RT System Modeling

Overview

- ♦ Model Construction
- ♦ Components with State
- ♦ Interfaces
- ♦ Real-Time Entities and Observations
- ♦ Temporal Accuracy
- ♦ Permanence and Idempotency
- ♦ Replica Determinism

What is a *Model*?

A model M of a system S and an experiment E is anything which can be applied on E in order to answer a question about S.

(M.Minky, 1965)

Model Construction

- ♦ Focus on the essential properties--eliminate the unnecessary detail (viewpoint important).
- ♦ The elements of the model and the relationships between the elements must be well specified.
- ♦ Understandability of the structure and the functions of the model is important.
- ♦ Formal notation to describe the properties of the model should be introduced to increase the precision.
- ♦ Model assumptions must be stated explicitly.

Assumption Coverage

Every model/design is based on a set of assumptions about the behavior of the components and the environment.

Assumption coverage: The probability that the assumptions cover the real-world scenario.

The dependability of a *perfect design* is limited by the assumption coverage (Also limits the utility of formal verification).

Specification of the assumptions is a system engineering task.

Load and Fault Hypothesis

Two important assumptions that must be contained in the requirements specification:

- ♦ Load Hypothesis: Specification of the peak load that a system must handle.
- ♦ Fault Hypothesis: Specification of the number and types of faults that a fault-tolerant system must tolerate.

The fault hypothesis partitions the fault space into two domains: those faults that must be tolerated and those faults that are outside the fault-tolerance mechanisms.

Outside Fault Hypothesis: NGU (Never-give-up) Strategy

Elements of a RT System Model

Essential:

- ♦ Representation of Real-Time
- ♦ Semantic Properties of Data Transformations
- ♦ Durations of the executions

Unnecessary Detail:

- ♦ Representation of information within a system (this is only important at the interfaces --specified by architectural style).
- ♦ Detailed characteristics of Data Transformations
- ♦ Time Granularity that is finer than the application requirement

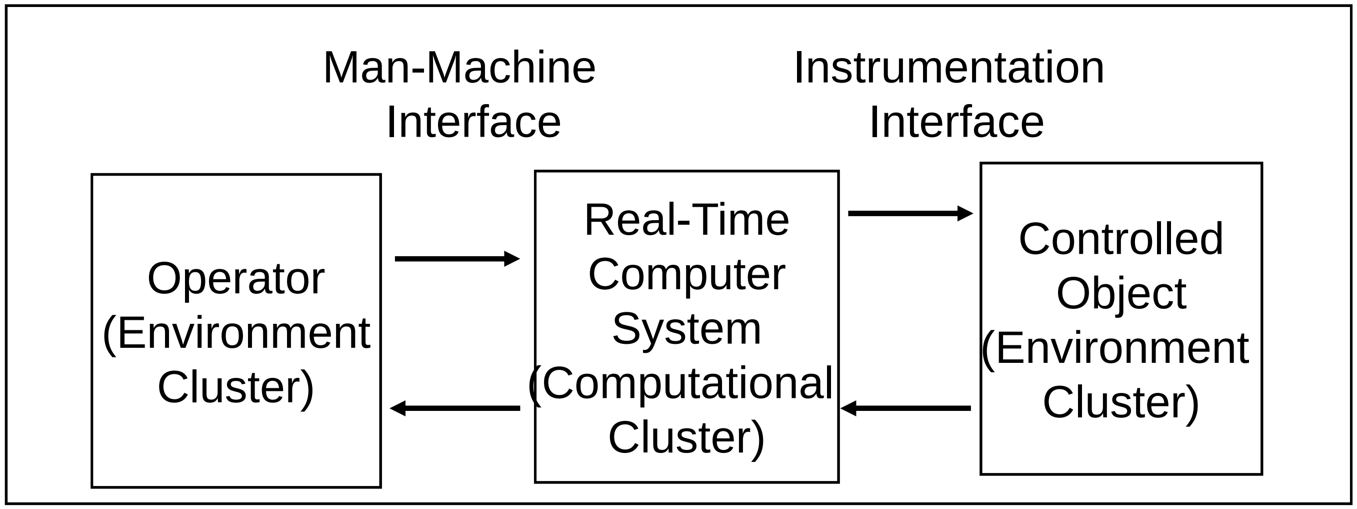
Structure Overview

Real-time System: Computer System + Controlled Object + Operator

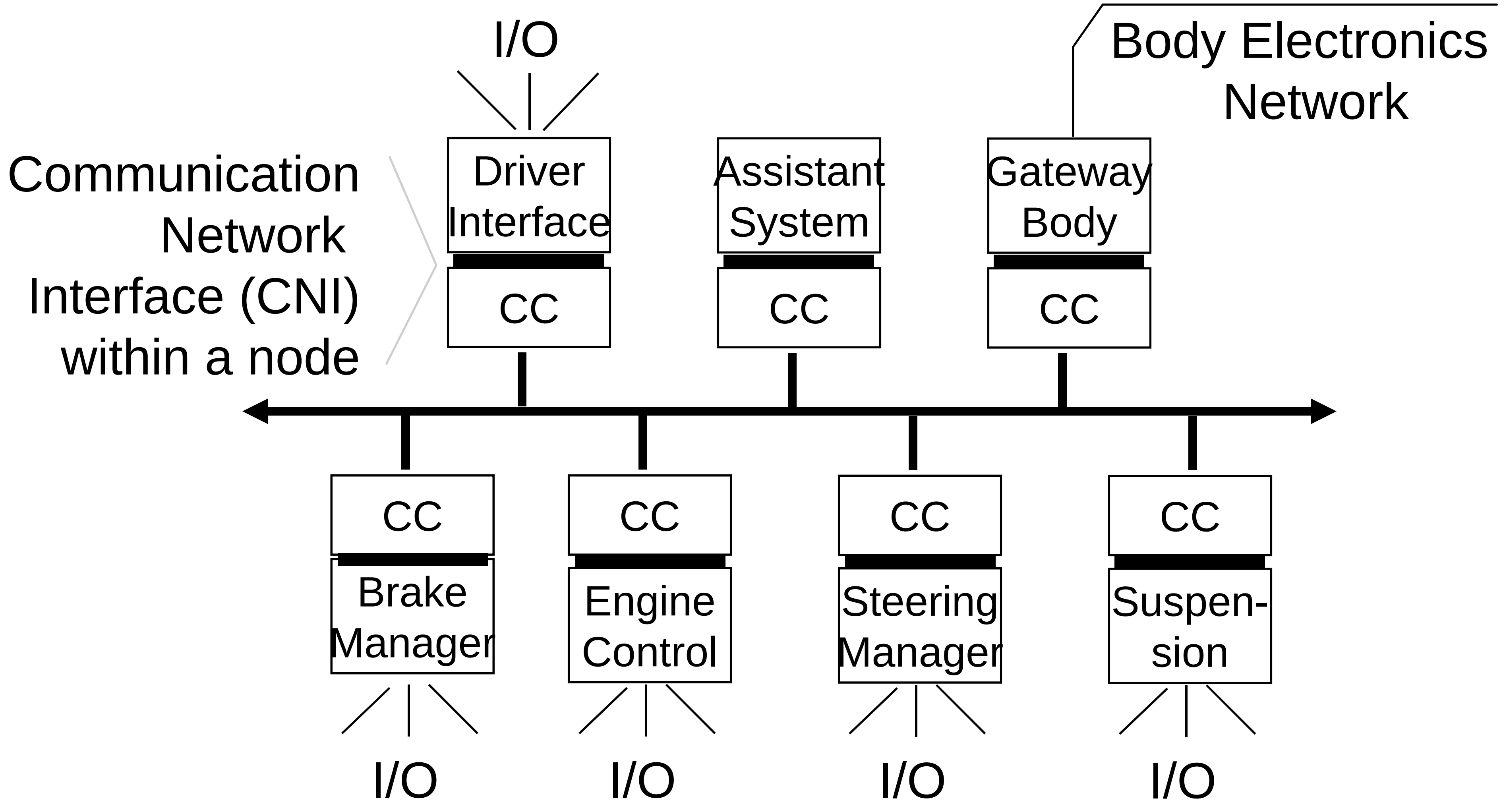
Cluster: A subsystem of the RT-system with high inner connectivity

Node: A hardware software unit of specified functionality

Task: The execution of a program within a component


Man-Machine Interface
Instrumentation Interface
Operator (Environment Cluster)
Real-Time Computer System (Computational Cluster)
Controlled Object (Environment Cluster)

Example of a Cluster



CC: Communication Controller

Characteristics of a Cluster

A cluster within the computing system consists of a set of co-operating components that

- ♦ provide a specified service to some subsection of the environment
- ♦ support an internally unified representation of the information (messages)
- ♦ solve the dependability problem, e.g., by grouping replica determinate components into Fault Tolerant Units
- ♦ provide small interfaces to other clusters, not necessarily in a hierarchical fashion

What is a Component?

Agreement at an abstract level:

A component is a *building block* for the construction of large systems. It must be possible to integrate components on the basis of their *service interface specifications* without any knowledge of the *internal structure and the internal behavior* of the component.

Is this ***building block***

- A *software unit* (software component) for independent deployment or a
- A *hardware-software unit* (system component) that has behavior and state?

Component in our Context?

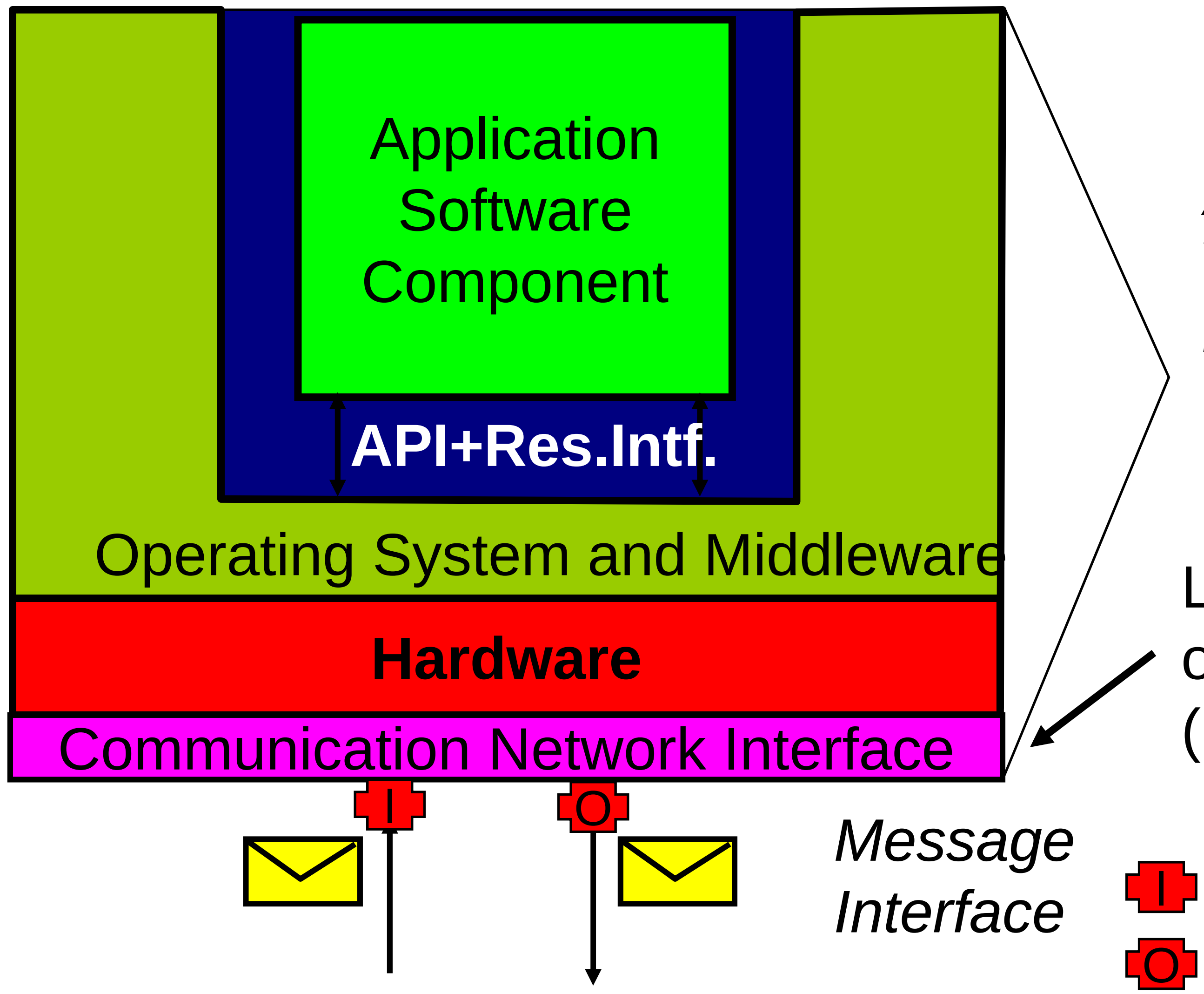
In our context, a *component* is a complete computer system--*a node*-- that is time aware. It consists of

- ♦ The hardware
- ♦ The system and application software
- ♦ The internal state

The component interacts with its environment by the exchange of messages via interfaces.

What is a *software* component?

Software Components vs. System Components


Application Software Component
API+Res.Intf.
Operating System and Middleware
Hardware
Communication Network Interface
I
O
Message Interface

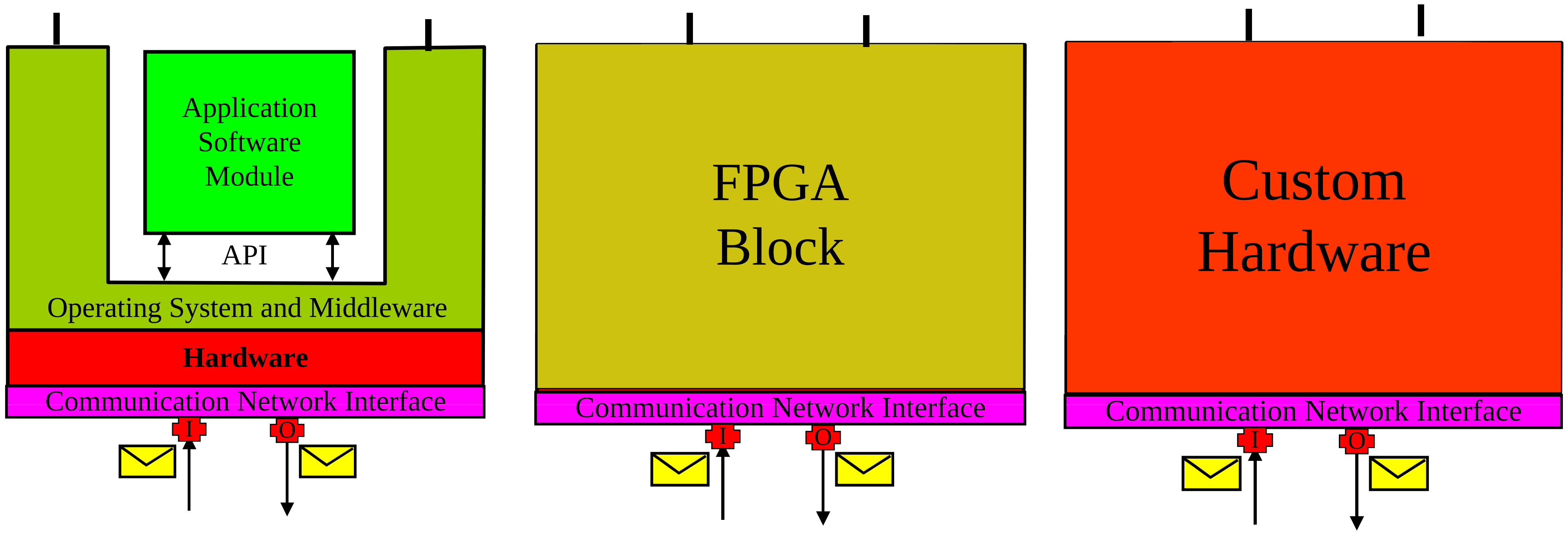
Programmable System Component consists of *Application Software Module* plus *Container* (*Operating System, Midddleware, Hardware)*

Linking Interface of the Component (LIF)

I Input Port
O Output Port

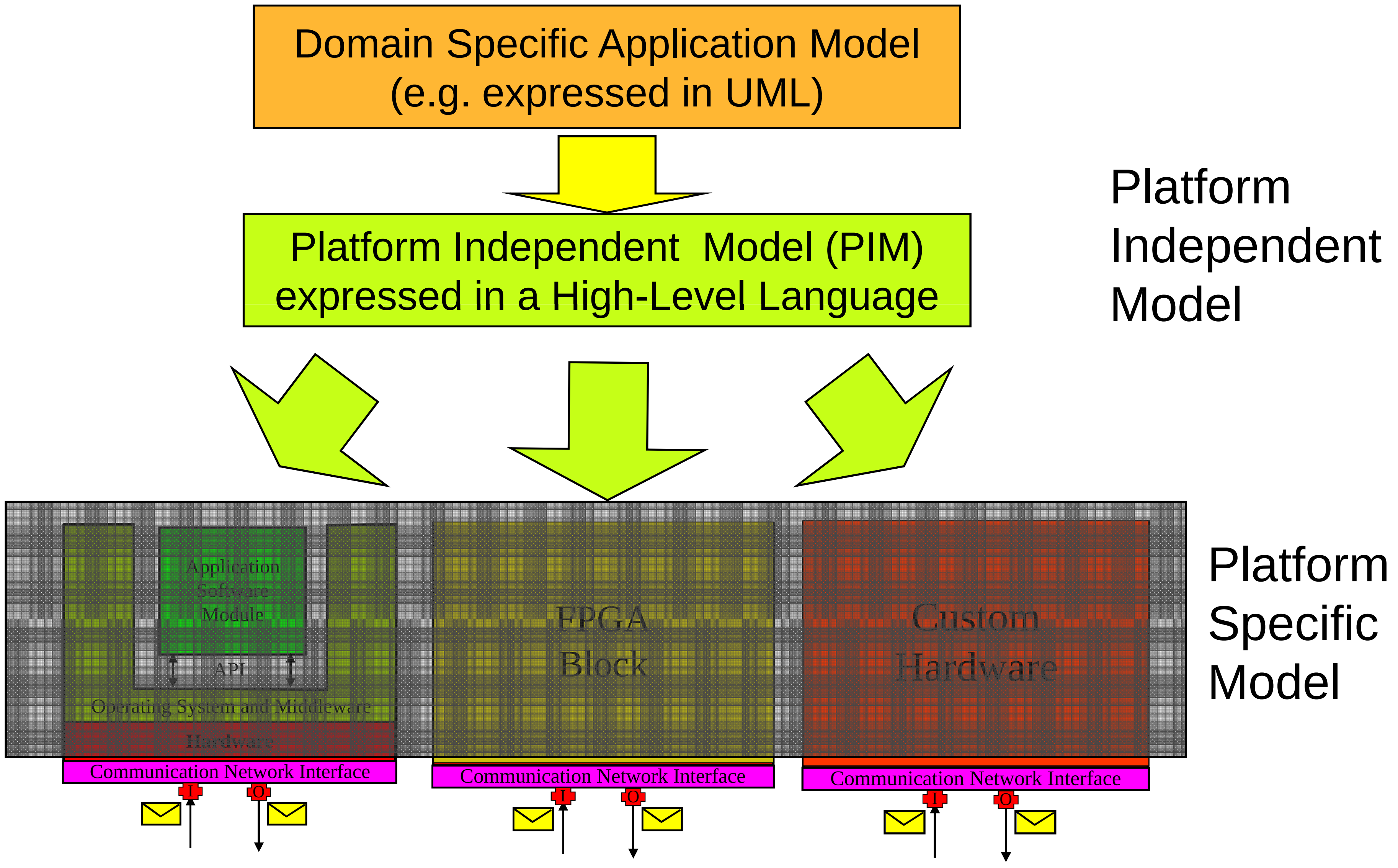
Different Types of System Components

Local Interfaces--Open Components


Application Software Module
API
Operating System and Middleware
Hardware
Communication Network Interface
I
O
FPGA Block
Communication Network Interface
I
O
Custom Hardware
Communication Network Interface
I
O

The Communication Network Interfaces (CNI) of all three different types of system components should have the same *syntax, timing* and *semantics*. For a user, it should not be discernible which type of system component is behind the CNI.

Model Driven Design: From the PIM to the PSM


Domain Specific Application Model
(e.g. expressed in UML)
Platform Independent Model (PIM)
expressed in a High-Level Language
Platform Independent Model
Application Software Module
API
Operating System and Middleware
Hardware
Communication Network Interface
I
O
FPGA Block
Communication Network Interface
I
O
Custom Hardware
Communication Network Interface
I
O
Platform Specific Model

Closed Component vs. Open Component

- ♦ **Closed Component:** Contains no local interface to the *real world*, but can contain local interfaces to other closed components.
  Semi-closed if it is time-aware.
- ♦ **Open Component:** Contains an interface to the *real world*.
  Semi-open if no control signals are accepted from the real-world (e.g., a sampling system).

The real world has an unbounded number of properties.

Component contains State

A component consists of:

- ♦ Hardware: CPU, memory, real-time clock, process I/O, communication interface
- ♦ Operating System: CPU Management, I/O support, communication support, error manager
- ♦ i-state(initialization state): static data structure, i.e., application program code, initialization data (can be put into ROM)
- ♦ h-state (history state): dynamic data structure that contains information about the current and past computations (has to be put into RAM)
- ♦ g-state (ground state): minimal h-state when all tasks are inactive and all channels are flushed--*reintegration point*.

Consistent Notion of State

A system-wide consistent notion of a discrete time is a prerequisite for a consistent notion of state, since the notion of *state* is introduced in order to separate the *past* from the *future*:

“The state enables the determination of a future output solely on the basis of the future input and the state the system is in. In other word, the state enables a “decoupling” of the past from the present and future. The state embodies all past history of a system. Knowing the state “supplants” knowledge of the past. Apparently, for this role to be meaningful, the notion of past and future must be relevant for the system considered.” (Taken from Mesarovic, Abstract System Theory, p.45)

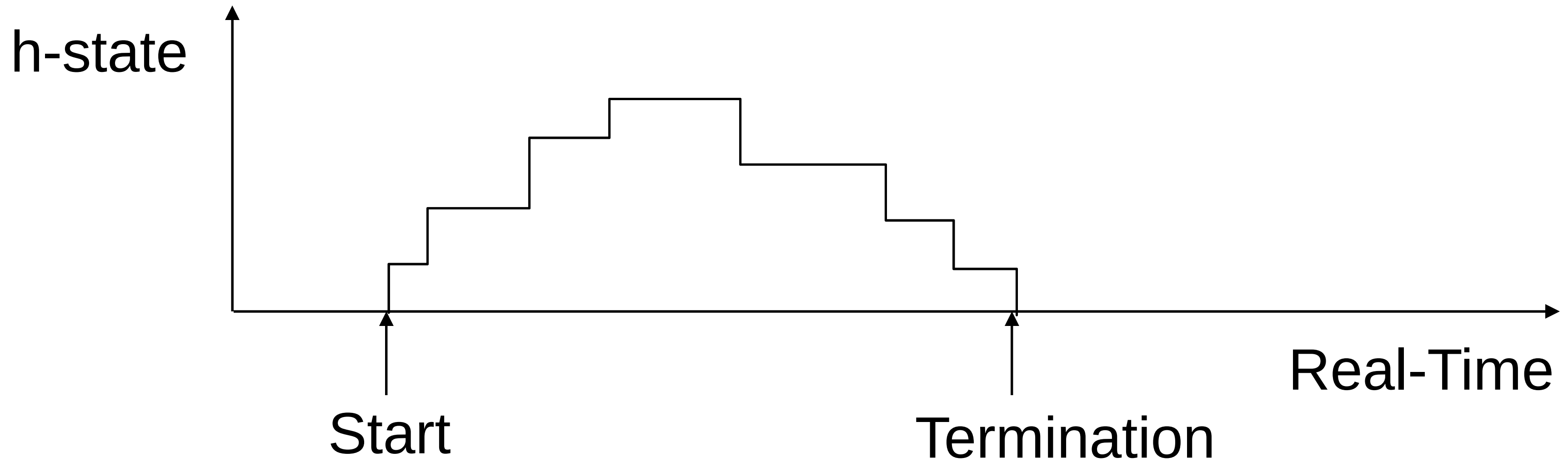
History State (h-state)

The h-state comprises all information that is required to start an “empty” node (or task) at a *given point in time*:

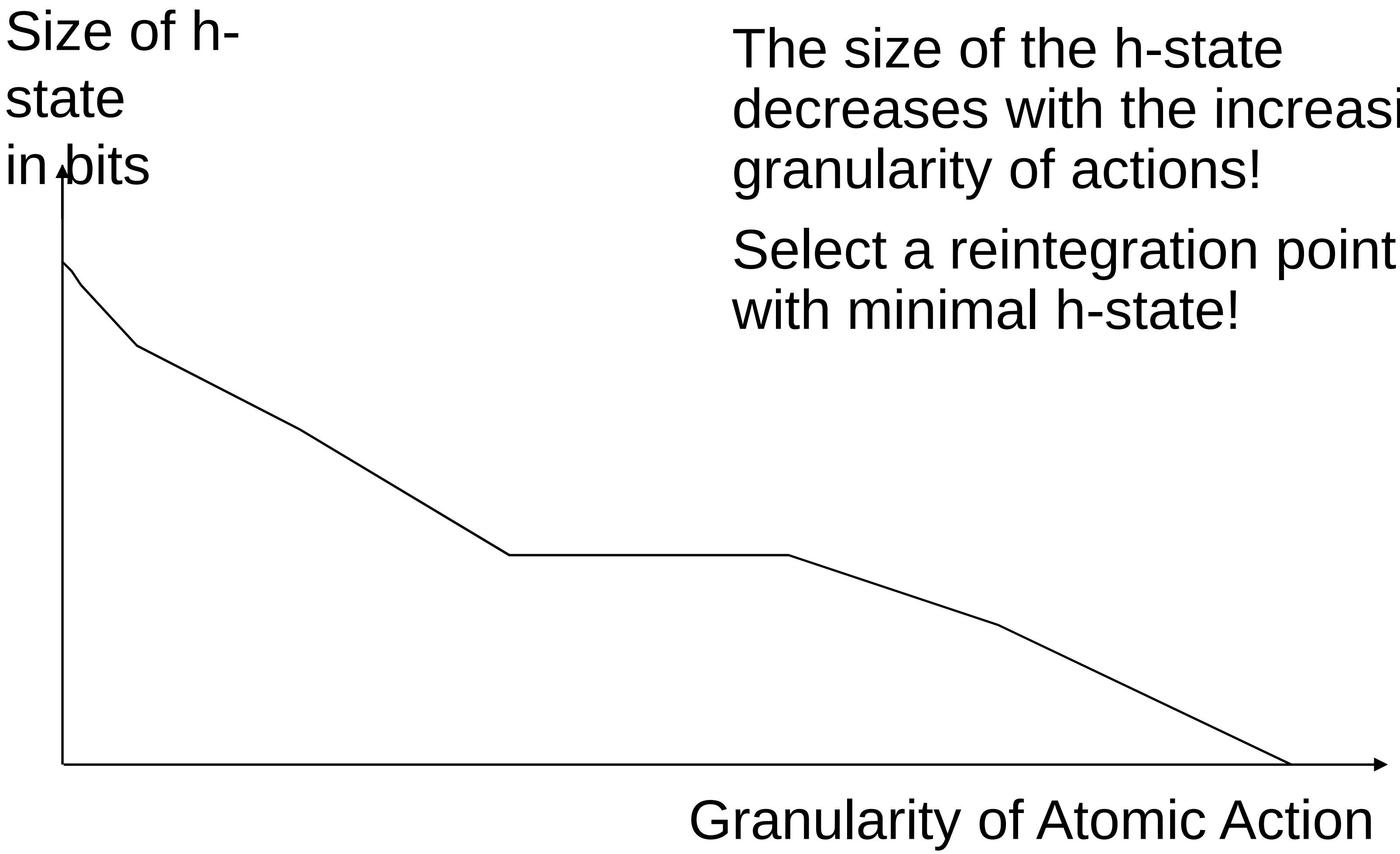
- ♦ Size of the h-state depends on the point in time chosen
- ♦ relative minimum immediately after a computation (an atomic action) has been completed.
- ♦ System in *ground state*: no messages in transit and no activity occurring.
- ♦ shall be small at reintegration points.

If no h-state has to be stored between successive activations of the node, the node is called “stateless” (at the chosen level of abstraction!).

Size of H-State During an Atomic Action



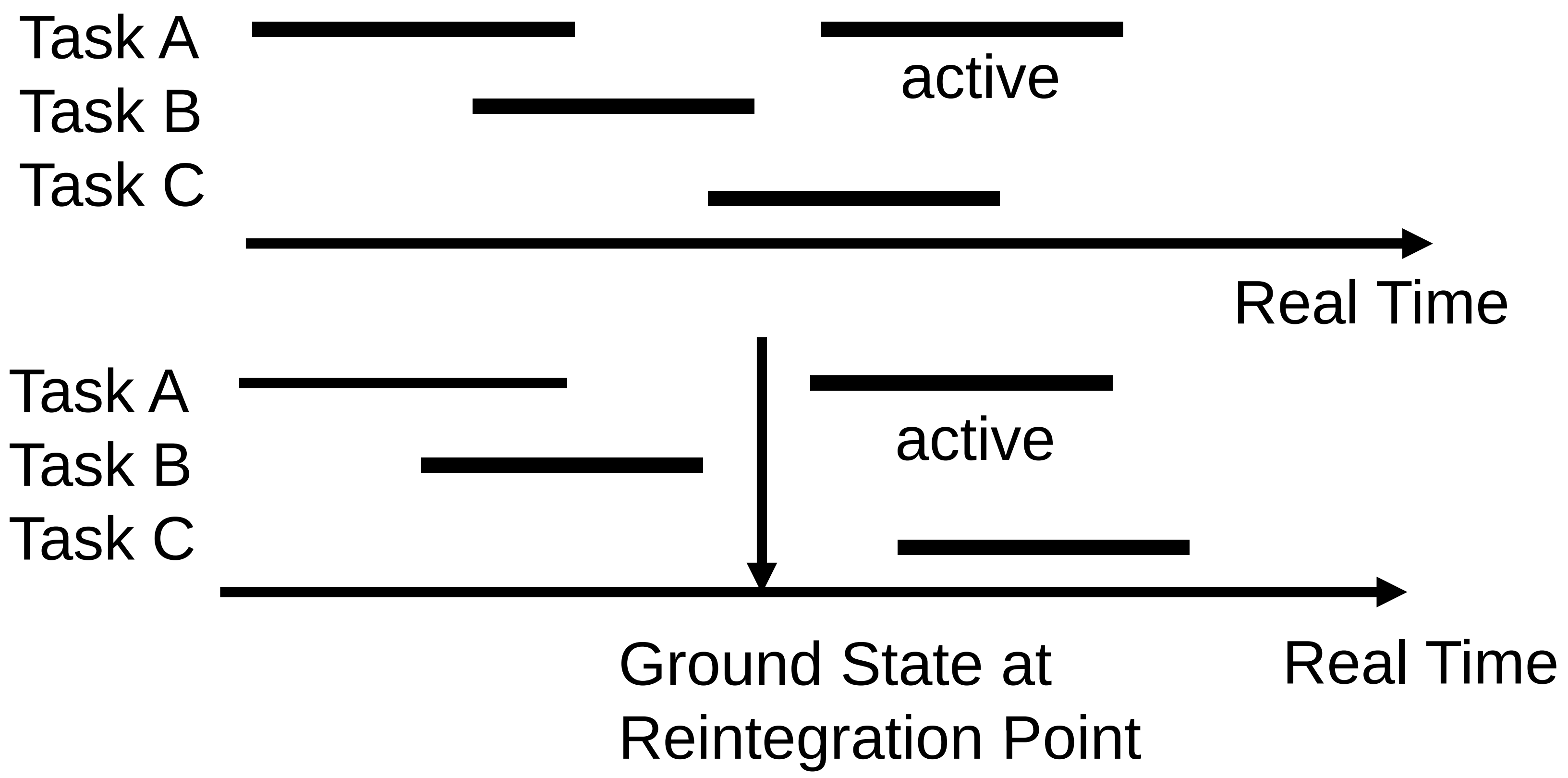
Size of h-state at the End of an Atomic Action



The size of the h-state decreases with the increasing granularity of actions!

Select a reintegration point with minimal h-state!

Ground State (g-state)



g-state: Minimal h-state of a subsystem (node) where are tasks are inactive and all channels are flushed. Needed for
reintegration of nodes.

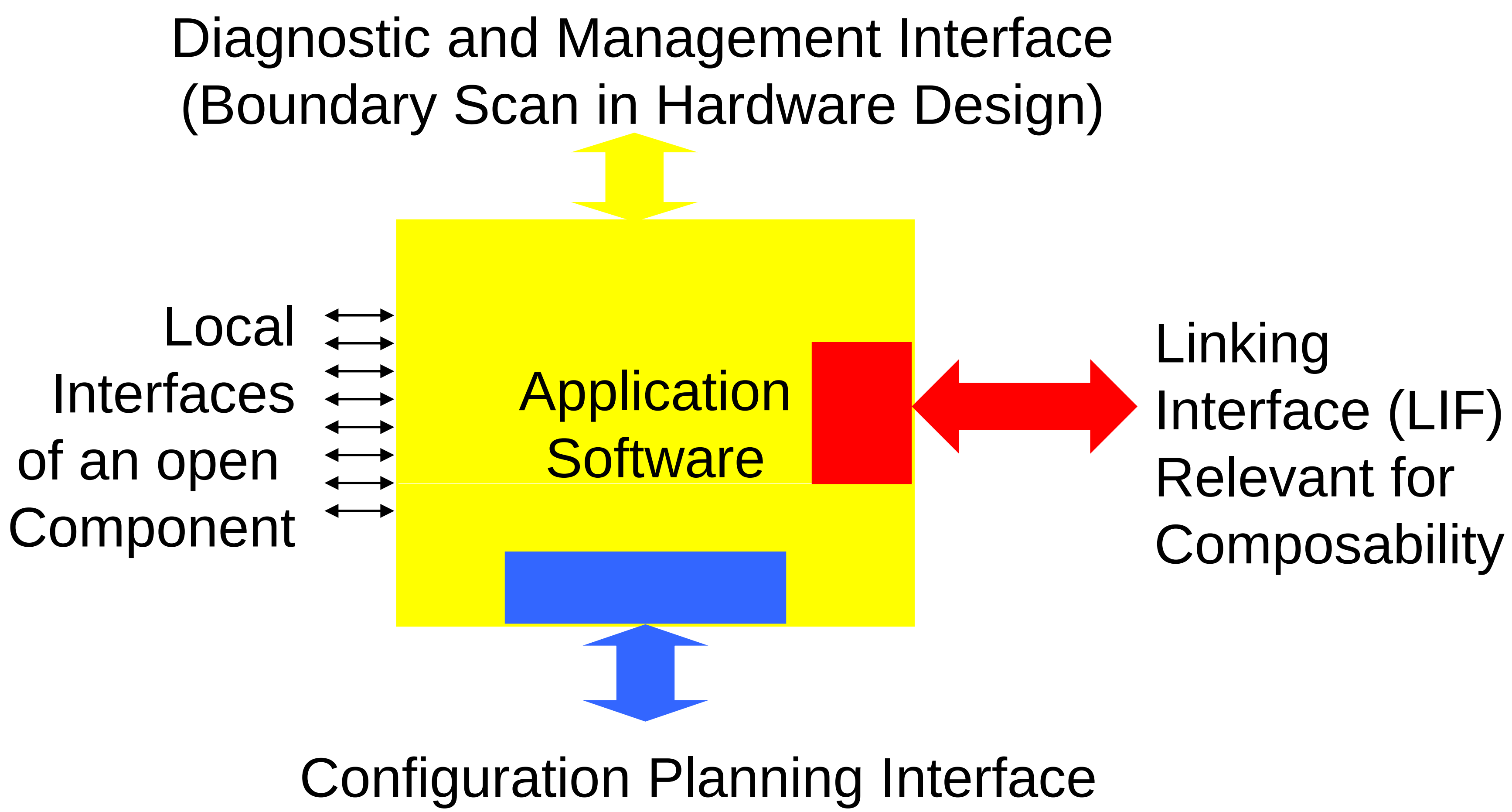
Interface

The interface between two subsystems (cluster, component, etc.) is a common boundary that is characterized by

- ♦ Its *data properties*, i.e., the structure and semantics of the data items crossing the interface. The semantics include the *functional intent*, i.e., the assumptions about the functions of the interfacing partner
- ♦ Its *temporal properties*, i.e., the temporal conditions that have to be satisfied by the interface: control and temporal data validity.

In a non-real-time computer system, there is little concern about the temporal properties.

Interfaces of a Component


Diagnostic and Management Interface
(Boundary Scan in Hardware Design)
Local
Interfaces
of an open
Component
Application
Software
Linking
Interface (LIF)
Relevant for
Composability
Configuration Planning Interface

The Four Interfaces of a Component

Realtime Service (RS) Interface--LIF:

- ♦ In control applications periodic
- ♦ Contains RT observations
- ♦ Time sensitive

Diagnostic and Maintenance (DM) Interface:

- ♦ Sporadic access
- ♦ Requires knowledge about internals of a node
- ♦ Not time sensitive

Configuration Planning (CP) Interface:

- ♦ Sporadic access
- ♦ Used to install a node into a new configuration
- ♦ Not time sensitive

Local Interface to the Component Environment

LIF is Important for Composability

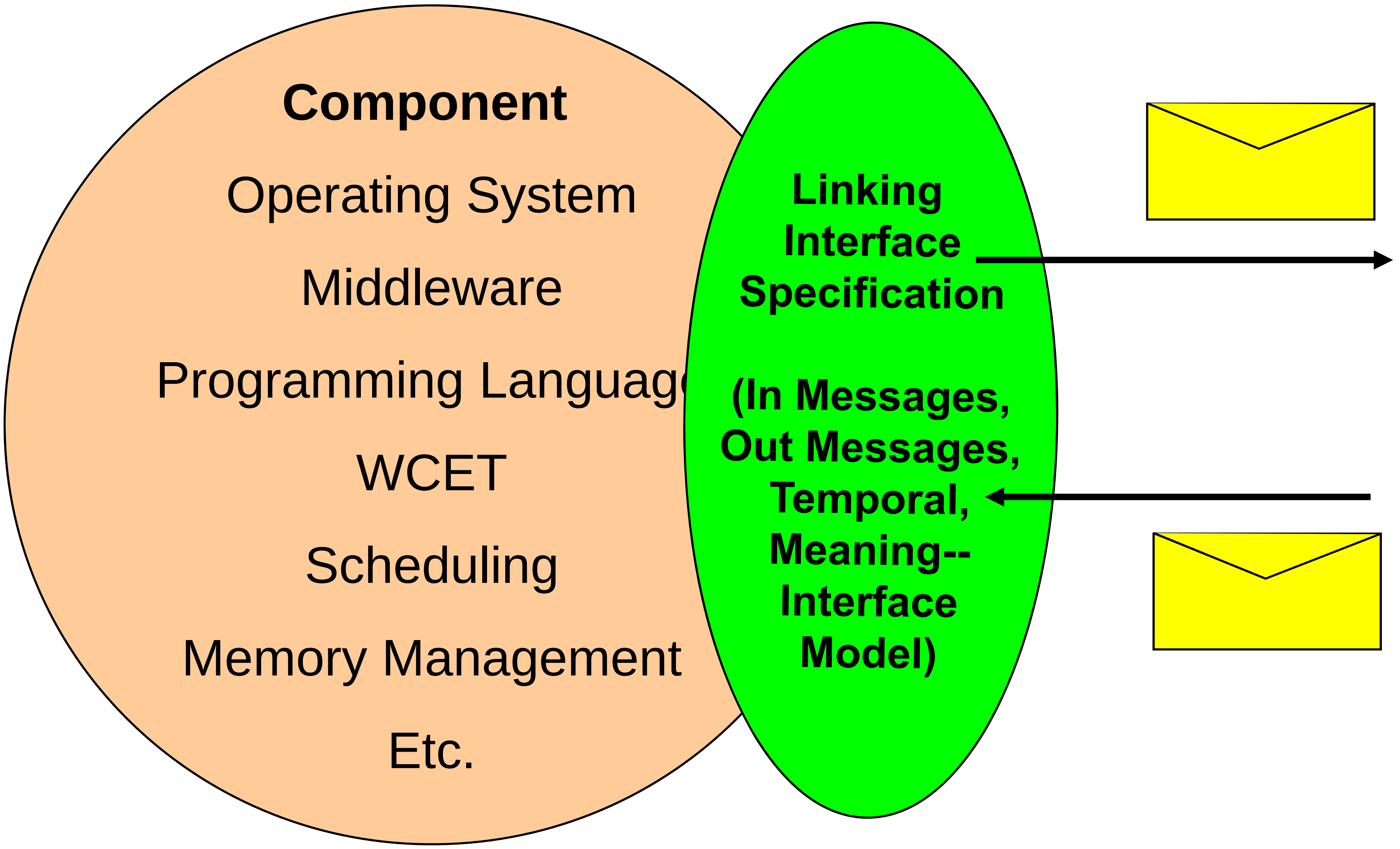
For the temporal composability, only the LIF interface is relevant.

An LIF (e.g., a control algorithm) must specify:

- ♦ At what point in time the input information is delivered to a module (temporal pre-conditions)
- ♦ At what point in time the output information must be produced by the module (temporal post-conditions).
- ♦ The properties of the intended information transformation provided by the module (a proper model)

Focus on Message Based Interfaces!

The LIF Specification hides the Implementation


Component
Operating System
Middleware
Programming Language
WCET
Scheduling
Memory Management
Etc.
Linking Interface Specification
(In Messages, Out Messages, Temporal, Meaning-- Interface Model)

LIF Specification

Operational Specification

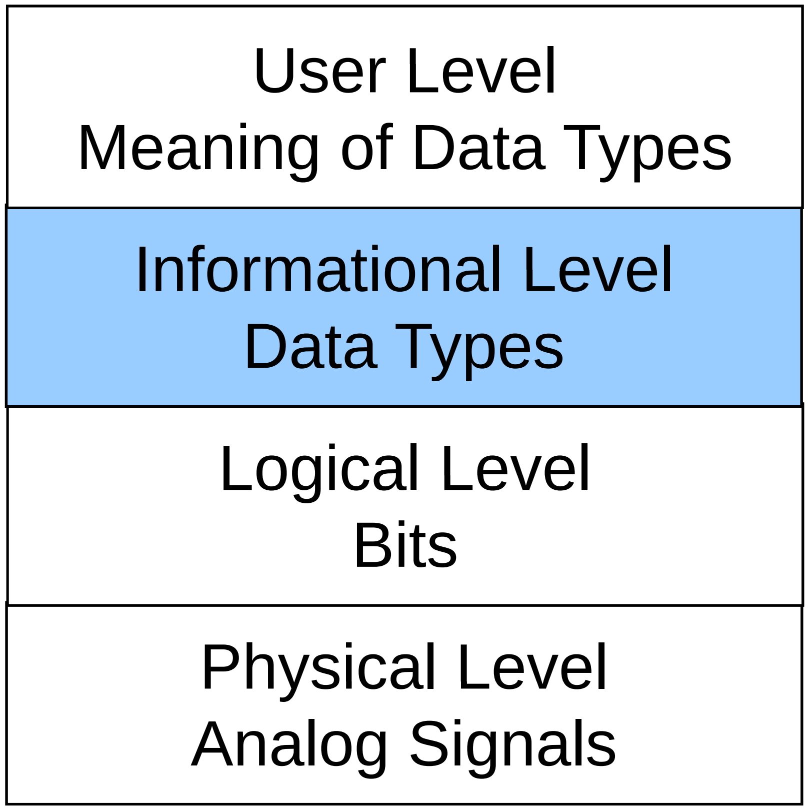
- ♦ Data Domain: Specification of syntactic units contained in a message, e.g, by an IDL. Bridges the gap between the logical level and the information level
- ♦ Temporal Domain: Specify the instants, the rate, the order, the phase relationship between messages

Meta-level Specification

- ♦ Assign meaning to the syntactic units generated by the operational specification by referring to a LIF service model. Bridges the gap between the information level and the user level

.

Views of a System: Four Universe Model



Meta-level Specification
Interpretation by the User

Operational Interface Specification
Value and Temporal

Avizienis, FTCS 12, 1982

LIF Specification (ii)

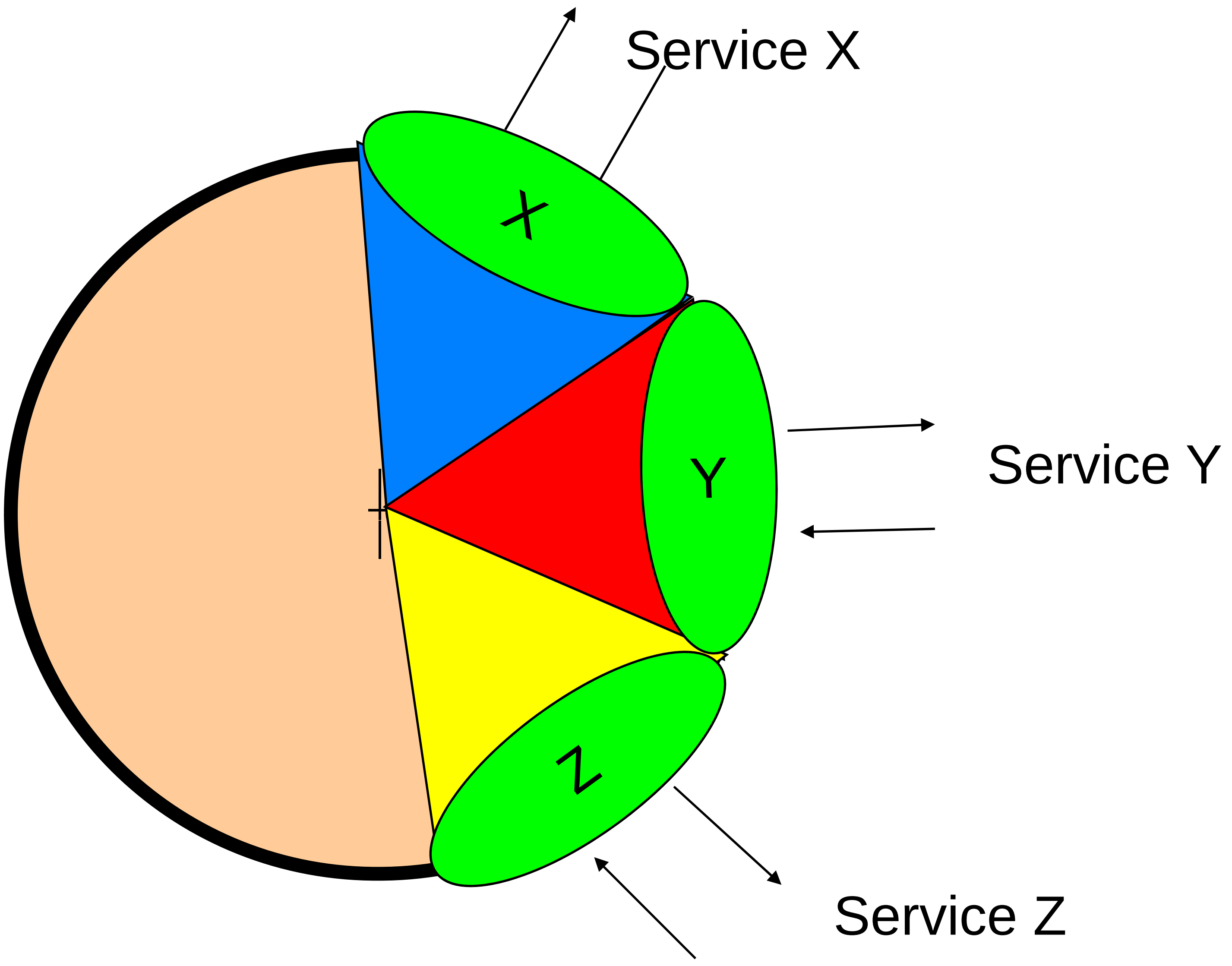
Operational Specification:

- ♦ **Operational Input Interface Specification**
  - Syntactic Specification
  - Temporal Specification
  - Input Assertion
- ♦ **Operational Output Interface Specification**
  - Syntactic Specification
  - Temporal Specification
  - Output Assertion
- ♦ **Interface State**

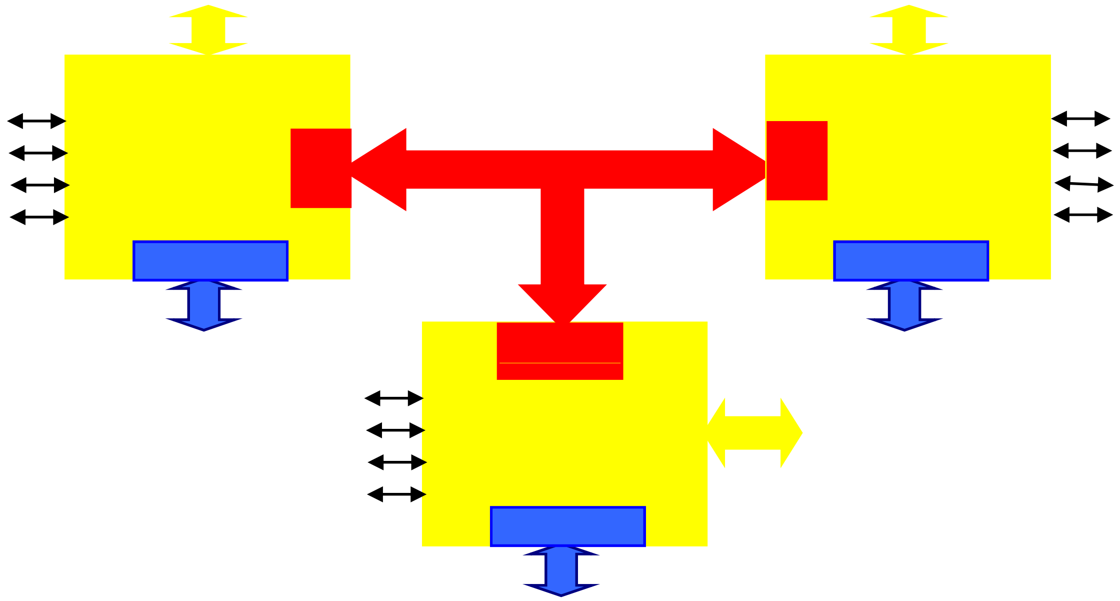
Meta-level Specification:

- ♦ Meaning of the data elements: Means-and-ends model

A Component may support many LIFs



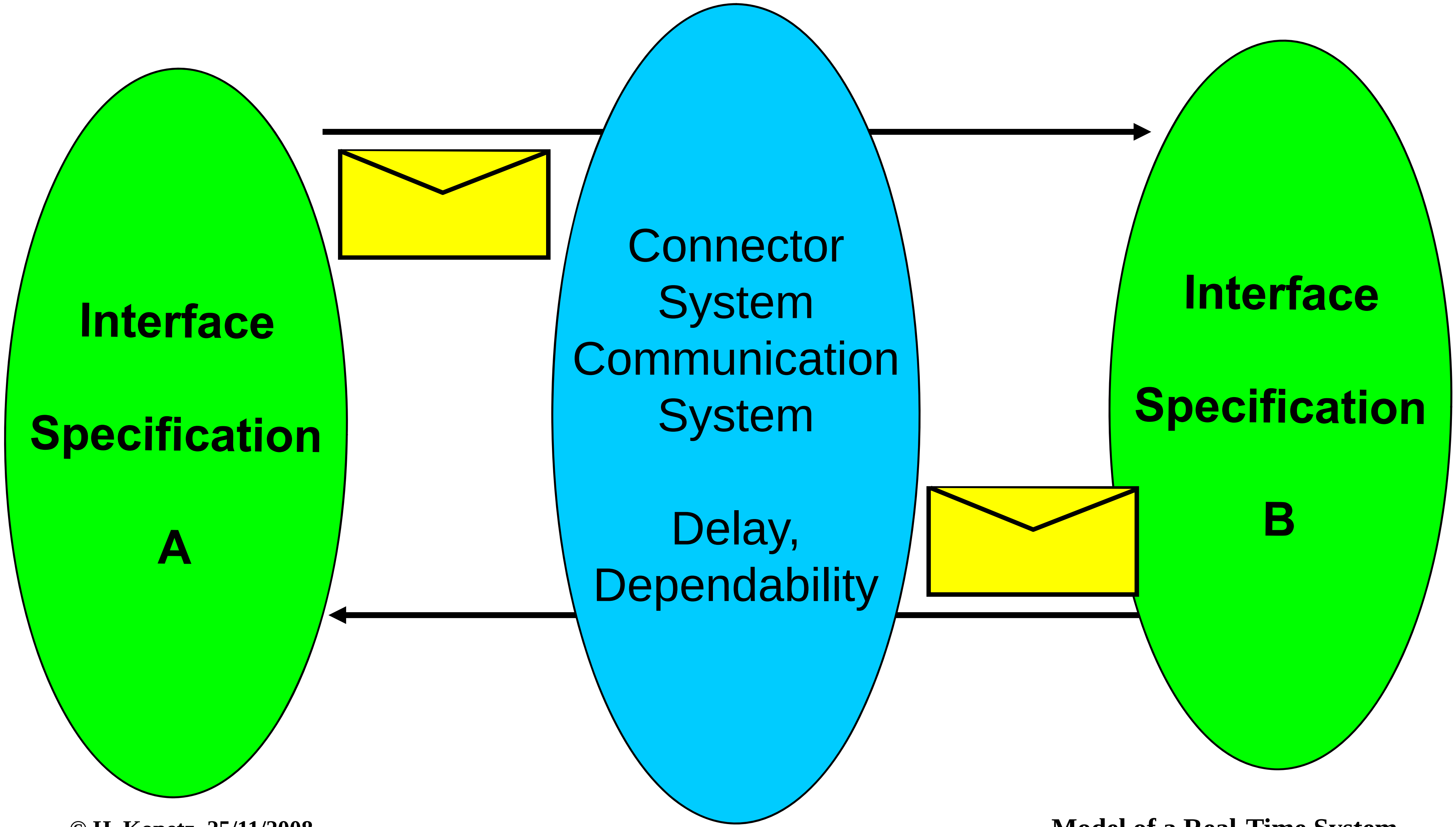
A Composition Involving three LIFs



Property Mismatches at Interfaces

Property	Example
Physical, Electrical Communication protocol	Line interface, plugs, CAN versus J1850
Syntactic	Endianness of data
Flow control	Implicit or explicit, Information push or pull
Incoherence in naming	Same name for different entities
Data representation representation	Different styles for data
	Different formats for date
Temporal	Different time bases Inconsistent time-outs
Dependability assumptions	Different failure mode
Semantics	Differences in the meaning of the data

Connector System to Resolve Property Mismatch

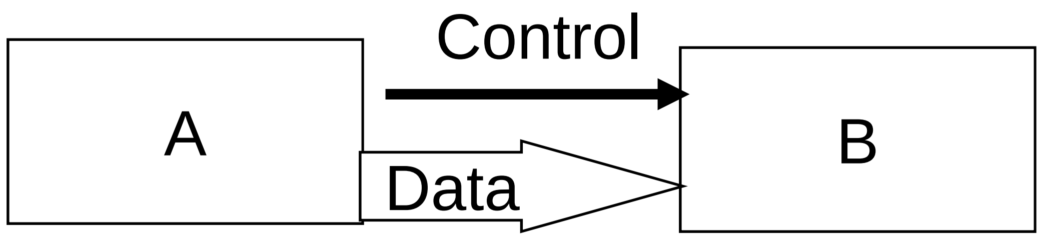


Elementary vs. Composite Interface

Consider a unidirectional data flow between two subsystems (e.g., data flow from sensor node to processing node).

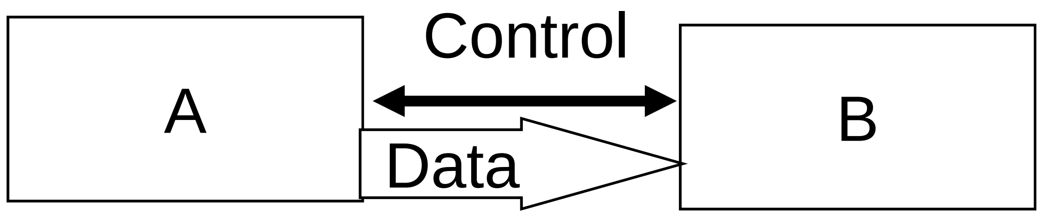
We distinguish between:

Elementary Interface:



Example: state message in a DPRAM

Composite Interface:



Queue of event message

Elementary interfaces are inherently simpler than composite interf

Information Push vs. Information Pull

Information Push Interface: Information producer pushes information on information consumer (e.g., telephone, interrupt)

Information Pull Interfaces: Information consumer requests information when required (e.g, email).

What is better in real-time systems?--For whom?

Sender: Information push

Receiver: Information pull

Example: Text to Speech

Input Interface:

Event-triggered acceptance of text according to the client-server paradigm

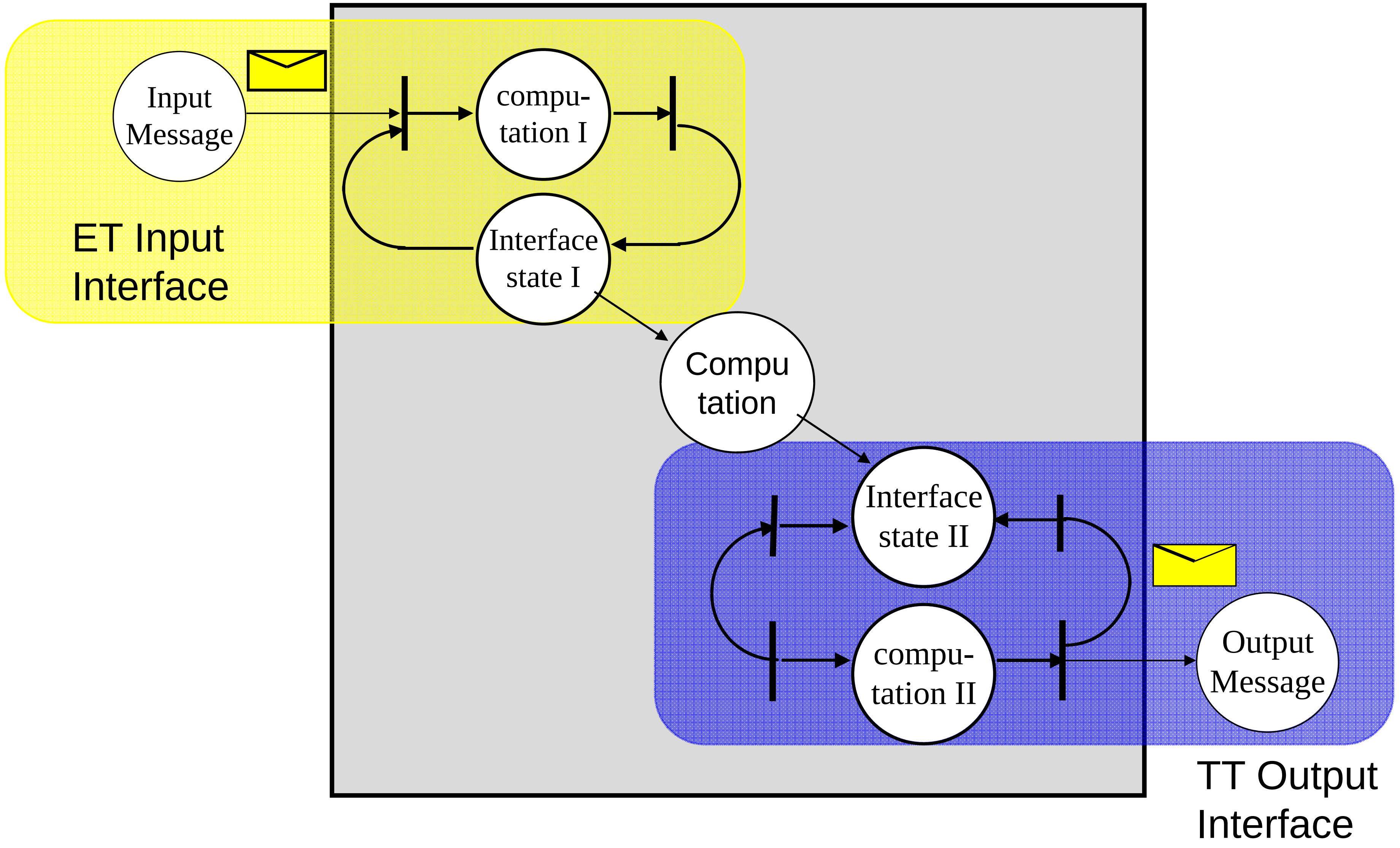
ET interface is composite

Output Interface:

Time-triggered output of a bit-stream that is encoding the sound. Temporal firewall interface.

TT interface is elementary

Interface State Machine: Text-to-Speech



Abstract Interface

Every interface belongs to two subsystems. The information crossing the interface may be coded differently within these two subsystems.

Viewed from a higher level of abstraction, these differences in the *information representation* are of no concern, as long as the semantic contents and the temporal properties of the information are maintained across the interface (abstract interface).

Issues of *information representation* are relevant at the low level interface between different subsystems only, but not within a properly designed subsystem, e.g., a cluster, where a information representation standard is chosen.

Message Interface versus World Interface

We call the concrete low level interface the *world interface* and the internal abstract message-based the *message interface* of a computational cluster.

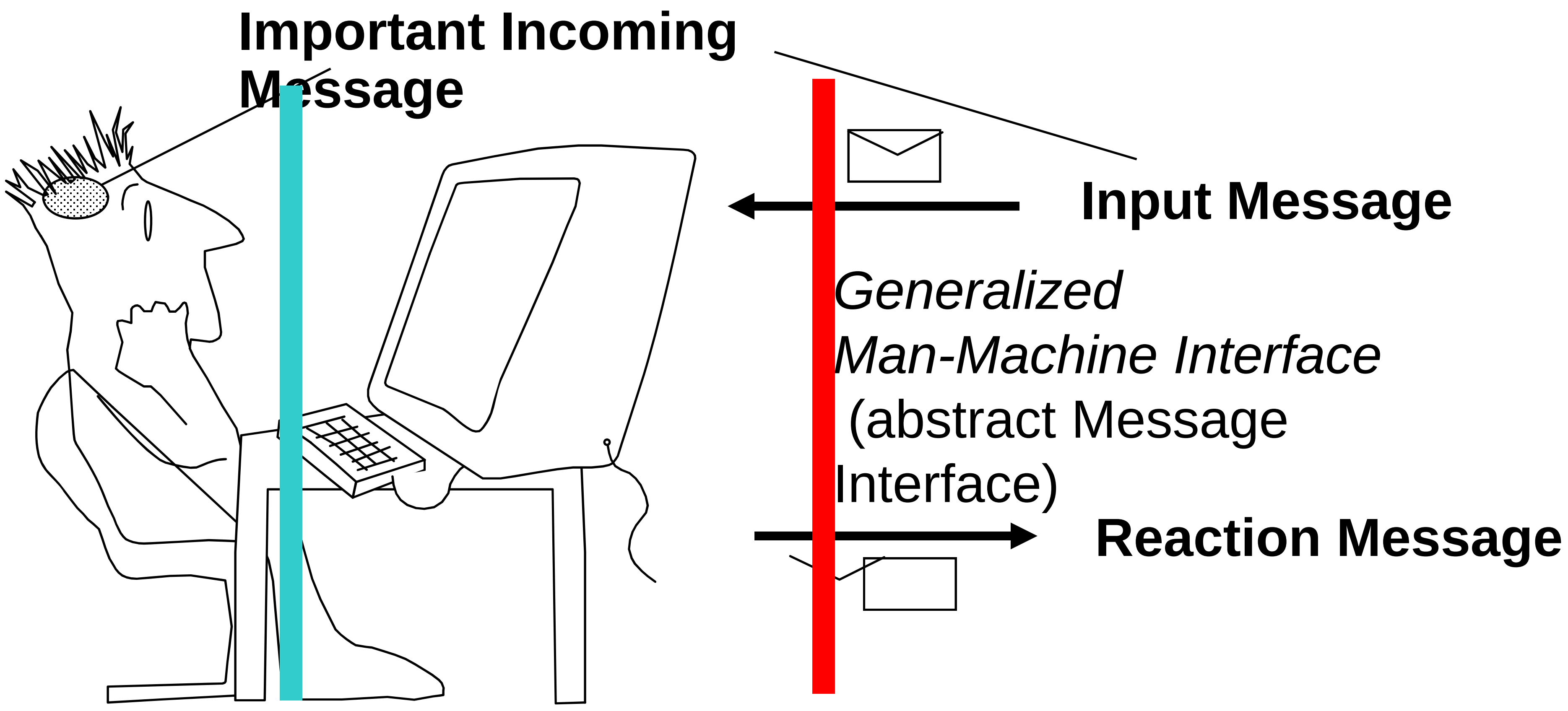
The interface component between the message and the world interface acts as an "information transducer*" and is called *resource controller*.

The resource controller hides the concrete physical interface of the real-world devices from the standardized information representation within a given cluster.

A resource controller is a kind of a gateway.

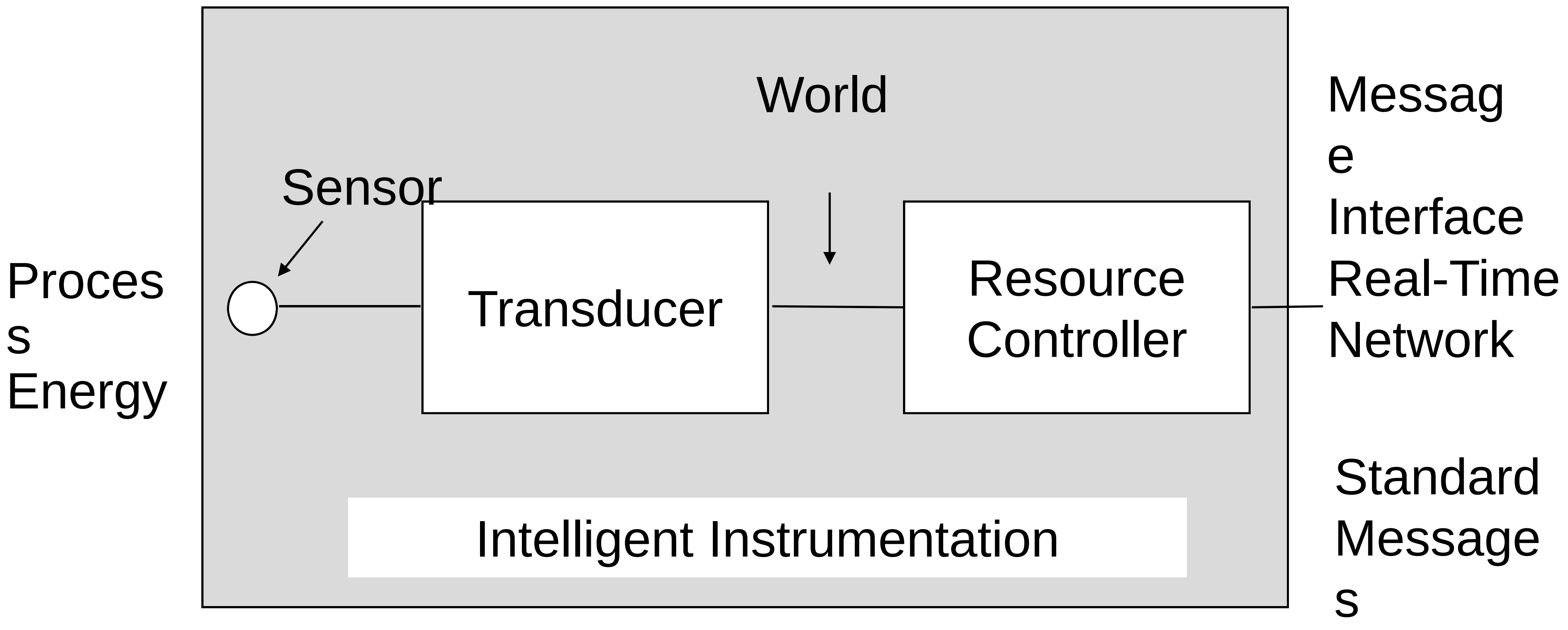
[*Transducer (Webster): a device that receives energy from one system, and retransmits it, often in a different form, to another].

Example: Man-Machine Interface

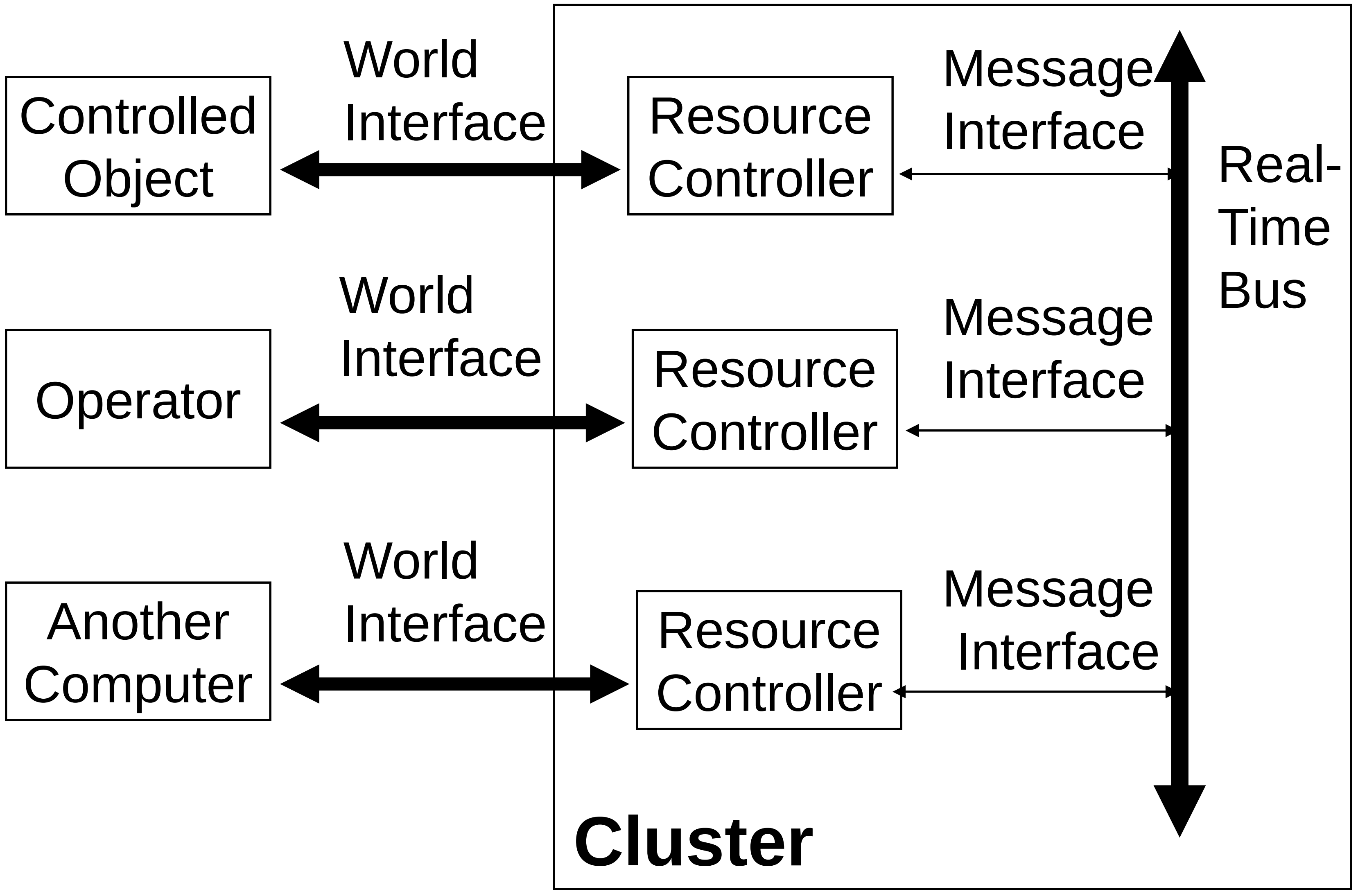

Important Incoming Message
Input Message
Generalized Man-Machine Interface
(abstract Message Interface)
Reaction Message

Specific Man-Machine Interface (View, Sound)
(concrete World Interface)

Example: Intelligent Instrumentation



World vs Message Interfaces



Comparison of World- and Message Interfaces

Characteristic	World Interface	Message Interface
Information Representation	unique	standardized
Coupling	tight	weaker
Coding	analog/digital	digital
Time Base	dense	sparse
Responsiveness	tight	weaker
Topology	1-to-1	multicast
Design Freedom	limited	given

SAE J 1587 Message Specification

The SAE has defined standard messages for heavy-duty vehicle applications (SAE J1587, p.20.374):

- ♦ Standardized Message IDs and Parameter IDs for many significant variables in the application domain
- ♦ Standardized data representation
- ♦ Definition of ranges of variables
- ♦ Update frequency
- ♦ Priority information

Message Classification

Attribute	Explanation	Antonym
valid	A message is *valid* if its checksum and contents are in agreement.	invalid
checked	A message is *checked at source* (or, in short, *checked*) if it passes the output assertion.	not checked
permitted	A message is *permitted* with respect to a receiver if it passes the input assertion of that receiver.	not permitted
timely	A message is *timely* if it is in agreement with the temporal specification	untimely
value-correct	A message is *value-correct* if it is in agreement with the value specification	not value-correct
correct	A message is *correct* if it is both timely and value-correct.	incorrect
insidious	A message is *insidious* if it is permitted but incorrect	not insidious

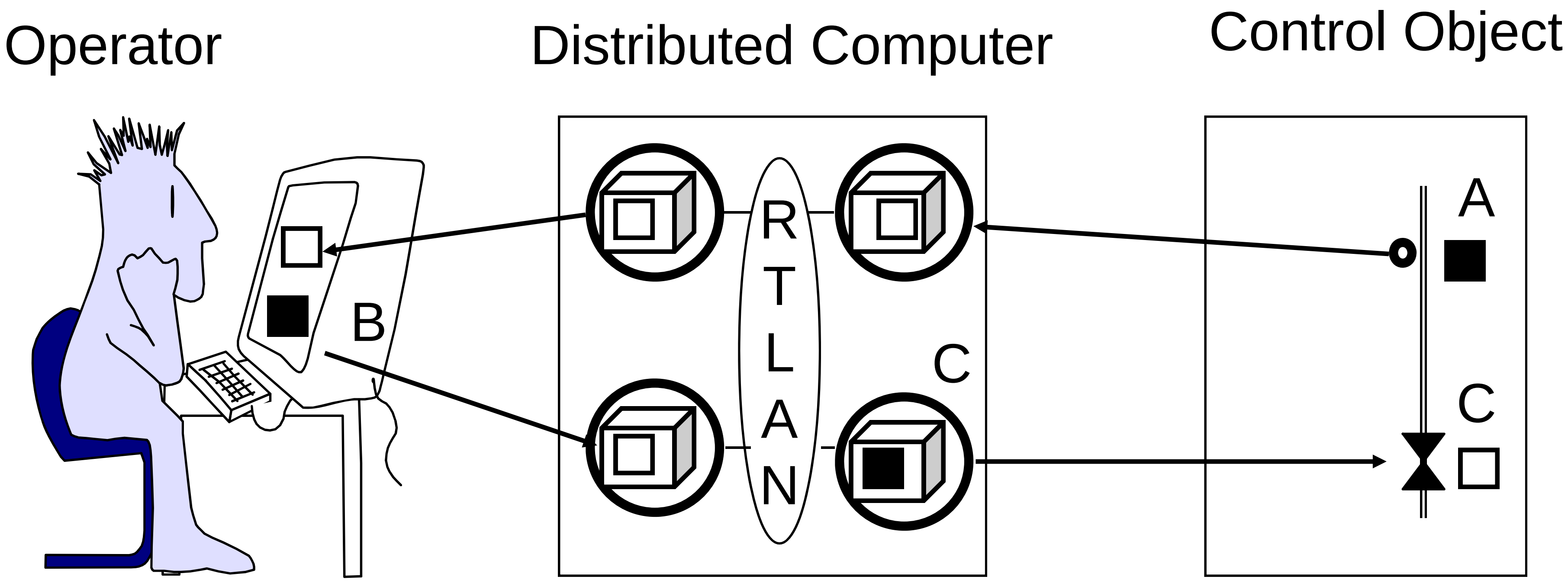
Architecture Design: Message Specification

During the architecture design the interactions among the components must be specified:

- ♦ Abstract message interface must be broken down to message data structures.
- ♦ Timing (periods, phases) of the messages
- ♦ Response time of the nodes
- ♦ Ground state of the nodes

The component design takes these message specifications as constraints.

RT Entities, RT Images and RT Objects



■ RT Entity   □ RT Image   RT Object

A: Measured Value of Flow

B: Setpoint for Flow   C: Intended Valve Position

Real Time (RT) Entity

A Real-Time (RT) Entity is a state variable of interest for the given purpose that changes its state as a function of real-time.

We distinguish between:

- ♦ Continuous RT Entities
- ♦ Discrete RT Entities

Examples of RT Entities:

- ♦ Flow in a Pipe (Continuous)
- ♦ Position of a Switch (Discrete)
- ♦ Setpoint selected by an Operator
- ♦ Intended Position of an Actuator

Attributes of RT-Entities

Static attributes of RT-Entity:

- ♦ Name
- ♦ Type
- ♦ Value Domain
- ♦ Maximum Rate of Change

Dynamic Attributes:

- ♦ Value set at a particular point in time

Sphere of Control

Every RT-Entity is in the Sphere of Control (SOC) of a subsystem that has the authority to set the value of the RT-entity:

- ♦ Setpoint is in the SOC of the operator
- ♦ Actual Flow is in the SOC of the control object
- ♦ Intended Valve Position is in the SOC of the Computer

Outside its SOC a RT-entity can only be observed, but not modified.

At this level of abstraction, changes in the representation of a RT-entity are not significant.

Observation

Information about the state of a RT-entity at a particular point in time is captured in an observation.

An observation is an atomic triple

Observation = <Name, Time, Value>

consisting of:

- ♦ The name of the RT-entity
- ♦ The point in real-time when the observation has been made
- ♦ The values of the RT-entity

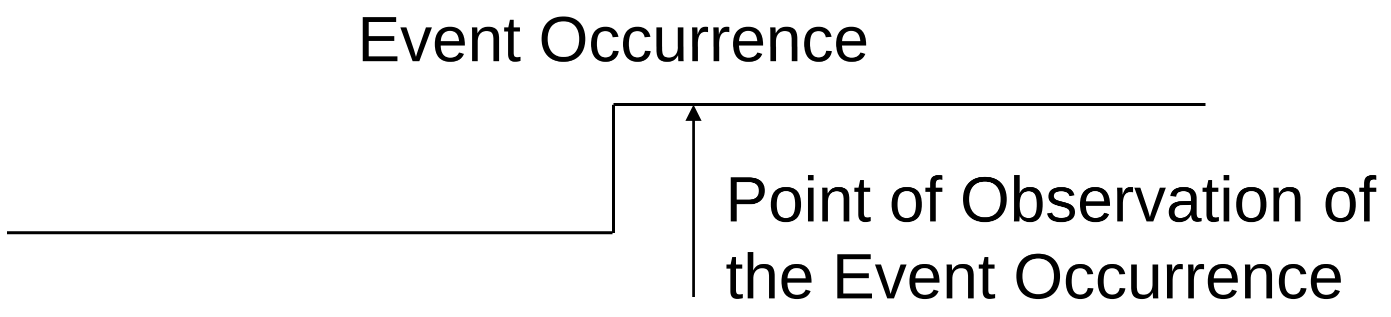
Observations are transported in messages.

Observation of Events

A state corresponds to a section of the timeline while an events occurs at the cut of the timeline.

Every change of state is an event.

An event cannot be observed--only the new state can be observed.



State and Event Observation

An observation is a *state observation*, if the value of the observation contains the full or partial state of the RT-entity. The time of a state observation denotes the point in time when the RT-entity was sampled.

An observation is an *event observation*, if the value of the observation contains the difference between the “old state” (the last observed state) and the “new state”. The time of the event information denotes the point in time of the L-event of the “new state”.

RT Images

A RT-Image is a picture of a RT Entity. A RT image is valid at a given point in time, if it is an accurate representation, both in the domains of value and time, of the corresponding RT Entity.

RT-Images

- ♦ are only valid during a specified interval of real-time.
- ♦ can be based on an observation or on a state estimation.
- ♦ can be stored in data objects, either inside a computer (RT object) or outside in an actuator.

RT Object

A RT-object is a “container” for a RT-Image or a RT-Entity in the Computer System.

A RT-object k

- ♦ has an associated real-time clock which ticks with a granularity t_k. This granularity must be in agreement with the dynamics of the RT-entity this object is to represent.
- ♦ Activates an object procedure if the time reaches a preset value.
- ♦ If there is no other way to activate an object procedure than by the periodic clock tick, we call the RT-object a *synchronous* RT object.

Distributed RT-Object

A distributed RT-object is 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.

The quality of service of a distributed RT-object must be in conformance with some specified consistency constraint.

Examples:

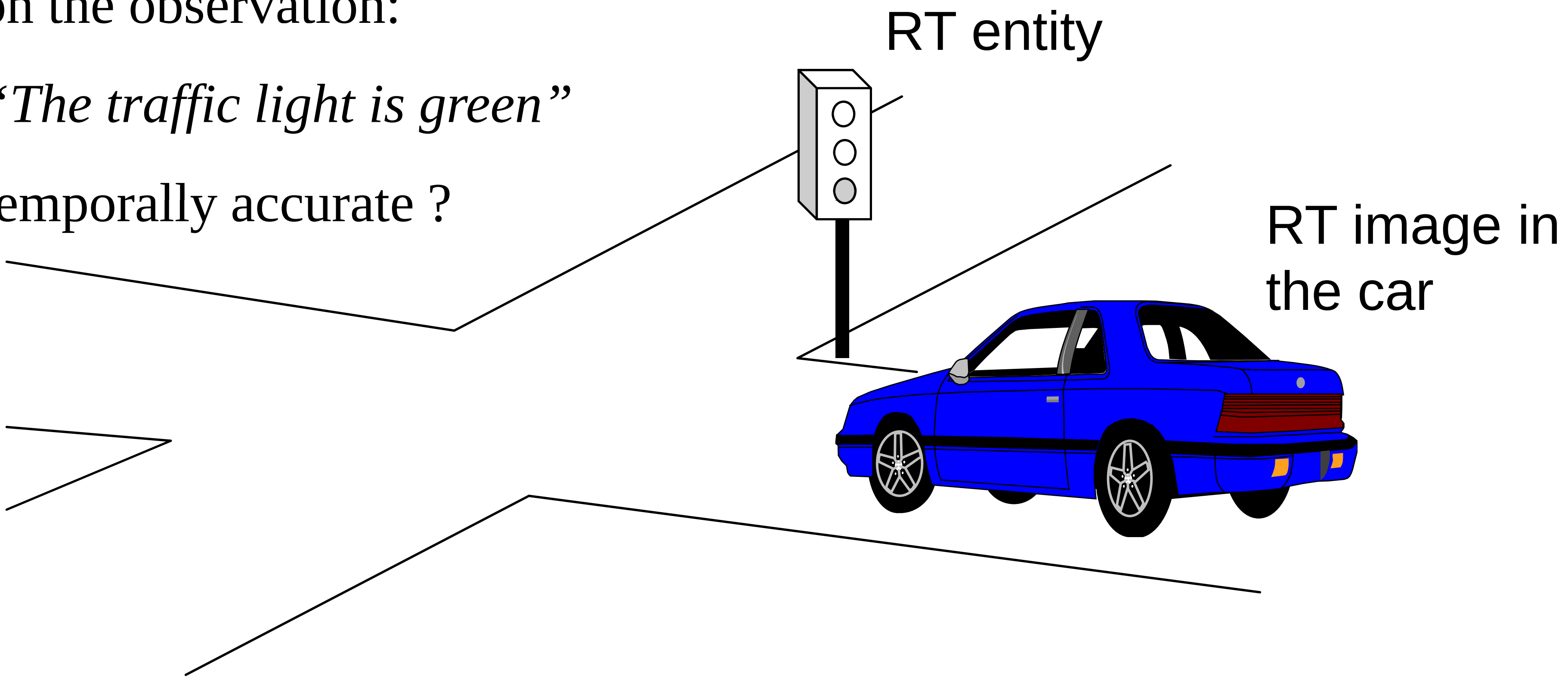
- ♦ Clock synchronization within a specified precision
- ♦ Membership service with a specified delay

Temporal Accuracy of Real-Time Information

How long is the RT image, based on the observation:

"The traffic light is green"

temporally accurate ?



Temporal Accuracy (II)

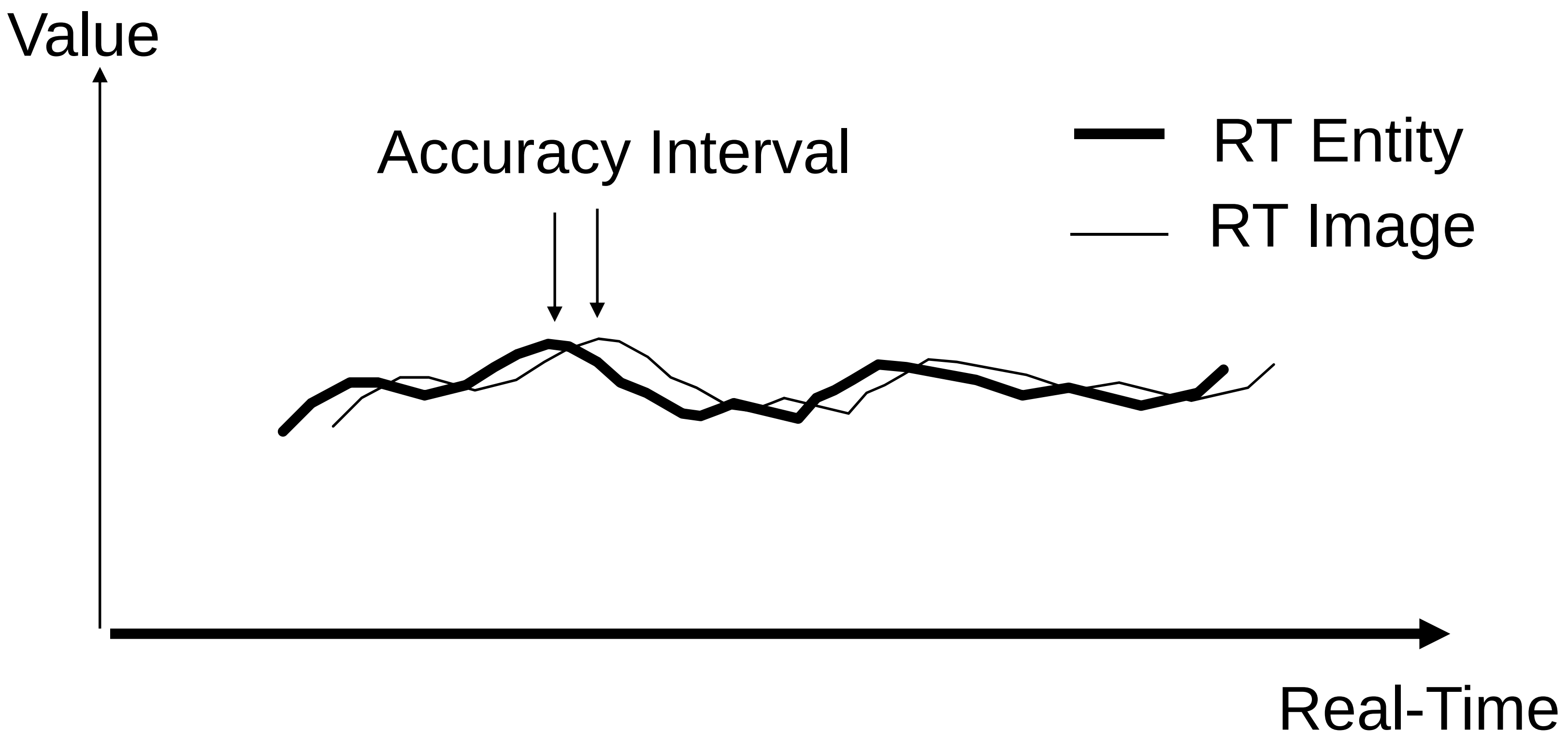
The temporal accuracy of a RT image is defined by referring to the recent history of observations of the related RT entity. A recent history RH_i at time t_i is an ordered set of time points $<t_i, t_{i-1}, t_{i-2}, \ldots\ t_{i-k}>$, where the length of the recent history

$$d_{acc} = t_i - t_{i-k}$$

is called the temporal accuracy. Assume that the RT entity has been observed at every time point of the recent history. A RT image is temporally accurate at the present time t_i

if $\exists t_j \in RH_i : Value(RT\ image\ at\ t_i) = Value(RT\ entity\ at\ t_j)$

Temporal Accuracy of RT Objects



If a RT-object is updated by observations, then there will always be
a delay between the state of the RT entity and that of the RT object

Point of Ignition

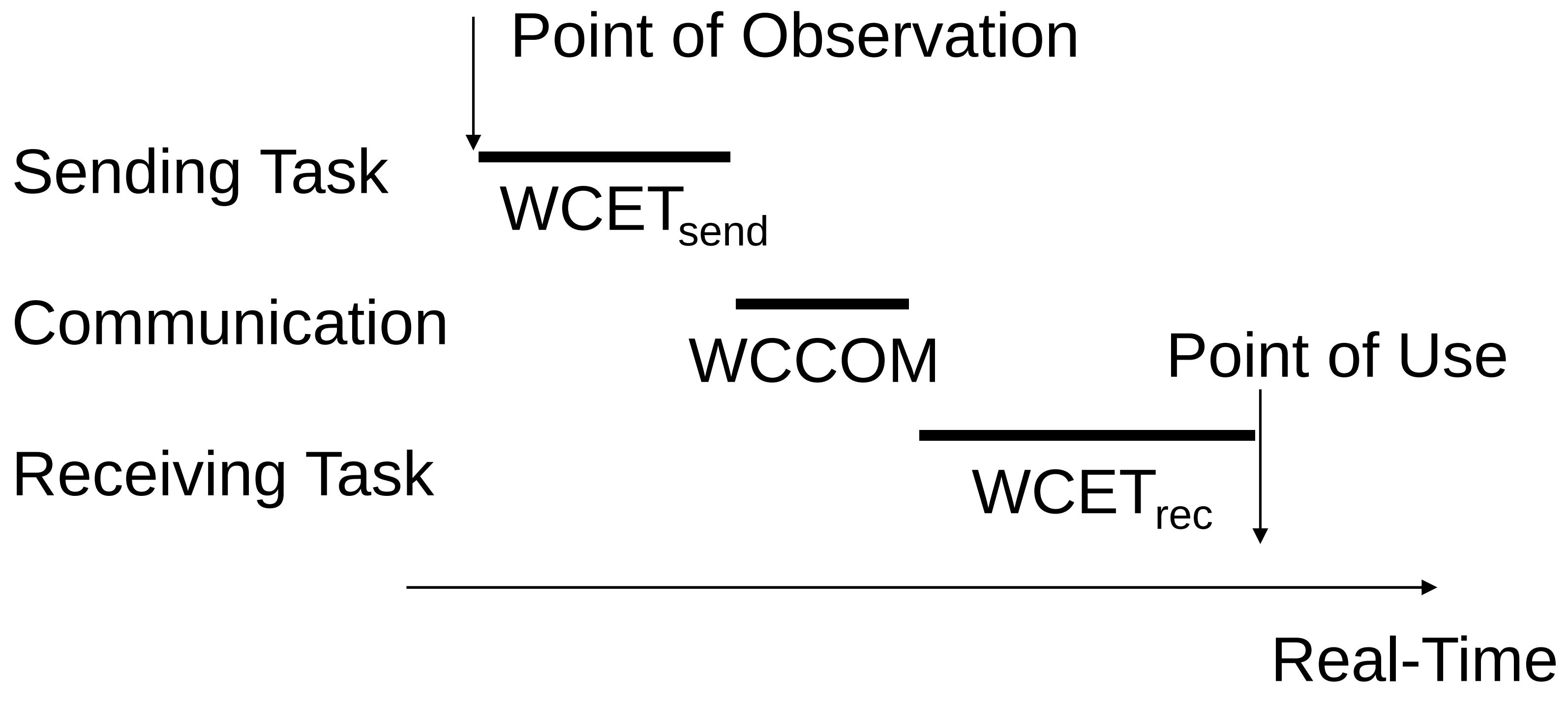
The point in time of ignition is a function of the following parameters:

Parameter	Recent History Accuracy(sec)
crank position	10^{-6}
h-state	10^{-3}
gas pedal position	10^{-2}
load	1
temperature	10^{+1}

There are seven orders of magnitude difference in the required temporal accuracy of the parameters.

There are five parameter, but only one can serve as control!

Synchronized Actions



State Estimation

A good accuracy of a RT-object can be obtained either by

- frequent sampling of the RT-entity or
- by state estimation

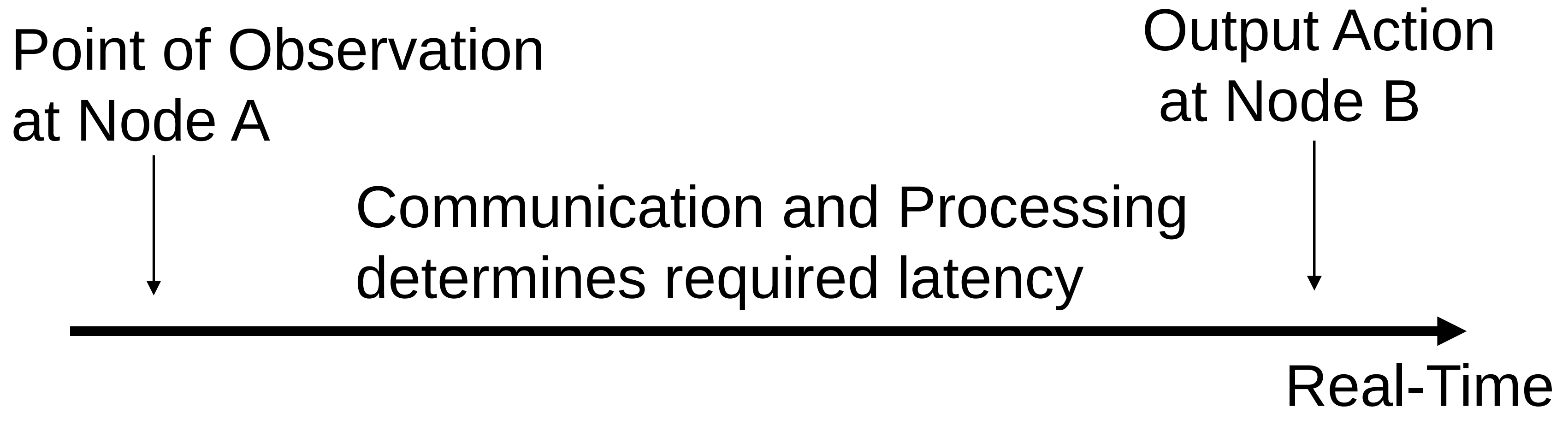
In state estimation, an estimation of the current state of the RT-entity is periodically calculated within a RT-object based on a computational model of the dynamics of the RT entity.

There is often the possibility for a tradeoff between computational and communication resources.

Timing Requirements for State Estimation

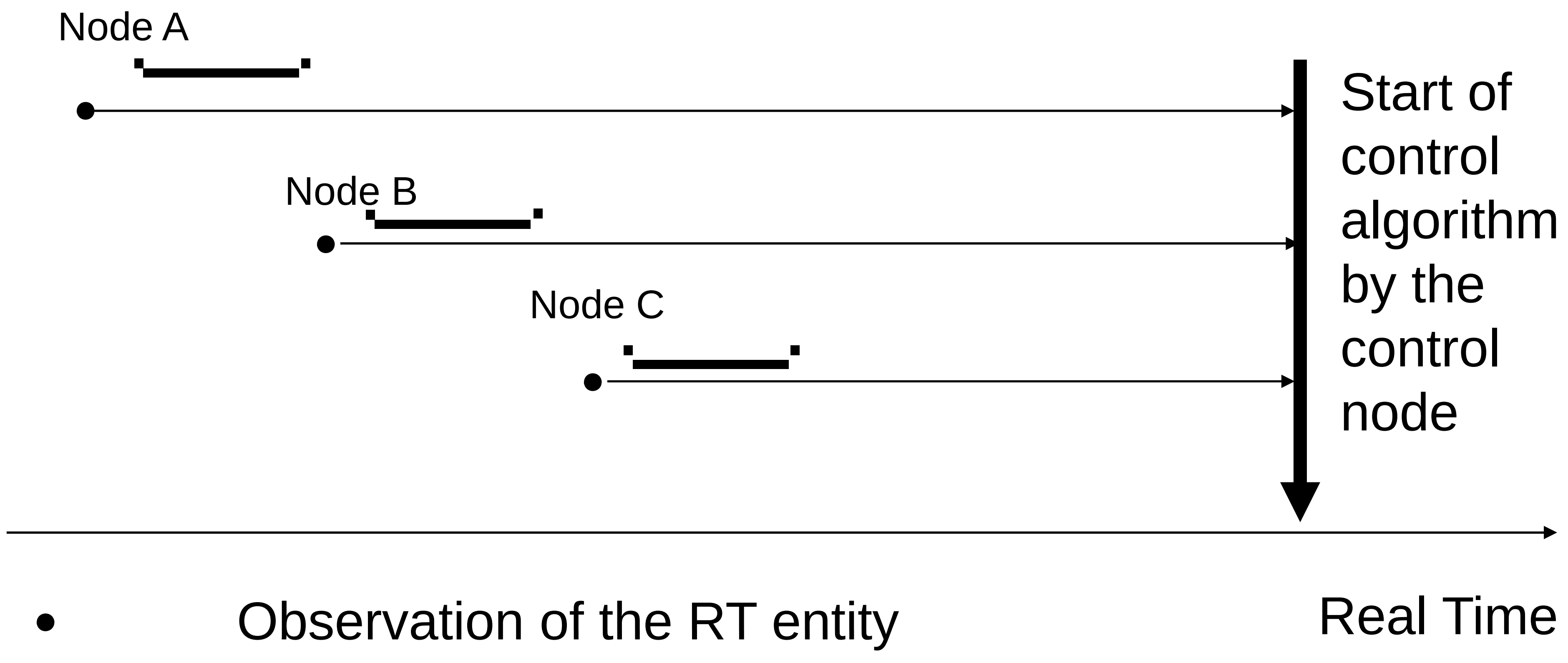
A state estimation program needs the time of observation of a RT-entity and the planned time of actuation to be able to compensate for the delay. The quality of state estimation depends on the

- ♦ Precision of the clock synchronization
- ♦ Size of latency and quality of latency measurement
- ♦ Quality of state estimation model



Point of Observation at Node A
Output Action at Node B
Communication and Processing determines required latency
Real-Time

State Estimation of Sensor Observations



● Observation of the RT entity

▪▬▪ Channel access interval

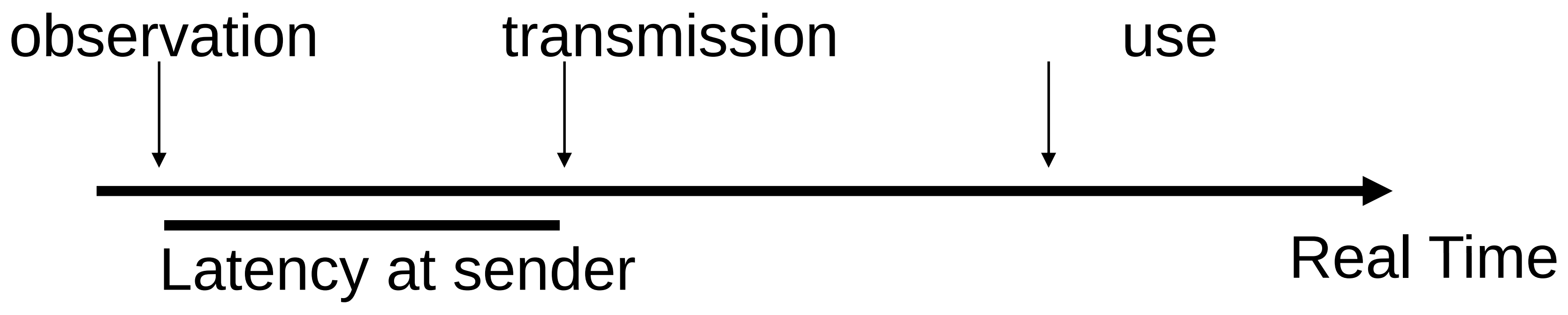
●⟶ Interval used for state estimation

Latency Guarantee

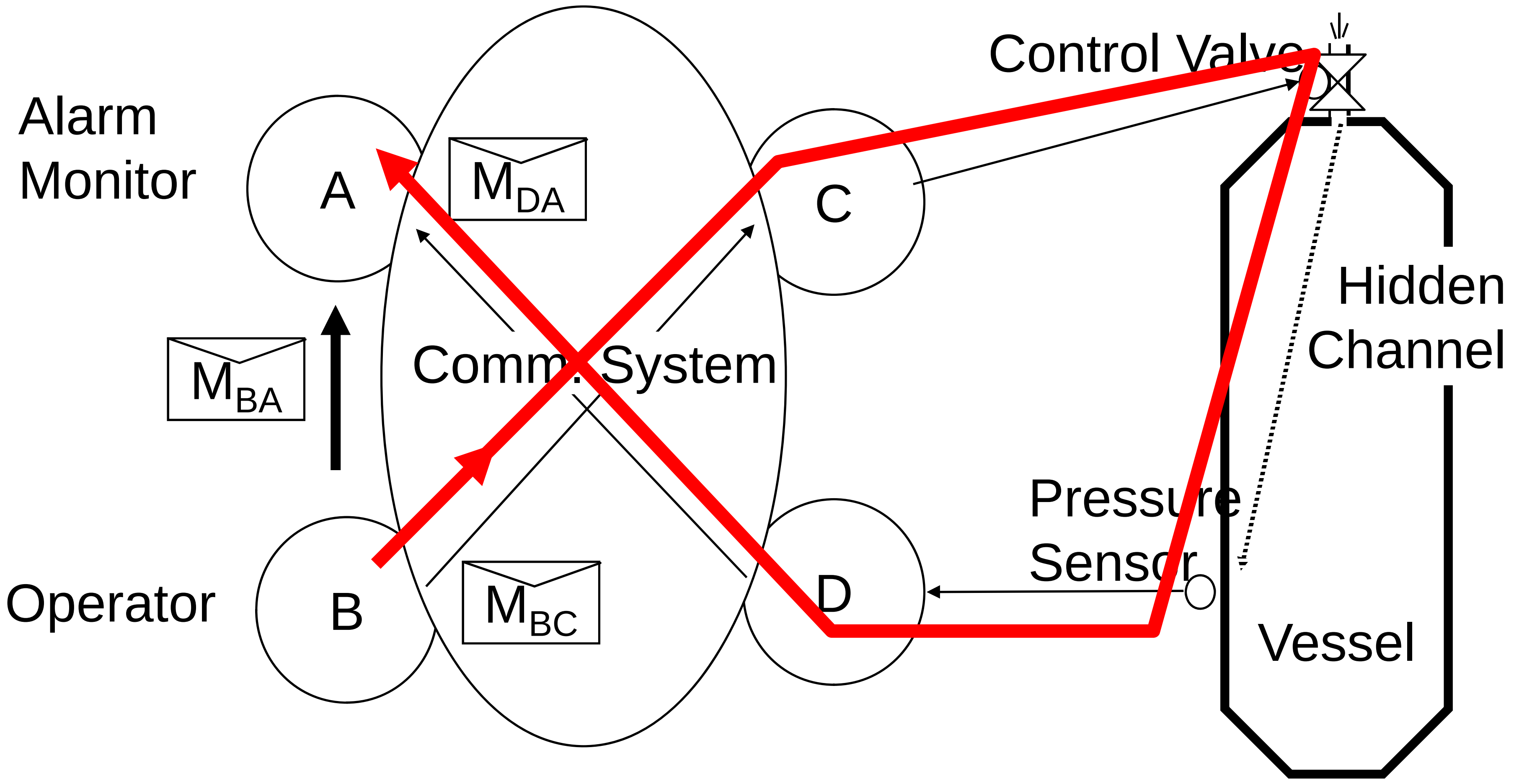
The composability of a distributed TT architecture can be improved, if the sender guarantees the latency between the point of sampling and the point of transmission.

Then the receiver can correct this known latency in his state estimation model.

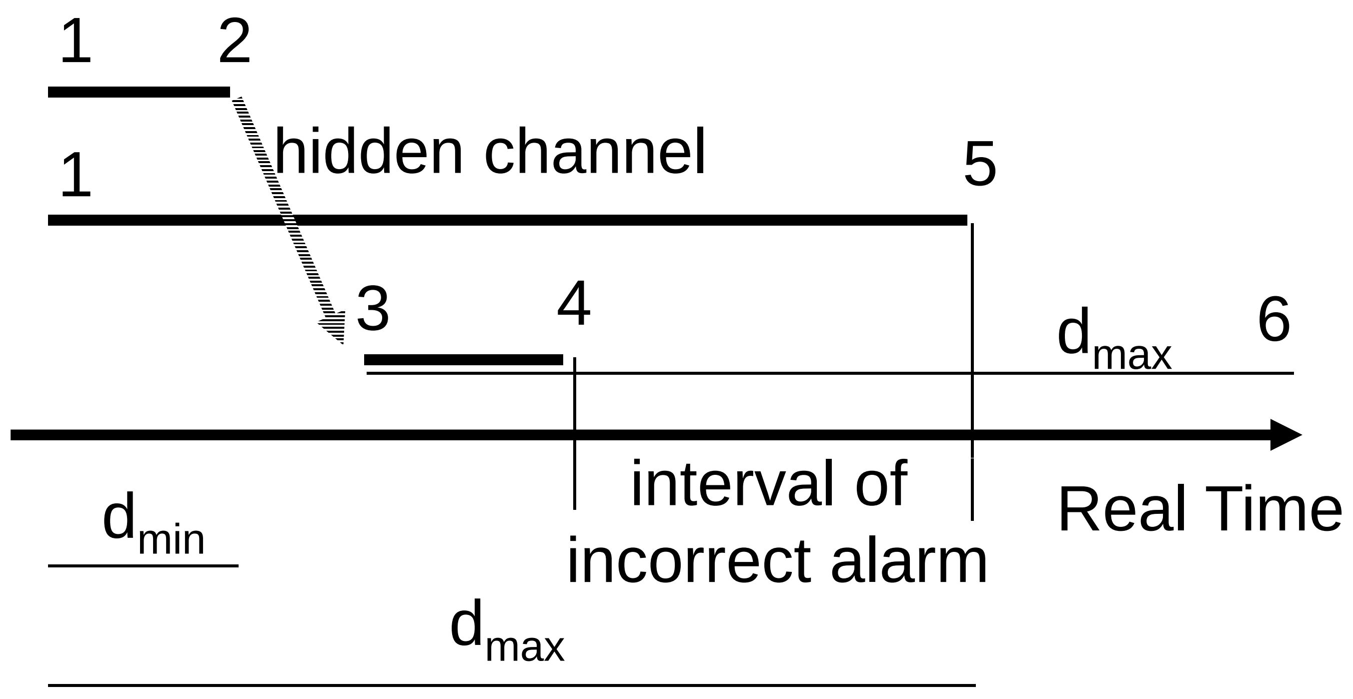
Alternatively, the sender can send “timed” messages that contain the interval between observation and transmission



Hidden Channel (red)



Hidden Channel (2)



1 Sending of M_{BC}

Sending of M_{BA}

2 Arrival of M_{BC}

3 Sending of M_{DA}

4 Arrival of M_{DA}

5 Arrival of M_{BA}

6 Permanence of M_{DA}

Permanence

Permanence is a relation between a given message M_i that has arrived at a RT-object O and all messages M_{i-1}, M_{i-2}, . . . that have been sent to this object before (in the temporal order) message M_i.

The message M_i becomes *permanent* at object O as soon as all previously sent messages have arrived at O.

If actions are taken on non-permanent messages, then an inconsistent behavior may result.

The *action delay* is the interval between the point in time when a message is sent by the sender and the point in time when the receiver knows that the message is permanent.

How long does it take until a message becomes permanent?

Action Delay

In distributed RT systems without a global time base the

maximum action delay: $d_{max} + \varepsilon = 2\, d_{max} - d_{min}$

but the consistent order problem is not yet solved!

In systems with a global time the maximum action delay: $d_{max} + 2g$

In distributed real time system the maximum protocol execution time and not the “median” protocol execution time determines the responsiveness.

Accuracy versus Action Delay

In a properly designed RT system

Action Delay < d_{acc}

The accuracy is an application specific parameter, while the action delay is an implementation specific parameter.

What happens if this condition is violated?

Then we need state estimation!

Idempotence

Idempotence is a relation between a set of messages.

A set of messages is *idempotent*, if the effect of receiving more than one messages of this set is the same as the effect of receiving a single message.

Duplicated state messages are idempotent.

Duplicated event messages are not idempotent.

If the idempotence of a set of redundant messages is given, then the design of fault-tolerant systems is simplified.

Definition of *Determinism*: First Try

First definition of *determinism*:

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.

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

Determinism: Second Try

We therefore need a revised, more realistic definition of determinism in a distributed real-time computer system that takes account of the fact that clocks cannot be fully synchronized. 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.*

Replica Determinism

Replica Determinism is a desired relation between replicated RT objects.

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.

In a Time-Triggered System, this time interval is determined by the precision of the clock synchronization.

Replica determinism is needed for

- ♦ consistent distributed actions
- ♦ the implementation of fault tolerance by active redundancy

Why do we need “Replica Determinism”?

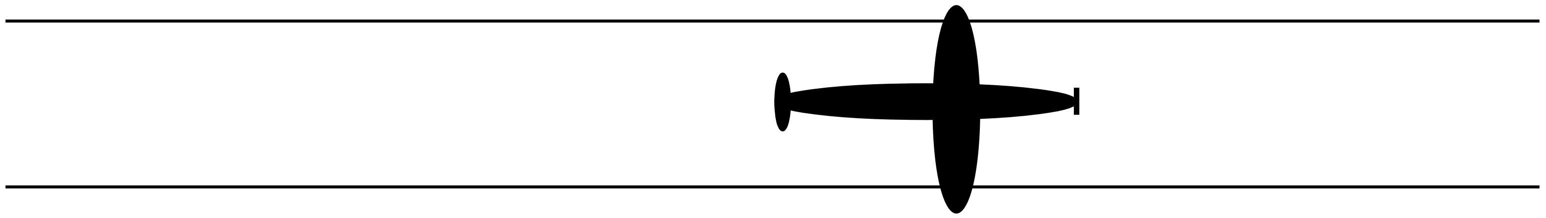
Replica Determinism is needed for two main reasons:

- ♦ 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!

Example: Airplane on Takeoff

Consider an airplane with a three channel flight control system taking off from a runway:



Channel 1	Take off	Accelerate Engine
Channel 2	Abort	Stop Engine
Channel 3	Take off	*Stop Engine (Fault)*
Majority	*Take off*	

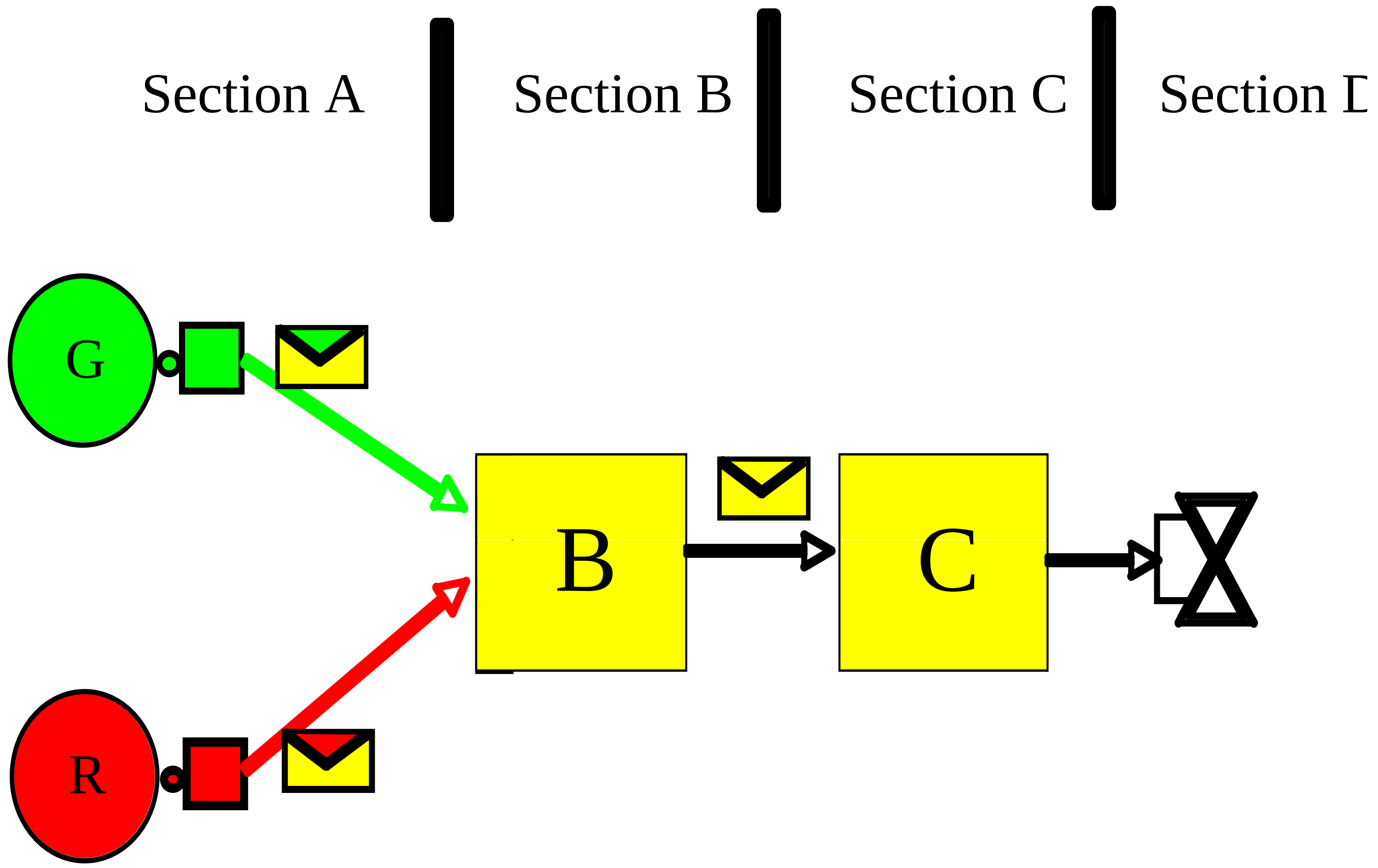
Determinism of a Communication Channel

A communication channel is called *deterministic* if (as seen from an omniscient external observer):

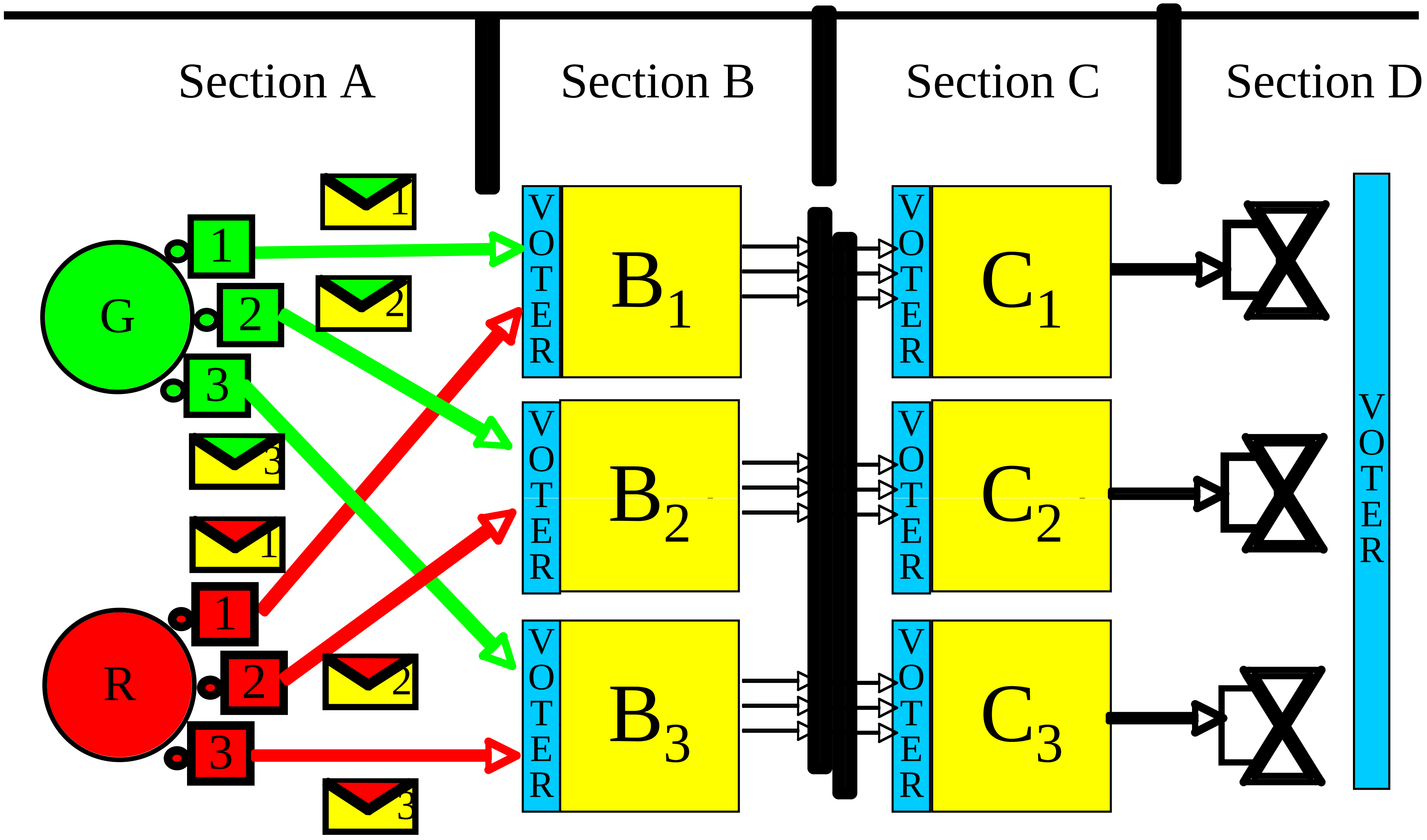
- ♦ The *receive order* of the messages is the same as the *send order.* The *send order* among all messages is established by the *temporal order* of the *send instants* of the messages as observed by an omniscient observer.
- ♦ If the *send instants* of *n (n>1)* messages are the *same,* then an order of the *n* messages will be established in an *a priori* known manner.

Two *independent* deterministic channels will deliver messages *always* in the same order.

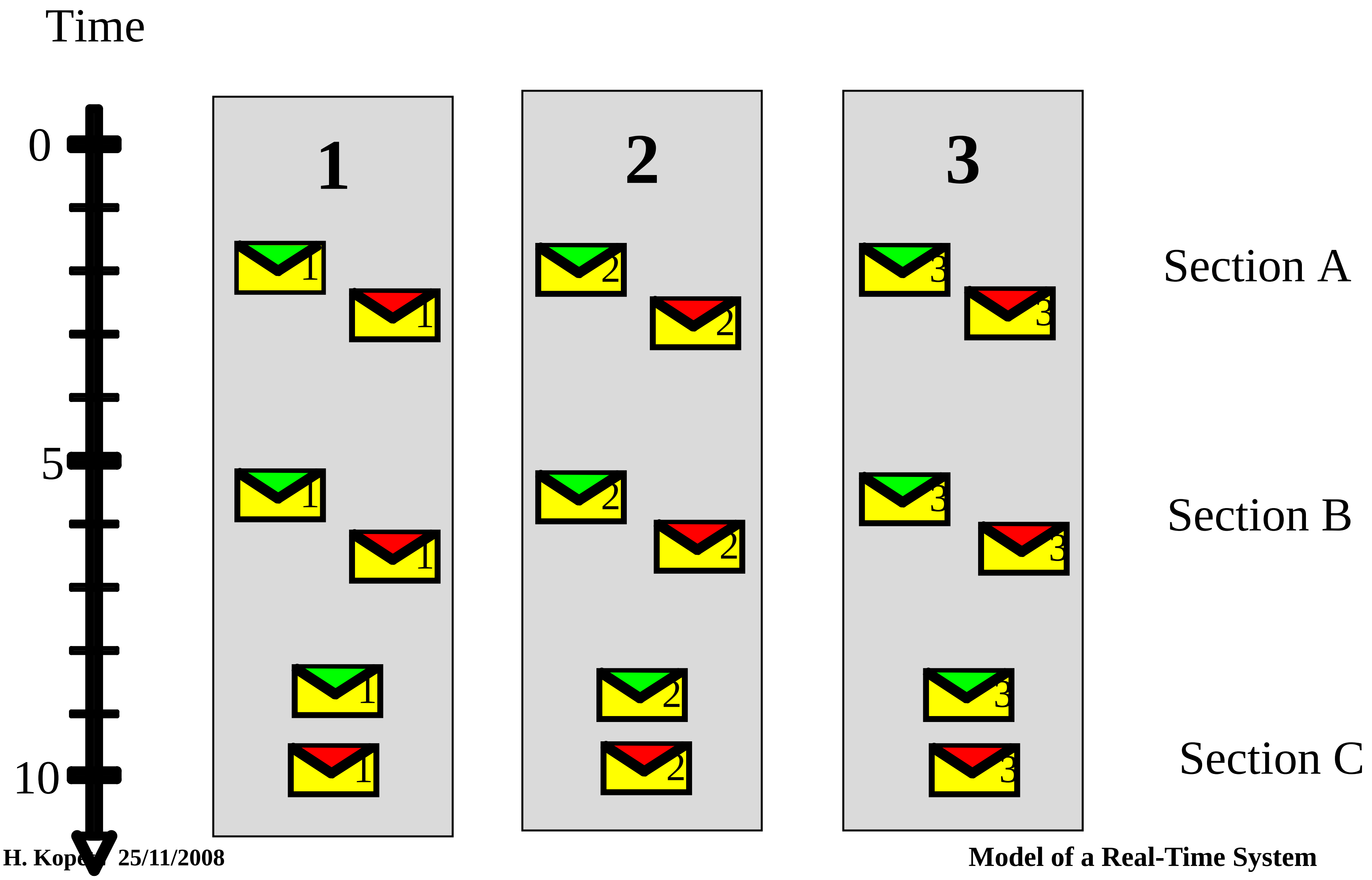
Non-Redundant Application



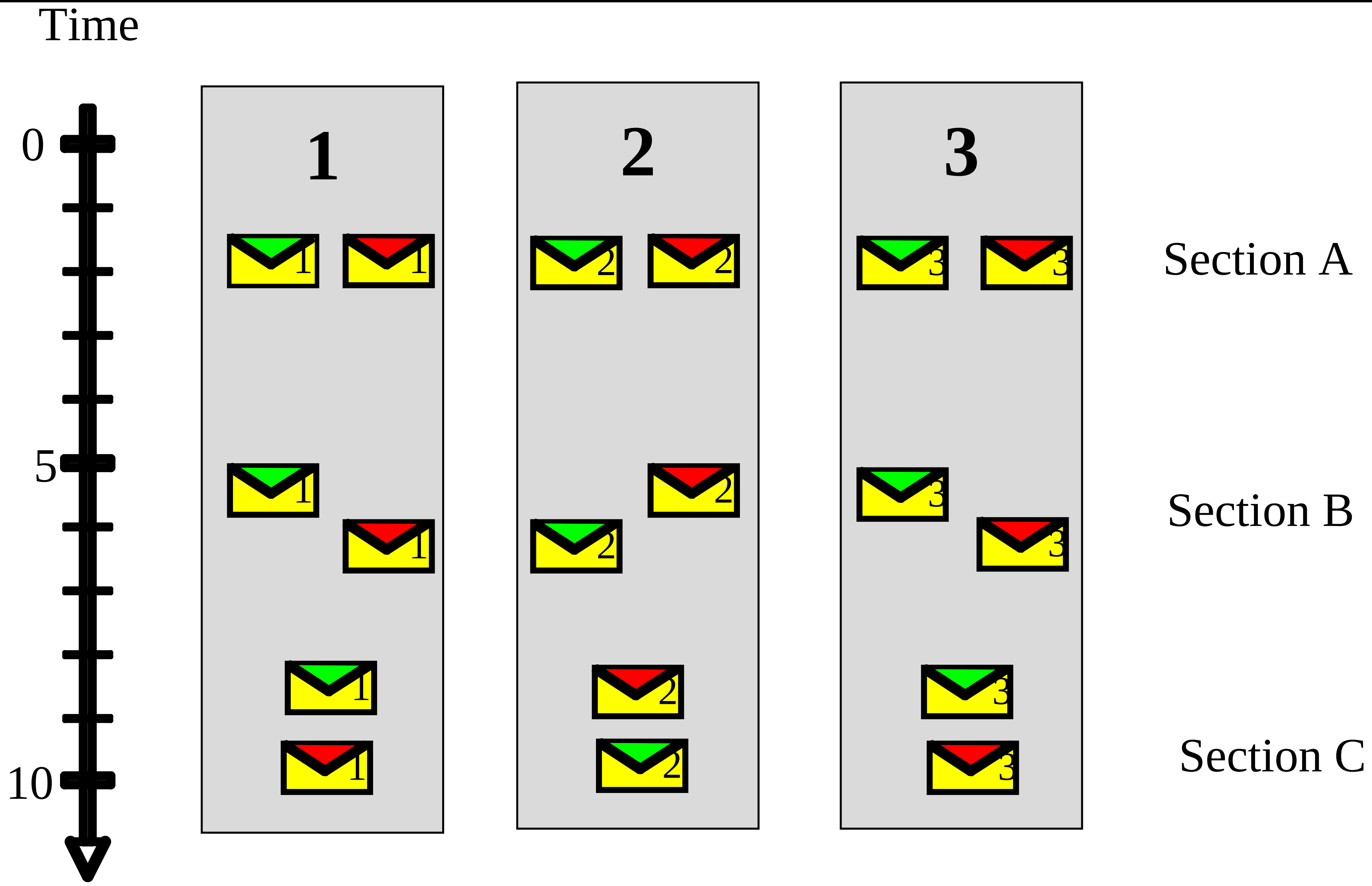
TMR Configuration



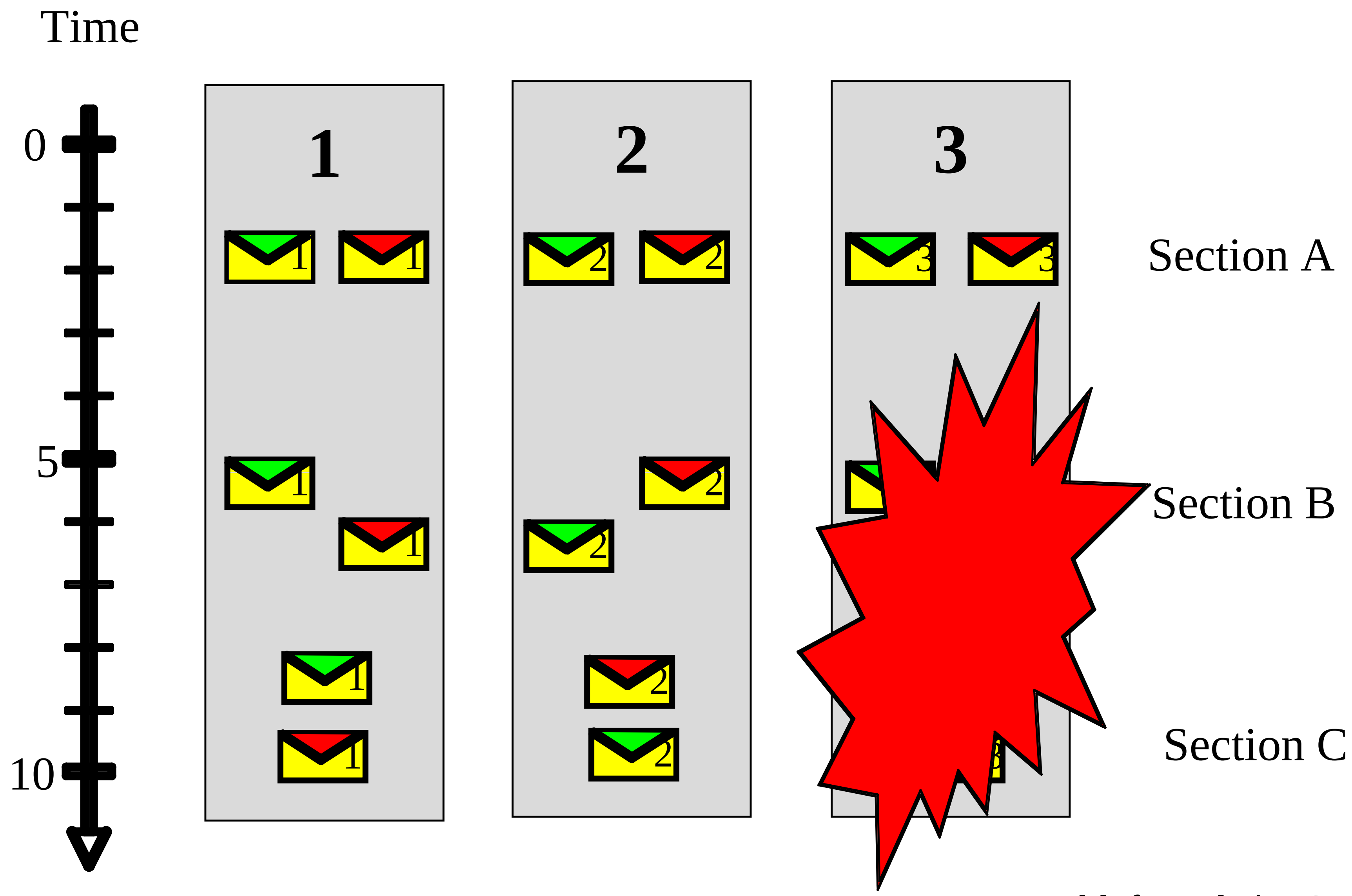
Order of Messages



What Happens in Case of Simultaneity



Loss of Fault-Masking Capability



Simultaneity: A Fundamental Problem

The ordering of simultaneous events is a fundamental problem of computer science:

- Hardware level: metastability
- Node level: semaphor operation
- Distributed system: ordering of messages

There are two solutions *within* a distributed system to solve the simultaneity problem:

- Distributed consensus--takes real-time and requires bandwidth (atomic broadcast)
- Sparse time

What destroys replica determinism?

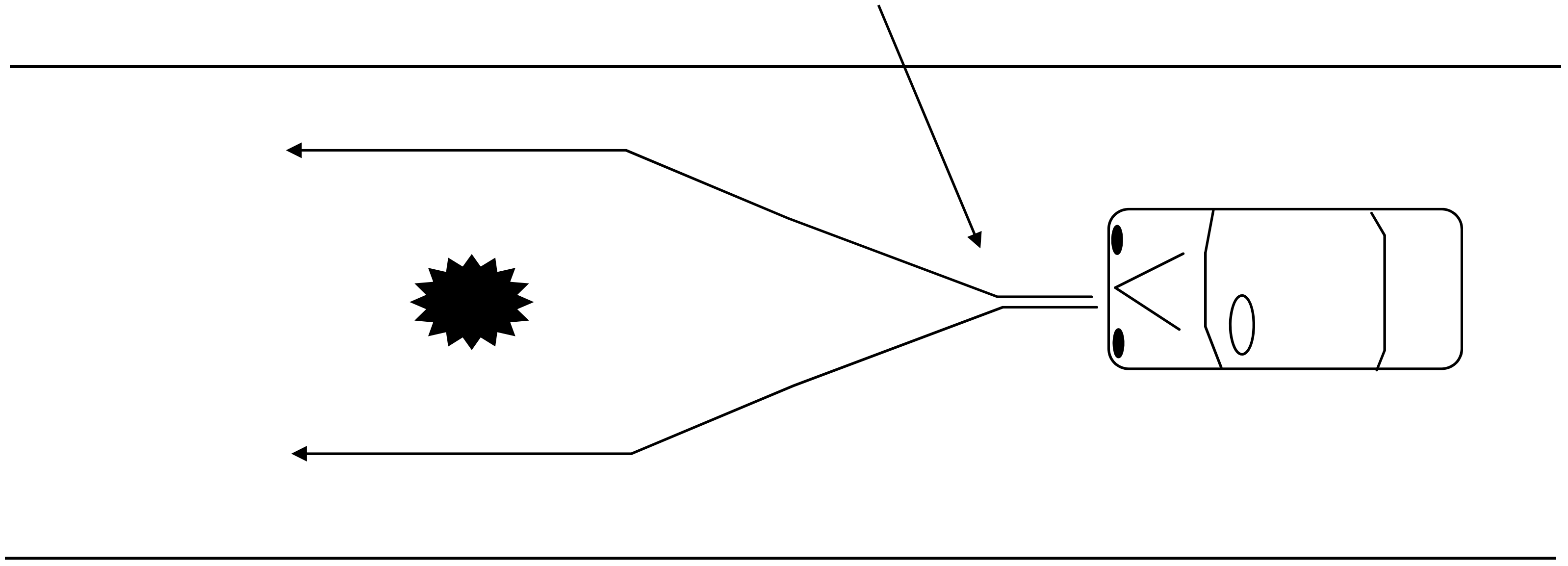
Replica determinism can be destroyed by:

- ♦ Inconsistent inputs
- ♦ Inconsistent order
- ♦ Non-deterministic program constructs
- ♦ Explicit synchronization statements (e.g., *Wait*)
- ♦ Uncontrolled access to the global time and timeouts
- ♦ Dynamic preemptive scheduling decisions
- ♦ Consistent comparison problem (software diversity)

This list is not complete!

Major Decision Point

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



Basic Causes of Replica Indeterminism:

At the root of replica indeterminism are:

- ♦ Slightly differing processing speeds, caused by differing crystal resonators (clocks)
- ♦ Finite representation of real values (e.g., real arithmetic)
- ♦ Differing inputs, caused by inconsistent order or sensor variations

Solutions:

- ♦ Sparse value (Time) Base, if possible
- ♦ Static or non-preemptive scheduling
- ♦ exact arithmetic
- ♦ agreement on input data and order