ensemble will drift apart if they are not synchronized periodically. The process of resynchronization of an ensemble to maintain a bounded precision is called **internal synchronization**.

To keep a clock within a bounded interval of the reference clock, it must be periodically resynchronized with the external reference clock. This process of re-synchronization is called **external synchronization**.

Time Standard

Two time bases are relevant for designer of a distributed real- time system, the

- International Atomic Time (TAI) and the
- Universal Time Coordinated (UTC).

TAI offers a second that has been derived from the period of the cesium atom 133. TAI is a **chronoscopic** timescale without any discontinuities.

UTC is a time standard that has been derived from astronomical observations of the rotation of the earth relative to the sun. UTC is not chronoscopic (e.g., leap seconds).

Global Time

If the real-time clocks of all nodes of a distributed system were perfectly synchronized with the reference clock, and all events were time stamped with this reference time, then it would be easy to measure the interval between two events or reconstruct the temporal order.

However, since each node has a local oscillator, such a tight synchronization of clocks is not possible. Therefore, the concept of **global time** is introduced.

A global time is an abstract notion that is approximated by properly selected microticks (e.g. each n-th microtick) from the synchronized local physical clocks of an ensemble.

Such a selected local microtick is called a **macrotick** (or a tick). It is assumed that all clocks are internally synchronized with a given precision π.

Reasonableness Condition

A global time is called reasonable, if all local implementations of the global time satisfies for the global granularity g the condition $g > \pi$.

This condition ensures that the offset between two local clocks counted in microticks is bound to less than the duration of the macrotick (of the global time) counted in microticks.

The macrotick timestamps of any event captured by clocks on different nodes vary by a maximum of 1.

If the difference between the timestamps of two events is equal or larger than 2 macroticks, that temporal order of events can be recovered.

Zeichnung

Limits of Time Measurement

1) If a single event is observed by two different nodes, there is always a possibility that the timestamps differ by one tick – and a one tick difference is not sufficient to reestablish the temporal order of the events.

2) If the observed duration of an interval is d_{obs}, then the true duration d_{true} is bound by

$$(d_{obs} - 2\ g) < (d_{true}) < (d_{obs} + 2\ g)$$

since a duration is bound by two events. For each event 1) is valid.

3) The temporal order of events can be recovered from their timestamps, if the difference is ≥ 2 ticks.

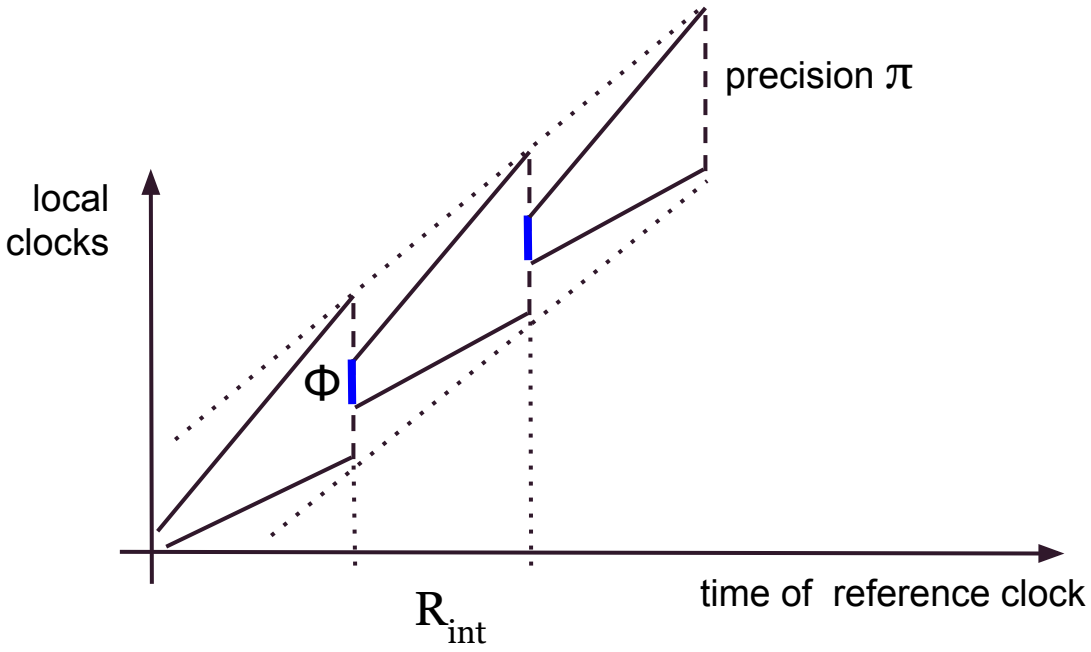
Internal Clock Synchronization

The purpose of internal clock synchronization is to ensure that the global ticks of all correct nodes occur within the specified precision π, despite the varying drift rate of the local real-time clock of each node.

Because the availability of a proper global time base is crucial for the operation of a distributed real-time system, the clock synchronization should be fault-tolerant.

Synchronization Condition

The period of resynchronization is called the resynchronization interval R_{int}. At each resynchronization point clocks are corrected.



Synchronization Condition

For a set of clocks with a deviation rate of ρ, which are synchronized each R_{int}, the maximum deviation between the clocks is limited by a drift offset Γ of

$$\Gamma = 2 \rho R_{int}$$

If Φ is the deviation between the clock immediately after the synchronization and the precision π the upper limit of the allowed deviation between two clocks, the the following condition must hold:

$$\Phi + \Gamma < \pi$$

Central Synchronization

A simple central synchronization mechanism could work like this [2]:

1. A master node periodically transmits its timestamps.
2. Each receiver node calculates the deviation between the local time and the timestamp in the message minus the transmission time as a correction value
3. Each receiver applies the correction value to the local clock

The precision of the algorithm is limited by the variation (jitter ε) of the communication time.

$$\pi_{central} >= \varepsilon + \Gamma$$

Disadvantage is the missing fault-tolerance against failure of the central clock and drift of the central clock.

Byzantine Error

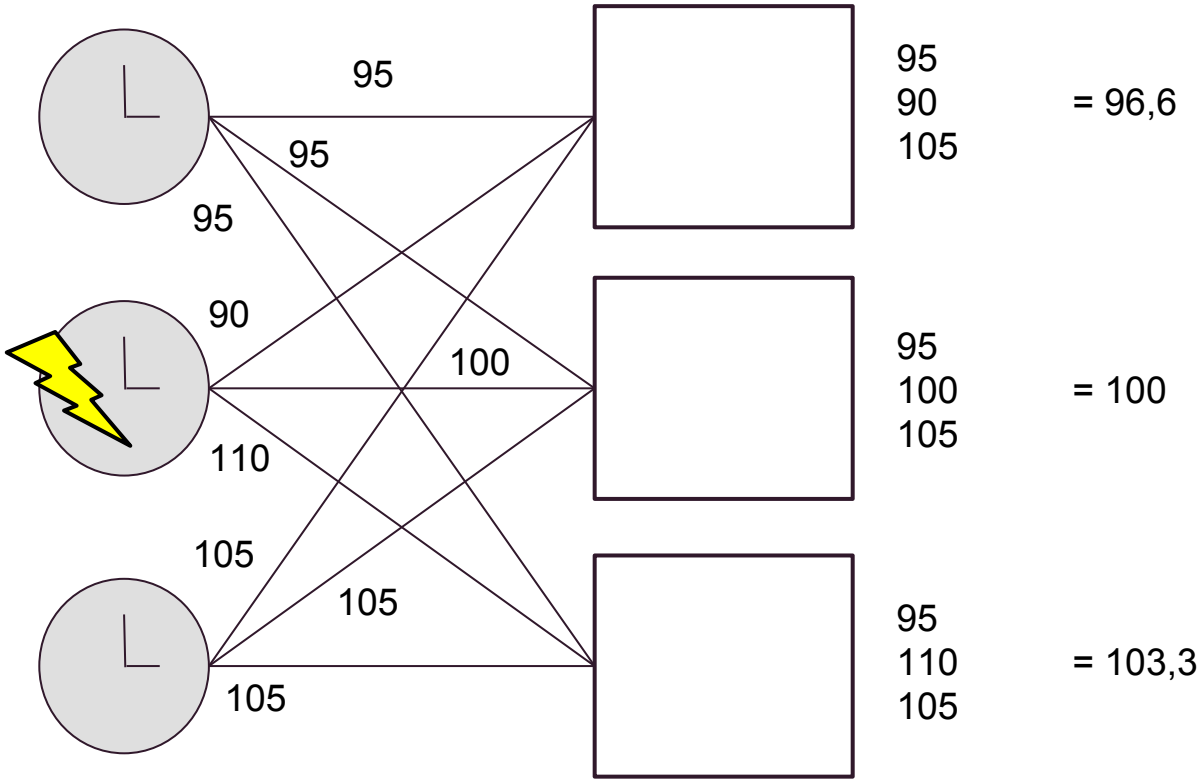
In an ensemble of three nodes, one malicious node can prevent the other two nodes from synchronizing their clocks.

Such a malicious, “two-faced” manifestation of behavior is sometimes called a malicious error or Byzantine error. During the exchange of the synchronization messages, a Byzantine error can lead to inconsistent views of the state of the clocks.

In that case clock synchronization can only be guaranteed if the number of clocks N is equal or larger to 3k+1, where k denotes the number of Byzantine faulty clocks.

$$N \geq (3k+1)$$

Byzantine Error



Latency Jitter

The most important term affecting the precision of the synchronization in a distributed system is the latency jitter of the synchronization messages that carry the current time values from one node to all other nodes of the ensemble.

Example ranges of the jitter:

- Application software level: 500µs to 5ms
- Kernel of operation system: 10µs to 100µs
- Hardware of communication controller: < 10µs

FTA Algorithm

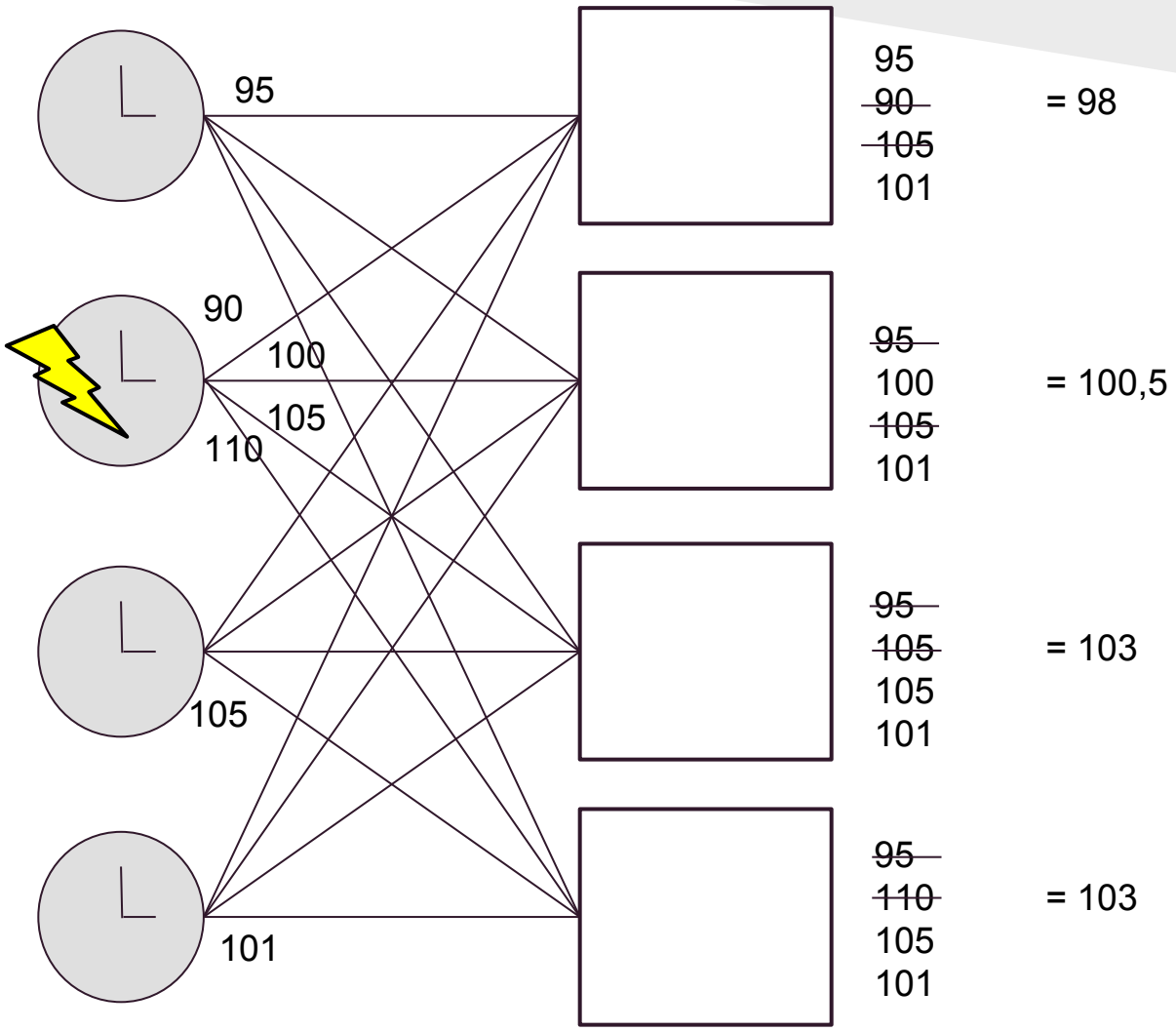
The Fault-Tolerant-Average (FTA) algorithm is an example for a distributed synchronization function.

In a system with N nodes k Byzantine faults should be tolerated. At every node, the N measured time differences between the node's clock and the clocks of all other nodes are collected.

These time differences are sorted by size, the k largest and the k smallest time differences are removed.

The remaining N-2k time differences are, by definition, within the precision window. The average of these remaining time differences is the correction term for the node's clock.

FTA Algorithm k=1



Clock Correction

The correction term calculated by the synchronization function can be applied to the local-time value immediately (**state correction**), or the rate of the clock can be modified so that the clock speeds up or slows down during the next resynchronization interval to bring the clock into a better agreement with the rest of the ensemble (**rate correction**).

State correction is simple to apply, but it is generating a discontinuity in the time base. Rate corrections have to be designed carefully, since they may lead to oscillations of the clock assemble.

External Clock Synchronization

External synchronization links the global time of a cluster to an external standard of time (e.g. GPS time, DCF time).

For this purpose it is necessary to access a time server, i.e., an external time source that periodically broadcasts the current reference time in the form of a time message.

This time message must raise a synchronization event in a designated node of the cluster.

Model of a Real-Time System

1. The Purpose of the Model
2. Load / Fault Hypothesis
3. Structure
4. Tasks
5. Timing
6. H-state, ground-state

The Purpose

The limited information processing capacity of the human mind requires a goal oriented information reduction strategy to develop a reduced representation of the world (a **model**) that help in understanding the problem.

Reality can be represented by a variety of models:

- a physical-scale model of a building,
- a simulation model of a process,
- a mathematical model of a phenomena,
- etc.

A model does never include all aspects of the modelled system!

Assumption Coverage

There is the danger of oversimplification, or of omitting a relevant property.

Information reduction, or abstraction, is only possible if the goal of the model-building process has been well defined. Otherwise it is hopeless to distinguish between the relevant and the irrelevant information.

The probability that the assumptions made in the model building process hold in reality is called **assumption coverage**. It limits the probability that conclusions derived from a model are valid in the real world.

Load / Fault Assumptions

Two important assumptions must be made while designing a model of a fault-tolerant real-time computer system.

Statements about the response time of a computer system can only be made under the assumption that the load offered to the computer system is below a maximum load, called the peak load (**load hypothesis**).

e.g. maximum requests per seconds

Assumptions about faults that can occur during the operation of a system have to be made (**fault hypothesis**). If the faults that actually occur in the real world are not covered by the fault hypothesis, then, even a perfectly designed fault-tolerant computer system will fail.

e.g. maximum one component fails per mission time

Structural Elements

A distributed fault-tolerant real-time application can be decomposed into a set of communicating **clusters**. A **computational cluster** can be further partitioned into a set of **fault-tolerant units** (FTUs) connected by a **real-time network**.

Each FTU consists of one or more node computers. Within a **node** computer, a set of concurrently executing **tasks** performs the intended functions.

A node is a self-contained computer with its own hardware (processor, memory, communication interface, interface to the controlled object) and software (application program, operating system), which performs a set of well-defined functions within the distributed computer system. A node is the most important abstraction in a distributed real-time system because it binds software and hardware resources into a single operational unit with observable behavior in both temporal and value domain.

Structure and State of a Node

A node hardware consists of a host computer, a communication network interface (CNI), the IO interfaces and a communication controller.

The node software can be divided into two data structures:

1. The initialization state (**i-state**) is a static data structure that contains the static program code and the initialization data.
2. The history state (**h-state**) is the dynamic data structure of the node.

The i-state is stored in ROM and the h-state is stored in RAM. After the start of a node the h-state is empty and has to be initialized either from ROM or from other redundant nodes.

Fault-Tolerant Unit

In many cases, a node of a distributed computer system is the smallest replaceable unit (**SRU**) that can be replaced in case of a fault.

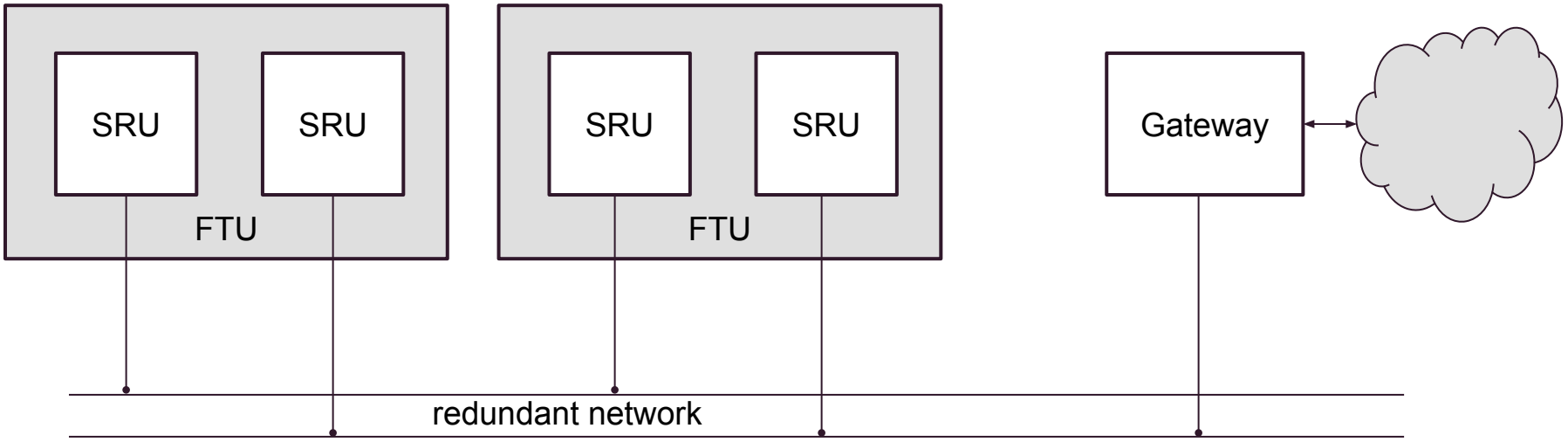
A fault-tolerant unit (**FTU**) is an abstraction that is introduced for implementing fault tolerance by active replication. An FTU consists of a set of replicated nodes that are intended to produce replica determinate result messages (same results at approximately the same points in time).

Computational Cluster

A fault-tolerant computational cluster consists of a set of FTUs that cooperate to perform the intended fault-tolerant service for the cluster environment. It consists of the **controlled object**, the **operator**, and other computational clusters.

The interfaces between a cluster and its environment are formed by the **gateway nodes** of the cluster.

Structural Elements



Interfaces

The most important activity in the design is the layout and the placement of the interfaces, since architecture design is primarily interface design.

An **interface** is a common boundary between two subsystems. It provides understandable abstractions to the interfacing partners. These abstractions capture the essential properties of the interfacing subsystems and hide the irrelevant details. An interface between two subsystems of a real-time system can be characterized by

1. Control properties, the properties of the control signals crossing the interface, i.e. interrupts, chip-select, ...
2. Temporal properties, i.e., the temporal constraints that must be satisfied by the control signals
3. Functional intend, i.e., the specification of the intended functions of the interfacing partners
4. Data properties, i.e., structure / semantics of data, bit ordering, ...

Actions

All activities in a real-time system can be seen as **actions**. Actions take place at every level of the system.

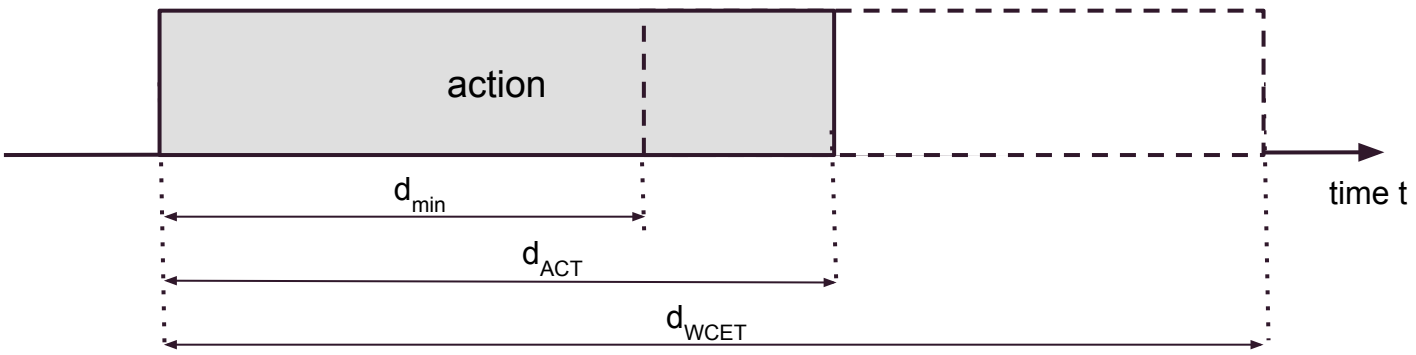
A model of a real-time system typically contains the following actions:

- tasks (execution of code in a computer)
- communication (transfer of information between computers)
- IO (transfer of information between environment and computer)

Duration of Actions

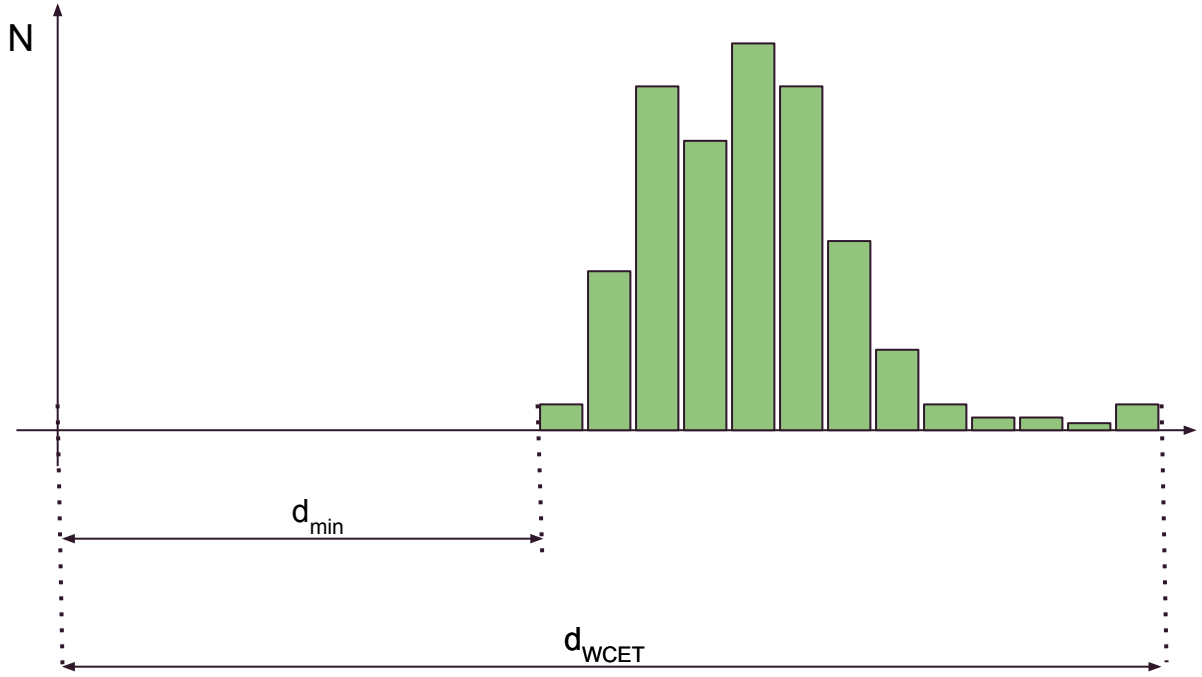
Four quantities describe the temporal behavior of an action a (e.g. task):

1. Actual duration (actual execution time) $d_{act}(a,x)$
2. Minimal duration $d_{min}(a)$: Smallest time interval it takes to complete action a
3. Worst-case execution time (WCET) $d_{WCET}(a)$: Is the maximum duration it may take to complete the action a under the assumed load and fault hypothesis – and under all possible input data
4. Jitter: $d_{WCET}(a) - d_{min}(a)$: jitter has negative impact on control systems



Action Duration Statistics

- d_{WCET} is usually very pessimistic
- only few runs will take d_{WCET}



Temporal Obligations

In a client-server model, a request from a client to a server causes a response from the server at a later time. Three temporal parameters characterize such a client-server interaction.

1. The maximum response time d_{RESP}, that is expected by the client
2. The worst-case execution time d_{WCET}, of the server
3. The minimum time d_{MINT}, between two requests

It is important to note that the d_{WCET} is in the sphere of control of the server, and the d_{MINT} is in the sphere of control of the client. In a hard real-time environment, the implementation must guarantee that the condition

$$d_{WCET} < d_{RESP}$$

holds, under the assumption that the **client keeps a minimum temporal distance** d_{MINT} between two requests.

Tasks

A **task** is the execution of a sequential program (action). It starts with reading the input data and of the internal state of the task. It terminates with the production of the results and with updating the internal state. A task that does not have an internal state at its point of invocation is called a **stateless task**; otherwise, it is called a task with state.

If there is no synchronization point within a task, we call it a **simple task** (S-task). i.e., whenever an S-task is started, it can continue until its termination point is reached. Because a S-task can not be blocked, the execution time of the task is not dependent on others tasks of the node, and can be determined in isolation.

A task is called a **complex task** (C-task) if it contains a blocking synchronization statement (e.g., a semaphore operation “wait”) within the task body. The WCET of a C-task is a global issue.

Control of action

Logical control is concerned with the control flow within a task that is determined by the given program structure and the particular input data to achieve the desired data transformation.

Temporal control is concerned with the determination of the points in time when a task must be activated or when a task must be blocked, because some conditions outside the task are not satisfied at the moment.

The only temporal control issue in an S-task is the determination of the moment, when this task must be activated. Once activated, it will run until its completion within its WCET.

A C-task blends issues of logical control with issues of temporal control. It is impossible to calculate the WCET of a C-task without analyzing the temporal properties of the complete system of interacting tasks.

Interrupts

An **interrupt** is an asynchronous hardware-supported request for a specific task activation caused by an external event (i.e., outside the node) to the currently active computation. It does not include an exception, i.e., a synchronous break in the control flow caused by a condition within the task.

The context switches for an interrupt require a **worst-case administrative overhead**, WCAO.

Every interrupt reduces the CPU activity that is available to the application by the amount of the size of WCAO. The administrative overhead is necessary even if the interrupt handler decides that the interrupt was erroneous.

The frequency of the interrupts must be limited, since if it reaches 1/WCAO no CPU capacity is left over. This requires additional protection mechanisms, because the interrupt source is outside the sphere of control of the node.

Trigger Task

A trigger task is a periodic time-triggered task that evaluates a trigger condition on a set of temporally accurate real-time variables.

The result of a trigger task can be a control signal that activates another application task.

The period of a trigger task must be smaller than the laxity (the difference between deadline and execution time) of an RT transaction that is activated by an event in the environment.

Worst Case Execution Time

A deadline for completing an RT transaction can only be guaranteed if the worst-case execution time (WCET) of all the application tasks that are part of the transaction are known a priori.

The WCET of an S-task depends on:

1. Source Code: Restrictions that must be met by a real-time program, a) no unbounded controls statements, b) no recursive function calls, and c) no dynamic data structures. The WCET-analysis of a program must determine which program path will be executed in the worst case scenario. This longest path is called the critical path.
2. Compiler Analysis: Determination of the maximum execution time of the basic language constructs of the source language (timing tree)
3. Microarchitecture Timing Analysis: Determination of the worst case execution time of the commands of the target hardware (e.g., caches)

Preemptive S-Tasks

If an S-task is preempted by another independent task, then the execution time of the S-task under consideration is extended by three terms:

1. The WCET of the interrupting task
2. The WCET of the context switch (operating system)
3. The time required for the reloading of data and instruction caches in the microarchitecture

The sum of delays 2 and 3 is called **Worst-Case Administrative Overhead** (WCAO) of a task preemption

In many applications, the microarchitecture delays (3) can be the most significant terms because the WCET of the interrupting task is normally quite short.

WCET of Complex Tasks

The WCET of a C-task depends not only on performance of the task itself, but also on the behavior of other tasks and the operating system within a node.

WCET analysis of a C-task is not a local problem of a single task, but a global problem involving all the interacting tasks within a node.

H-State and Ground State

The **h-state** at any point of interruption can be defined as the contents of the program counter and of all data structures that must be loaded into a “fresh” hardware device to continue operation at the point of operation. The size of a h-state depends on the level of abstraction and on the observation time. A small h-state eases the reintegration of a failed component.

The **ground state** of a node in a distributed system at a given level of abstraction is a state where no task is active and where all communication channels are flushed, i.e., there are no messages in transit. At the ground state, the h-state is minimal.

The reintegration of a node after a failure is simplified if a node periodically visits a ground state that can be used as reintegration point.

Real-Time Entities and Images

1. Real-Time Entity (discrete / continuous)
2. Observations
3. Real-Time Image / Object
4. Temporal Accuracy
5. Phase-Sensitivity
6. Permanence

Real-Time Entity

A **real-time entity** is a state variable of relevance for the given purpose, and is located either in the environment or in the computer system. E.g. a temperature of a liquid, voltage at a connector, a variable in a computer...

A RT entity has static attributes that does not change during lifetime (e.g., the name) and has dynamic attributes that change with time (e.g., rate of change at a point in time).

Every RT entity is in the **sphere of control** of a subsystem. Outside, it can only be observed, but **not modified**.

Continuous RT Entities

The intuitive case is that a RT entity has a **continuous** value set (continuous RT entity).

Observations at any points in time deliver a result.

Quantisation effects lead to a discrete set of values.

Discrete RT Entities

The value set of a **discrete** RT entity is defined and remains constant between a left event (L_event) and a right event (R_event).

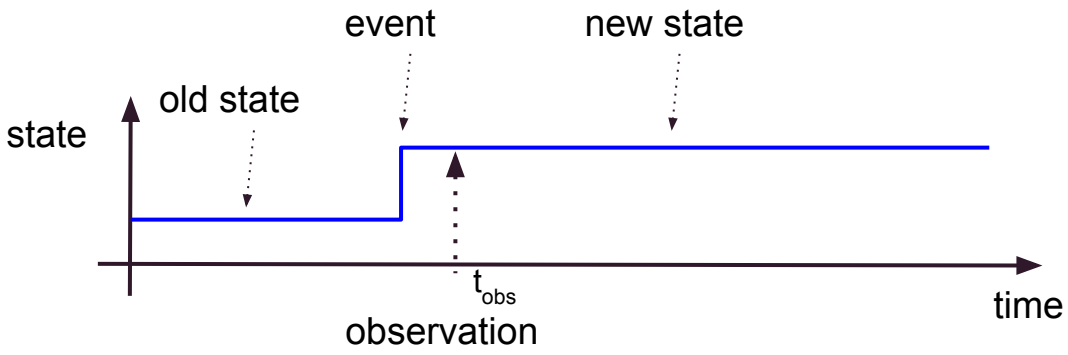
Between R_event and L_event, the set of values is undefined.

E.g., garage door, during opening, status is undefined. Observations during this transition do not deliver valid results.

Observations, States and Events

Observations

1. states can be observed
2. an event cannot be observed, only the new state can be observed.
3. every change of state is an event.



Observations

An observation of an RT entity is an atomic triple **<Name,t_{obs},Value>** consisting of the name of the RT entity, the point in real time when the observation was made, and the observed value of the RT entity.

A continuous RT entity can be observed at any point in time, whereas a discrete RT entity can only be observed between the L_event and the R_event.

An observation is a **state observation** if the value of the observation contains the absolute state of the RT entity. The time of the state observations refers to the point in time when the RT entity was sampled.

An observation is an **event observation** if it contains information about the change of value between the “old state” and the “new state”.

Event observations

There are some problems with event observations:

1. Where do we get precise time of the event occurrence?

2. An event observation contains the difference between old and new state – loss of state synchronization between observer and receiver is possible (e. g., loss of a single event observation).

3. An event observation is only sent if the RT entity changes its value. Detection of node failures is more difficult, but event observations are more efficient.

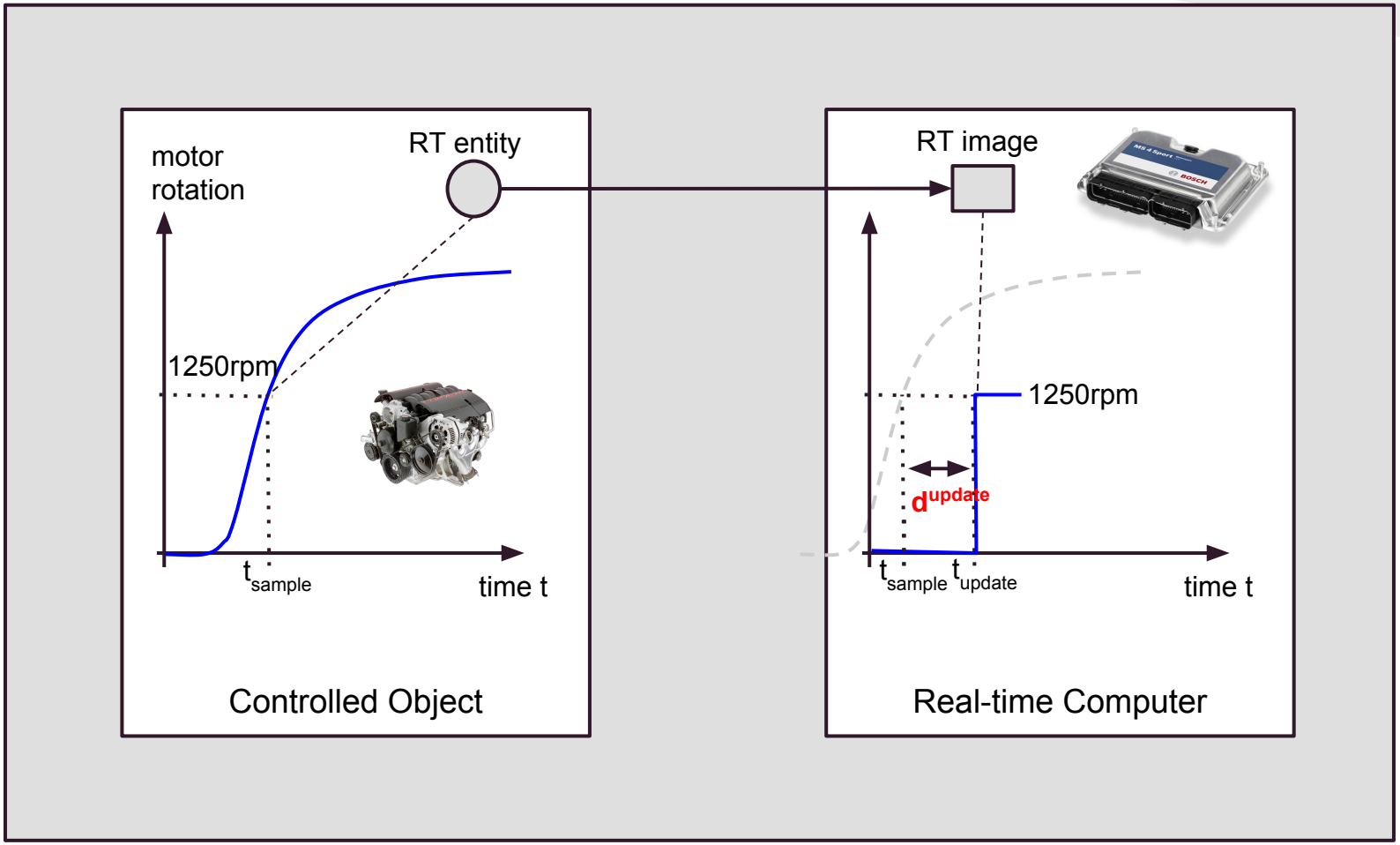
RT Images and RT Objects

A **real-time image** is a current picture of an RT entity. An RT image is valid at a given point in time if it is an accurate representation of the corresponding RT entity, both in value and in time domain.

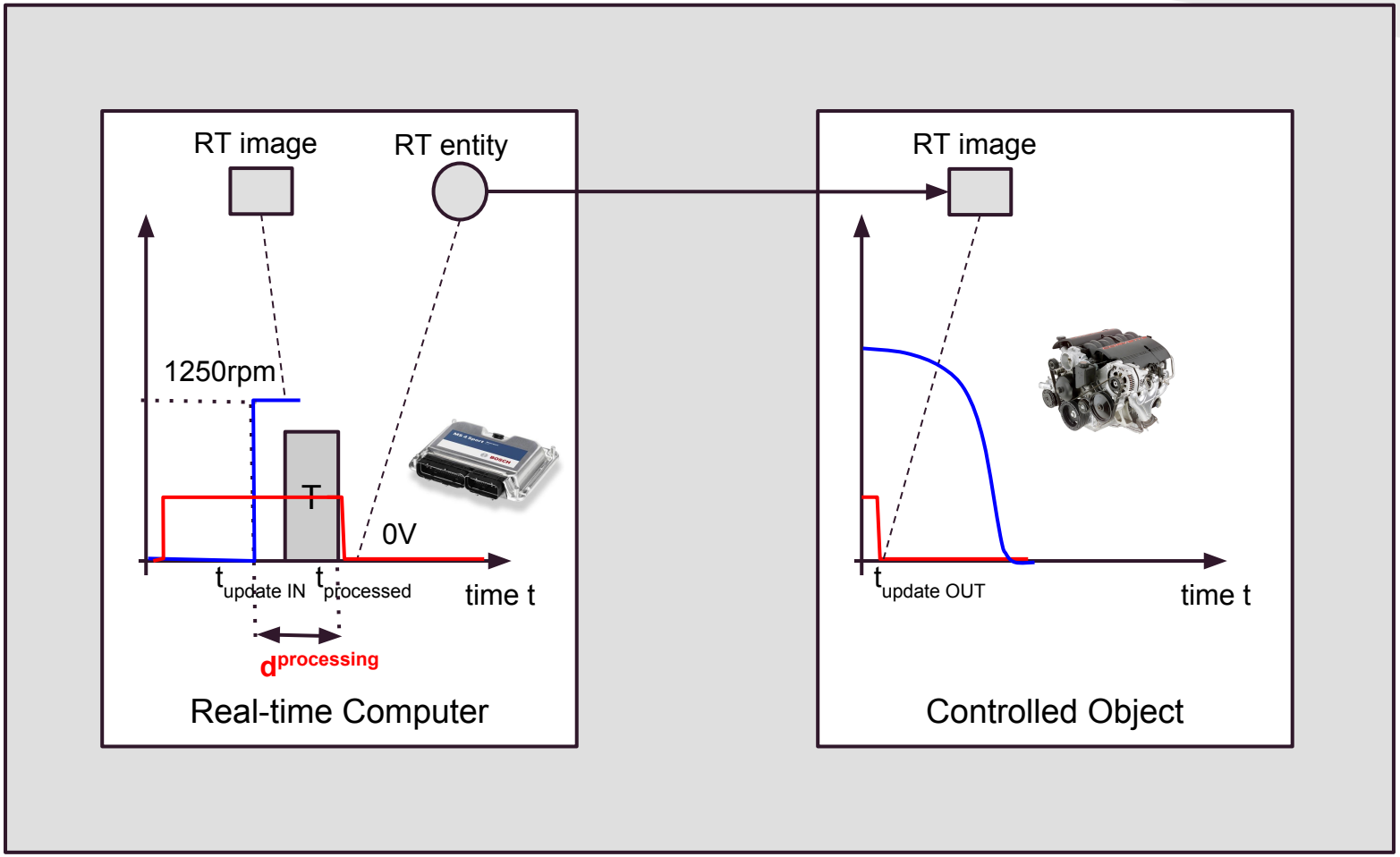
A **real-time object** is analogous to a container within a node of the distributed system holding an RT image of a RT entity. A real-time clock with a specific granularity is associated with every RT object.

A **distributed RT object** consists of a set of replicated RT objects located at different sites. Every local instance of a distributed RT object provides a specified service to its local site. Examples are clock synchronization within a specified precision, or membership service with a specified delay

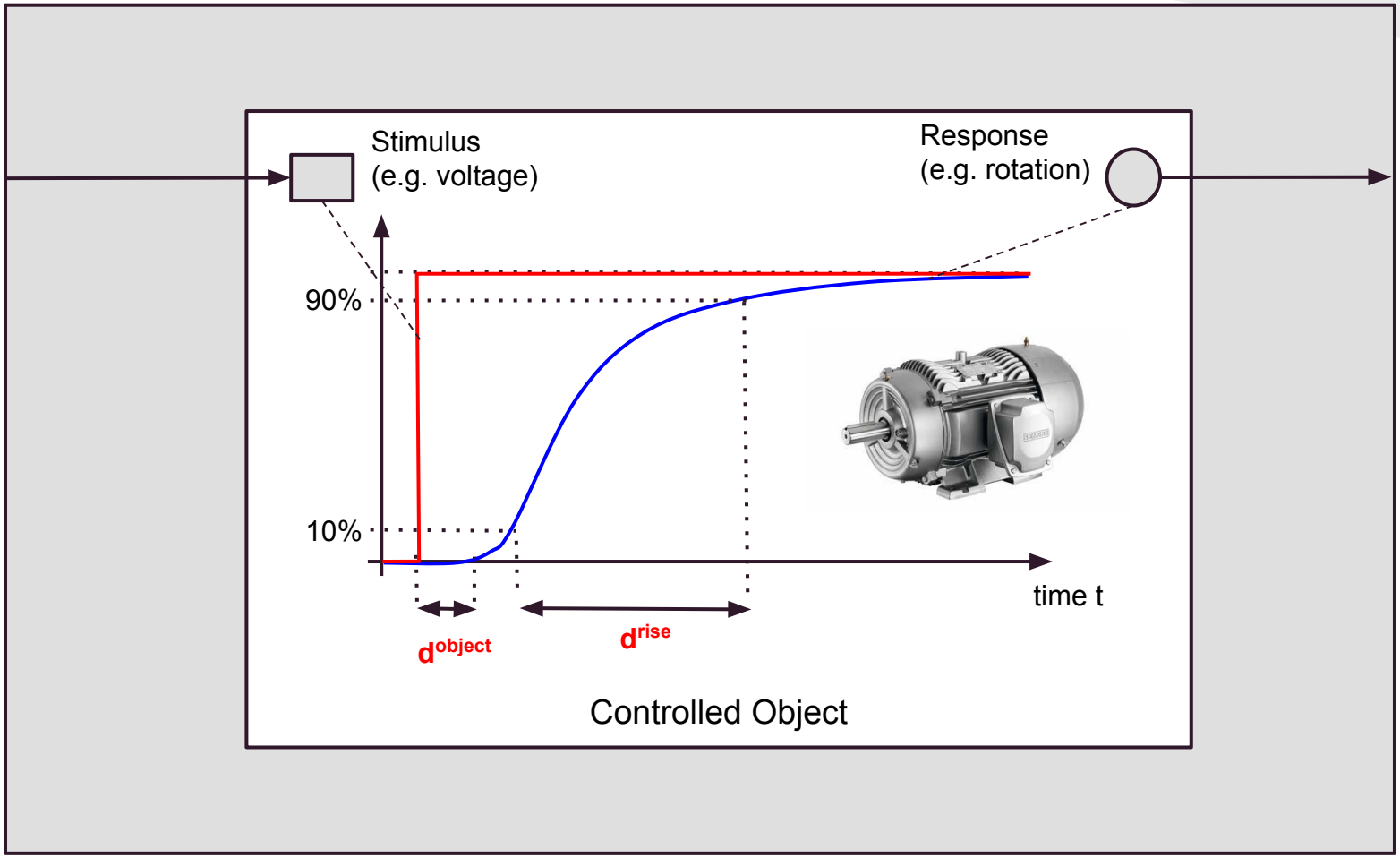
Temporal Model d^{update}



Temporal Model $d^{processing}$

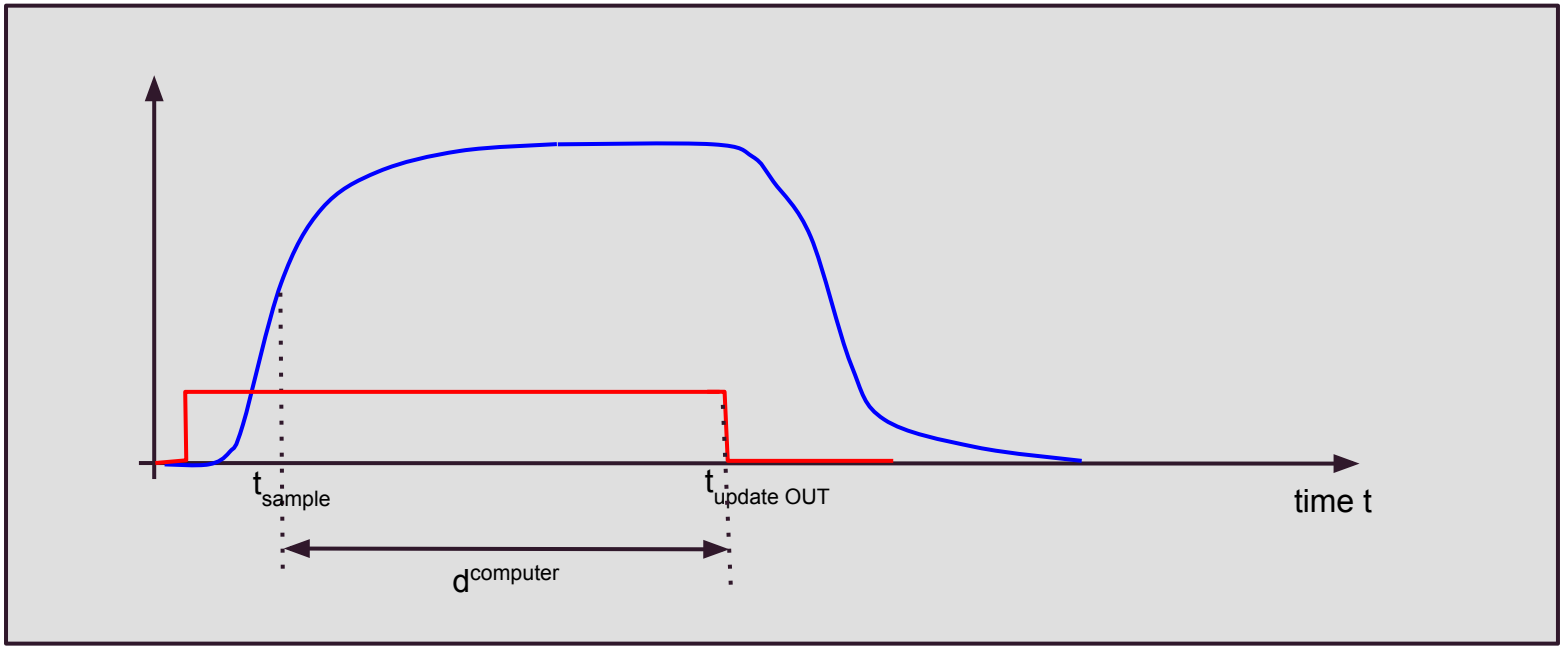


Temporal Model d^{rise} & d^{object}



Temporal Model $d^{computer}$

$$d^{computer} = d^{update\_in} + d^{processing} + d^{update\_out}$$

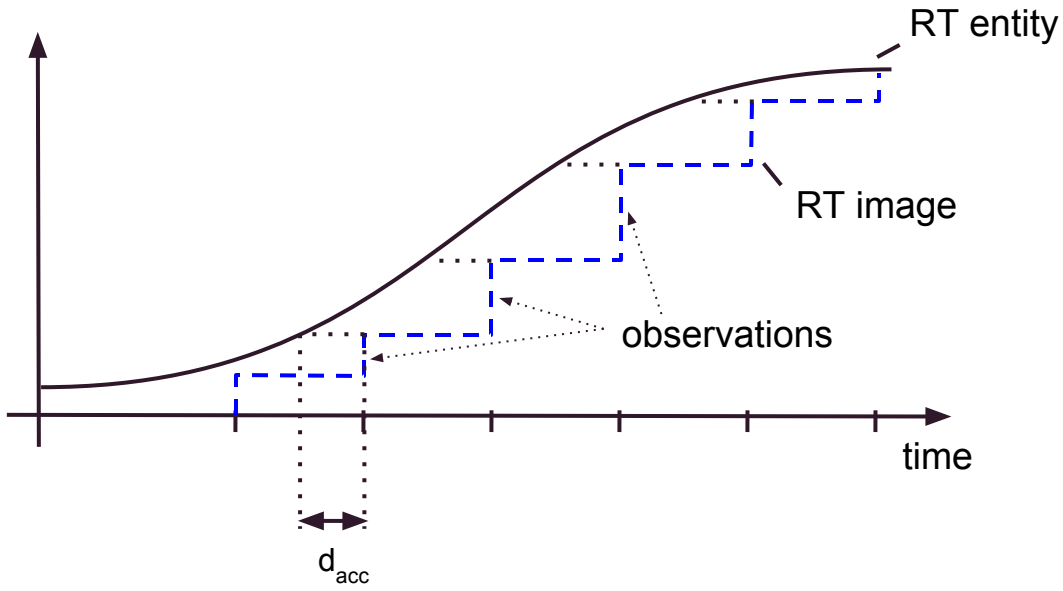


Temporal Model

d^{update}	update delay	time between observation of RT entity and update of RT image with new value
d^{object}	controlled object delay	time after which state variable begins to rise. part of the physical process of controlled object
d^{rise}	rise time of step response	part of the physical process of controlled object
$d^{processing}$	processing duration	duration of computing a new input into a new output value
d^{sample}	sampling period	$d^{sample} << d^{rise}$. depending on control requirements (e.g. $d^{sample} < d^{rise}/10$)
$\mathbf{d^{computer}}$	computer delay	time interval between the sampling point and the start of reaction of computer at output.
$\Delta\mathbf{d^{computer}}$	jitter of computer delay	distorts quality of control algorithms. $\Delta d^{computer} << d^{computer}$
$\mathbf{d^{deadtime}}$	dead time	$d^{computer} + d^{object}$

Temporal Accuracy

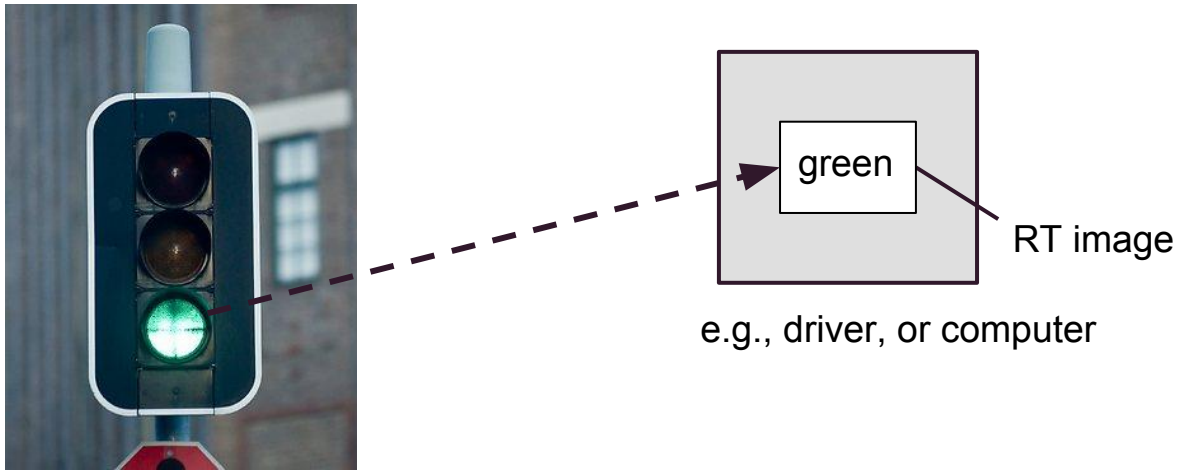
The temporal accuracy d_{acc} is the relationship between an RT entity and its associated RT image. The temporal accuracy is a relation between an RT entity and an RT object. The present value of a temporally accurate RT image is a member of set values that the RT entity had in its recent history.



Temporal Accuracy of RT Image

How long is the RT-image, based on the observation “The traffic light is green”, temporally accurate?

In detail, depending on application, however, one boundary is the duration of the yellow light phase.



Temporal Accuracy Intervall

RT Image	Change rate	Required accuracy	d_{acc}
piston in cylinder	6000 rpm	0,1°	2,77 us
position of gas pedal	100% / s	1%	10 ms
engine load	50% / s	1%	20 ms
oil temperature	10% / minute	1%	6 s

Figure: Temporal Accuracy Examples (Source: [1] p.105)

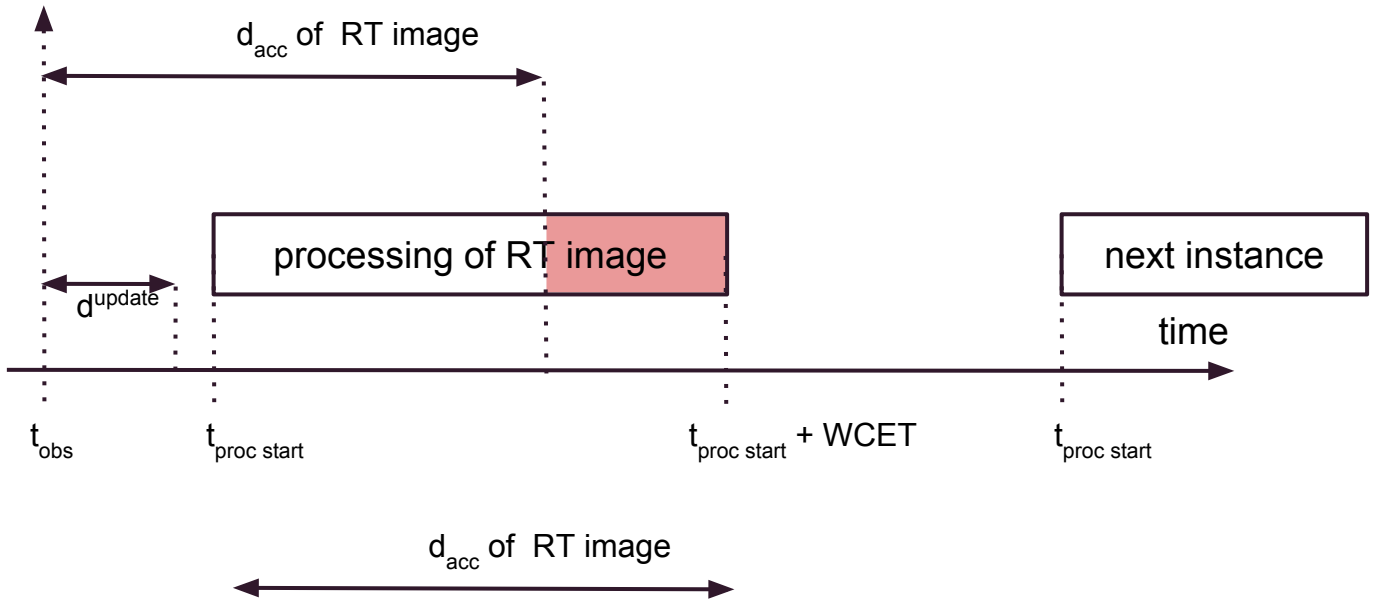
Error of the RT Image

The delay between the observation of the RT entity and the use of the RT entity can cause, in the worst case, a maximum deviation **error(t)** of the RT image. That error can be approximated by the product of the maximum gradient of the value v of the RT entity multiplied by the length of the accuracy interval plus the maximum of the quantisation error.

If an RT entity changes its value quickly, a short accuracy interval must be maintained.

Phase-sensitive RT Images

Depending on the accuracy interval d_{acc} and the duration of update from RT entity to RT image d^{update}, the RT image is either phase sensitive, or phase insensitive. A **phase sensitive** RT image cannot be accessed at arbitrary times by the processing task.



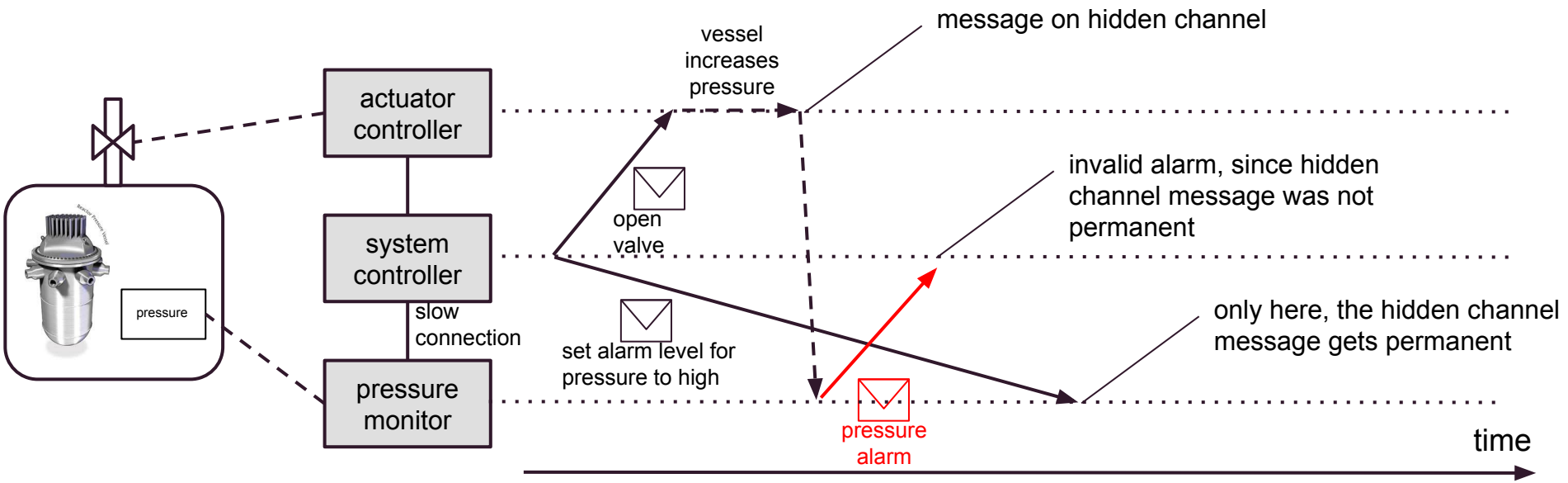
Phase-sensitive RT Images

Each phase-sensitive RT image imposes an additional constraint on the scheduling of the real-time task that uses this RT image – so, try to minimize the number of phase sensitive RT images in a real-time system.

This can be done by either increasing the update frequency of the RT image or by deploying a state estimation model to extend the temporal accuracy. It is a tradeoff between utilizing communication or processing resources.

Permanence

Permanence is a relation between a particular message arriving at a node and the set of all messages that have been sent to this node before. A message becomes **permanent** at a given node at that point in time when the node knows that all messages that have been sent to it prior to the send time of the message, have arrived or will never arrive.



Action Delay

The time interval between the start of transmission of a given message and the point in time when this message becomes permanent at the receiver, is called the action delay.

The receiver must delay any action on the message until after the action delay has passed to avoid an incorrect behavior.

An RT image may only be used if the message that transported the image is permanent, and the image is temporally accurate. Both conditions can only be satisfied in the time window ($t_{permanent}$,t_{obs}+d_{acc}).

d_{acc} depends on the dynamic on the application, while $t_{permanent}$- t_{obs} is an implementation specific duration

Idempotency

Idempotency is the relationship among the members of a set of replicated messages arriving at the same receiver. No matter whether the receiver receives one or more out of a set of replicated idempotent messages, the result will always be the same.

If messages are idempotent, the implementation of fault tolerance by means of replicating messages is simplified.

References

[1] Real-Time Systems: Design Principles for Distributed Embedded Applications, Hermann Kopetz, 1997, Springer Verlag

[2] http://www6.in.tum.de/pub/Main/TeachingWs2009Echtzeitsysteme/20100127_Zeit.pdf