ARTIST2 Summer School 2008 in Europe
Autrans (near Grenoble), France
September 8-12, 2008

Implementation of Control Systems in Resource-Constrained Embedded Systems

Lecturers: Karl-Erik Årzén & Pedro Albertos

Karl-Erik Årzén	Pedro Albertos
Professor	Professor
Lund University	UPVLC

Outline

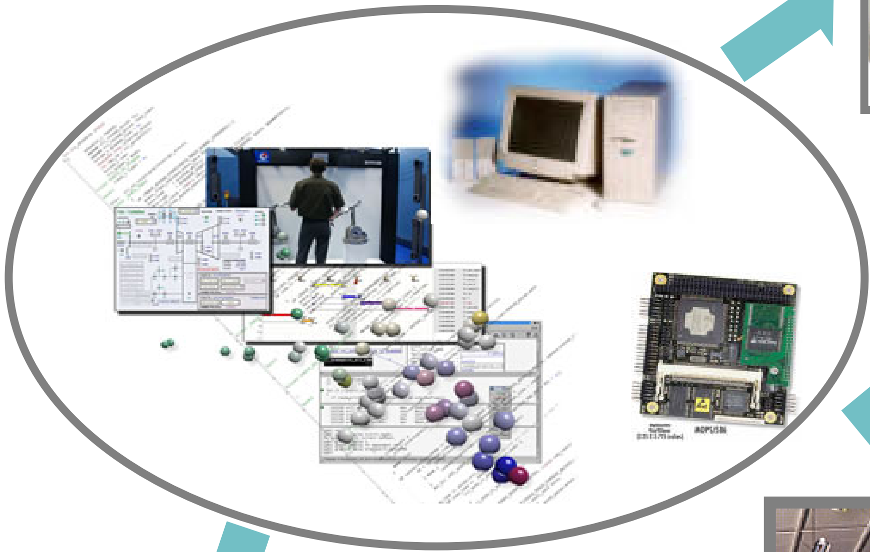
- Ir
- T
- F
- E
- F
- C

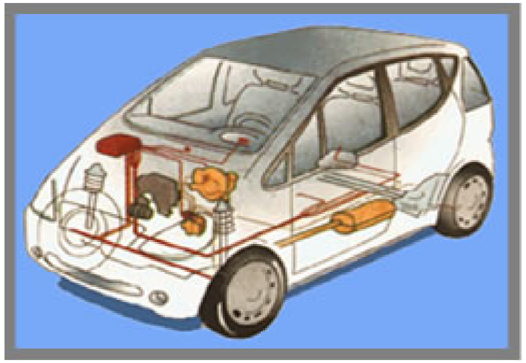
Introduction

Why Control in Artist2?

- Important application area for RT embedded systems
  - Many embedded systems are control systems
  - Motivation for real-time systems
- Embedded control systems have special characteristics and challenges
- Control is a basic technology for managing uncertainty that also can be used to generate robustness and performance also in embedded systems

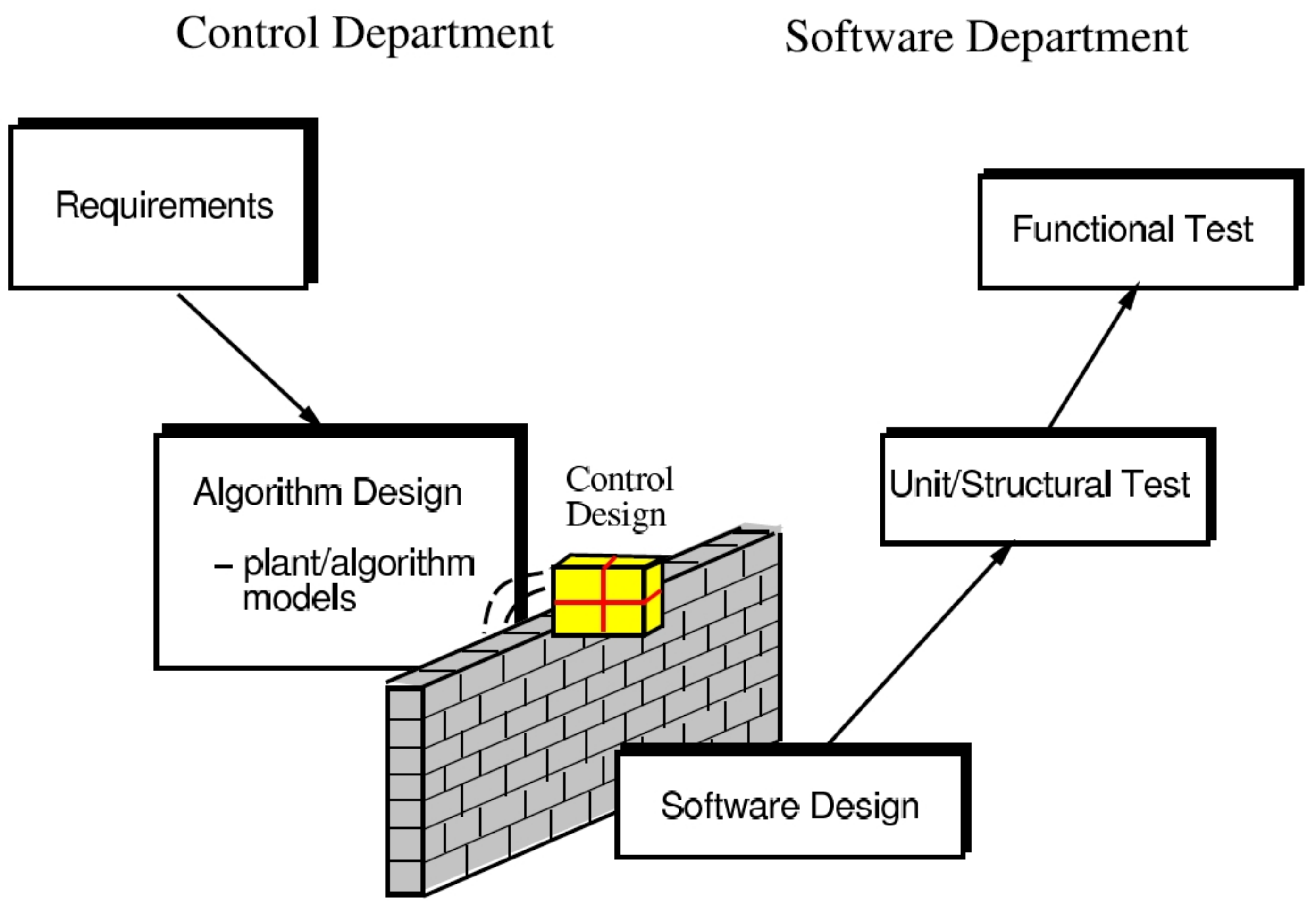
Embedded systems





Control is present in most embedded applications

Industrial Control System Development

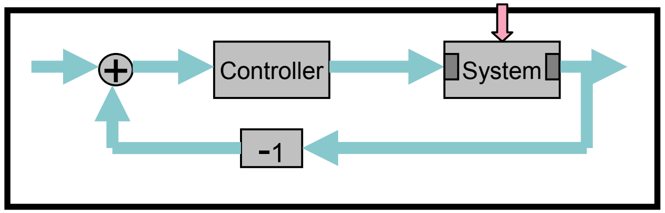


Problems

- The control engineer does not care about the implementation
  - **“trivial”**
  - **“buy a fast computer”**
- The software engineer does not understand controller timing
  - **“τ_i = (T_i, D_i, C_i)”**
  - **“hard deadlines”**
- Little mutual understanding

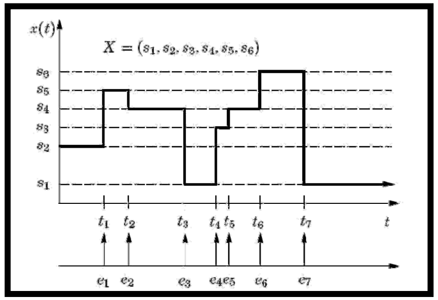
Different Types of Control

Continuous



- Time-driven
- Deterministic (periodic) sampling
- Jitter and latency

Discrete



- Event-driven
- Reactive
- Finite state machines

Hybrid Control

Embedded Control Characteristics

What distinguishes embedded control?

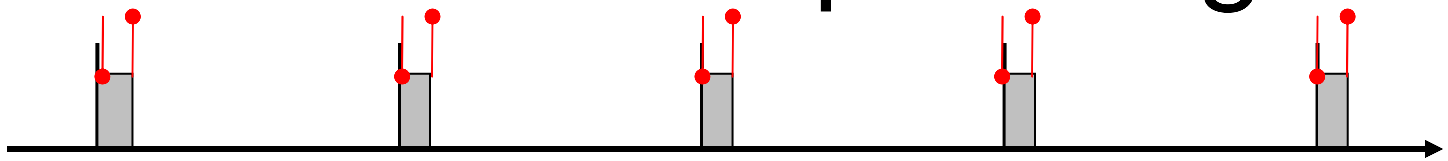
– **Limited computing and communication resources**
  - CPU time, communication bandwidth, memory, energy, …

– **Autonomous operation**
  - No human ”operator”
  - Complex functionality
  - Often large amounts of software
  - Need for formal approaches
  - Need for design methodology

Embedded Control Characteristics

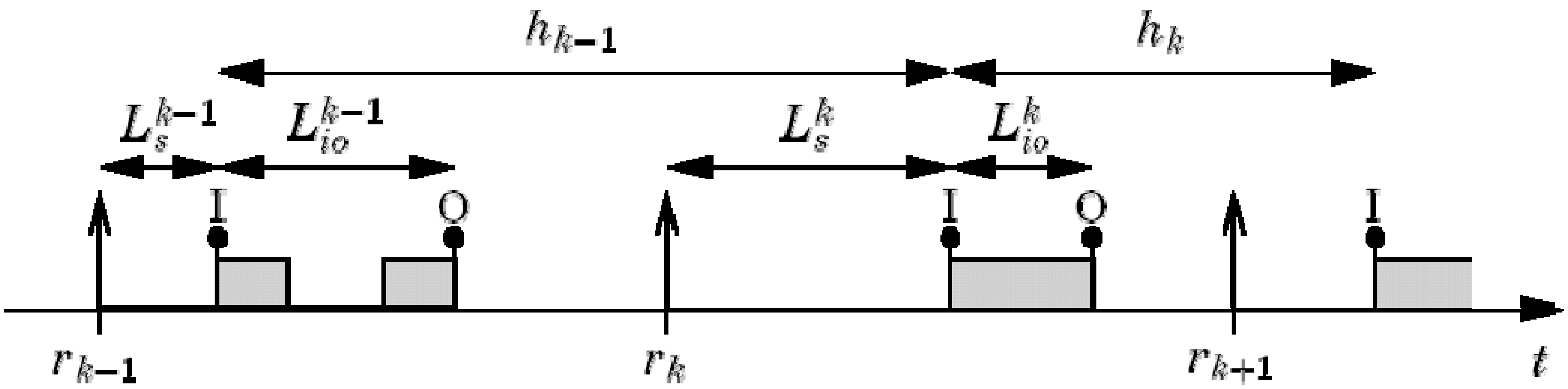
- Limited resources → Efficiency
  - Code-size efficient
  - Run-time efficient
  - Energy efficient
  - Size and weight efficient
  - Cost efficient
- Autonomous operation → Dependability
  - Reliability
  - Maintainability
  - Availability
  - Safety
  - Security

Control Loop Timing



- Classical control normally assumes periodic sampling
  - too long sampling interval or too much jitter give poor performance or instability
- Classical control assumes negligible or constant input-output latency (from sampling to actuation)
  - if the latency is small compared to the sampling interval it can be ignored
  - if the latency is constant it can be included in the control design
  - too long latency or too much latency jitter give poor performance or instability
- Not always possible to achieve with limited computing resources that are shared with other applications

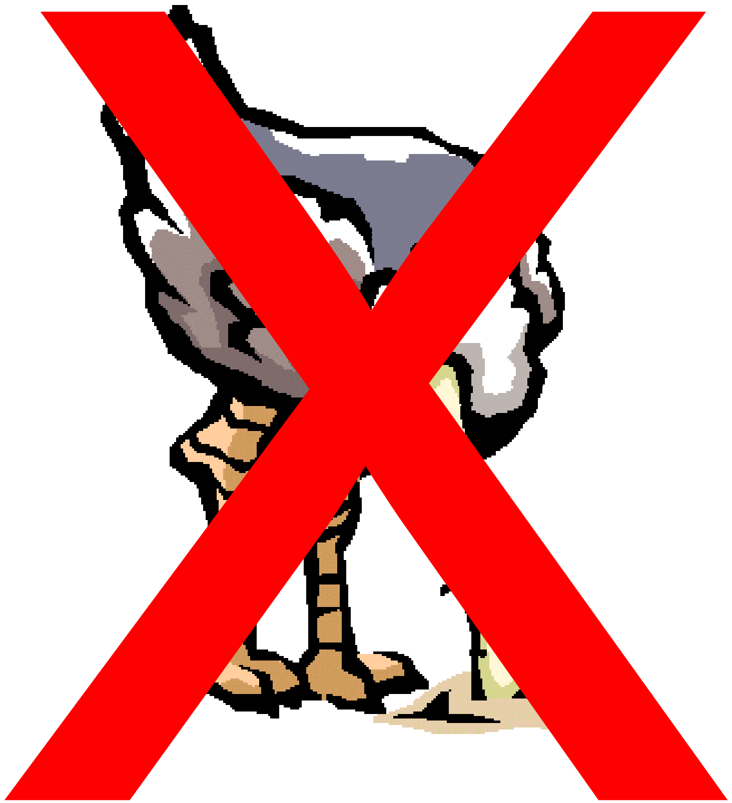
Networked Embedded Control Timing



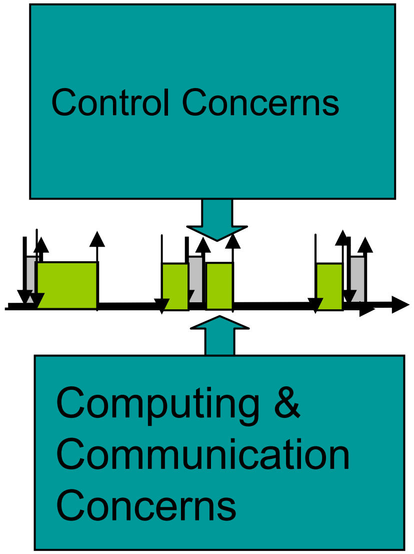
- Embedded control often implies temporal non-determinism
  - resource sharing

How should we handle this?

- Networked control often implies temporal non-determinism
  - network interface delay, queuing delay, transmission delay, propagation delay, link layer resending delay, transport layer ACK delay, ...
  - lost packets



Design Approaches



- Separation-of-concerns
  - Time-triggered and synchronous approaches
  - Simple, deterministic, dependability, …
  - But, difficult to achieve in practice due to
    - Lack of resources
    - Incorrect assumptions
    - Technology incompatibility
- Integration
  - Optimize performance subject to limited resources
  - Codesign of control computing and communication
    - Implementation-aware control techniques
    - Control-aware computing and communication techniques
    - New analysis and design tools

Control Performance

Scheduling & Network Parameters (T,D,Prio, Protocol,…) → Task Timing Parameters (latencies, jitter, …) → Control Performance (variance, rise time, overshoot, ….)

Complex relationship

Complex relationship

- In general:
  - sampling jitter has a negative effect on performance
  - a short latency is better than a long latency
  - latency jitter is bad, but a short jittery latency is in most cases better that a longer constant latency, also if the latter is compensated for
- However, anomalies exists

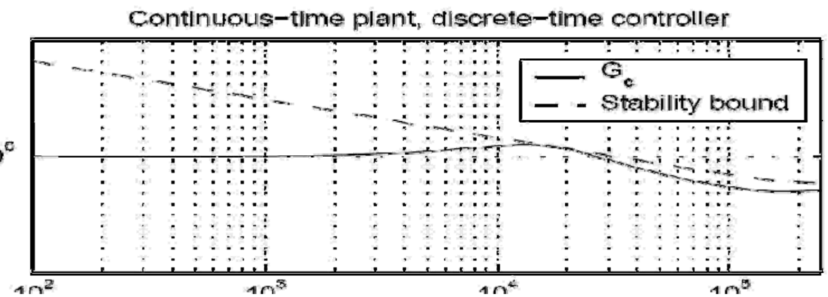
Control Design Methodology

1. Maximize temporal determinism
   – Time-triggered implementation techniques
      • static scheduling, TDMA protocols, …
      • **but,** inflexible, scalability problems, ….
   – Event-triggered approaches
      • dynamic scheduling techniques (RMA, EDF)
      • **but**,
         – often disregards latency jitter
         – seldom any concern for best-case and average-case values
   – Buffering schemes to achieve constant latencies
      • buffers at the actuator nodes in networked control loops
      • one sample latency for control tasks
   – Estimation of signal values at unavailable time points
   – ….

Control Design Methodology

1. ….
2. Temporal Robustness Analysis
   - Apply analysis techniques that determine how much temporal non-determinism that the control loop can tolerate
   - Use this information to relax the requirements on the implementation platform
   - E.g. The Jitter Margin
     - A measure of how much time-varying input-output latency a control loop can tolerate before becoming unstable

$$\left|\frac{P_{\mathrm{alias}}(\omega)K(e^{i\omega})}{1+P_{\mathrm{ZOH}}(e^{i\omega})K(e^{i\omega})}\right| < \frac{1}{\sqrt{J}\,|e^{i\omega}-1|},$$



Control Design Methodology

1. ....
2. ....

3. Off-line (Passive) Compensation

- design the controller to be robust towards the timing variations using the knowledge about the timing variations that is available at design-time
  - mean value
  - variance
  - max/min values
  - distributions,
  - ….
- example:
  - a controller that compensates for average i-o latency

Control Design Methodology

1. ….
2. ….
3. ….

4. Online (Active) Compensation
   – dynamic compensation based on measurements
   – Example:
     • Controller where the parameters are expressed as function of the sampling interval (gain-scheduling)

Task Models for Control

- **Periodic task $\tau_i = (C_i, D_i, T_i)$ with hard deadlines**
  - The classical
  - Does not take the internal structure of the task into account
  - Does not allow for occasional overruns
- **Subtask models**
  - Two parts: CalculateOutput and UpdateStates
  - Four parts: Sampling, CalculateOutput, Actuation, UpdateStates
  - Minimize input-output latency
- **Firm task models**
  - E.g. (m,k) models
  - Allow for occasional overruns
- **Elastic task models**
  - Models how the control performance depends on the size of the overruns
- **Imprecise task models**
  - Mandatory part + optional “anytime” part
  - Suitable for controllers that do online optimization, e.g., MPC
- **….**

Other Issues

- Safety & Fault-tolerance
  - Feedback often safety-critical
- Security
  - E.g. wireless control makes security highly relevant also in embedded control
- Verification
  - Control loops involve both physical parts, hardware, and software
  - Necessary to verify the operation of the entire system, not only the software → Hybrid verification
- Component technology
  - Used a long time in industrial automation (e.g., IEC 61131-3)
  - The need to minimize input-output latency puts important requirements on component frameworks, e.g., AUTOSAR– not always well understood

Control Kernel

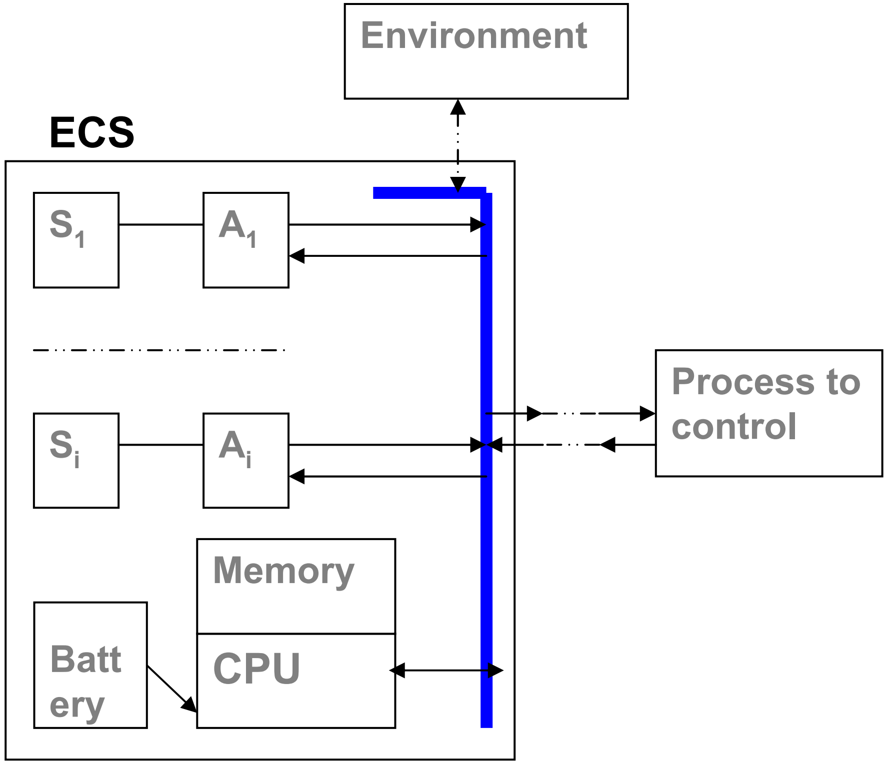
Main Concerns in Computer Control

- Unavoidable delays between sampling & updating
- Sampling period may be changed
- Signal transmission may be delayed (or missing)
- Time sequencing may depend on other tasks
- Additional tasks may change the allocated resources: computation time, memory, data access

as a result →

- Non conventional sampling/updating pattern
- Delays and missing data
- Modes and sampling rate changes (alternatives)

ECS Operating Conditions



- Multiloop control
- No regular sampling
- Information loss
- CPU optimization
- Variable delays
- Sampling period changes
- Mode changes
- Fault tolerant
- Battery control
- Safe operation

ECS Design →

ECS: Control Algorithm viewpoint

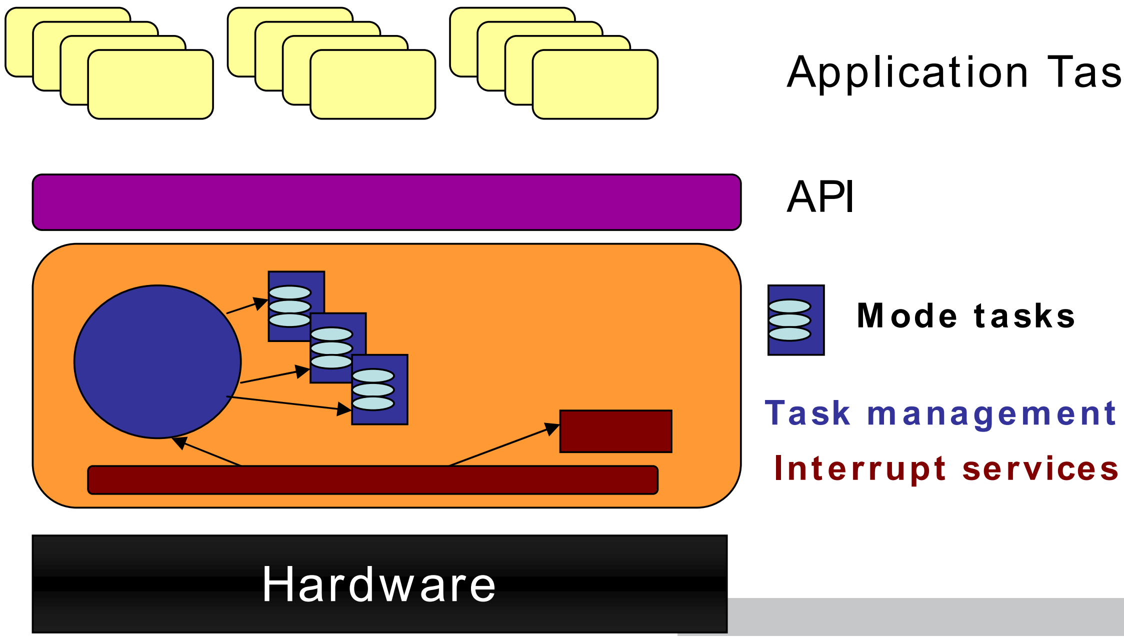
- Reduced order models
- Non-conventional sampling and updating patterns
  - Missing data control
  - Event-triggered control
- Decision and supervisory control
  - Hybrid control systems
  - Multimode control
  - Sampling rate changes
- Fault-tolerant control
- Degraded and back-up (safe) control strategies
- Battery monitoring and control

Kernel Concept

OS kernel:

- Basic services:
  - Task and time management
  - Interrupt handling
  - Interface to the applications (API)
  - Mode changes
  - Fault tolerance

OS Kernel structure



Additional services

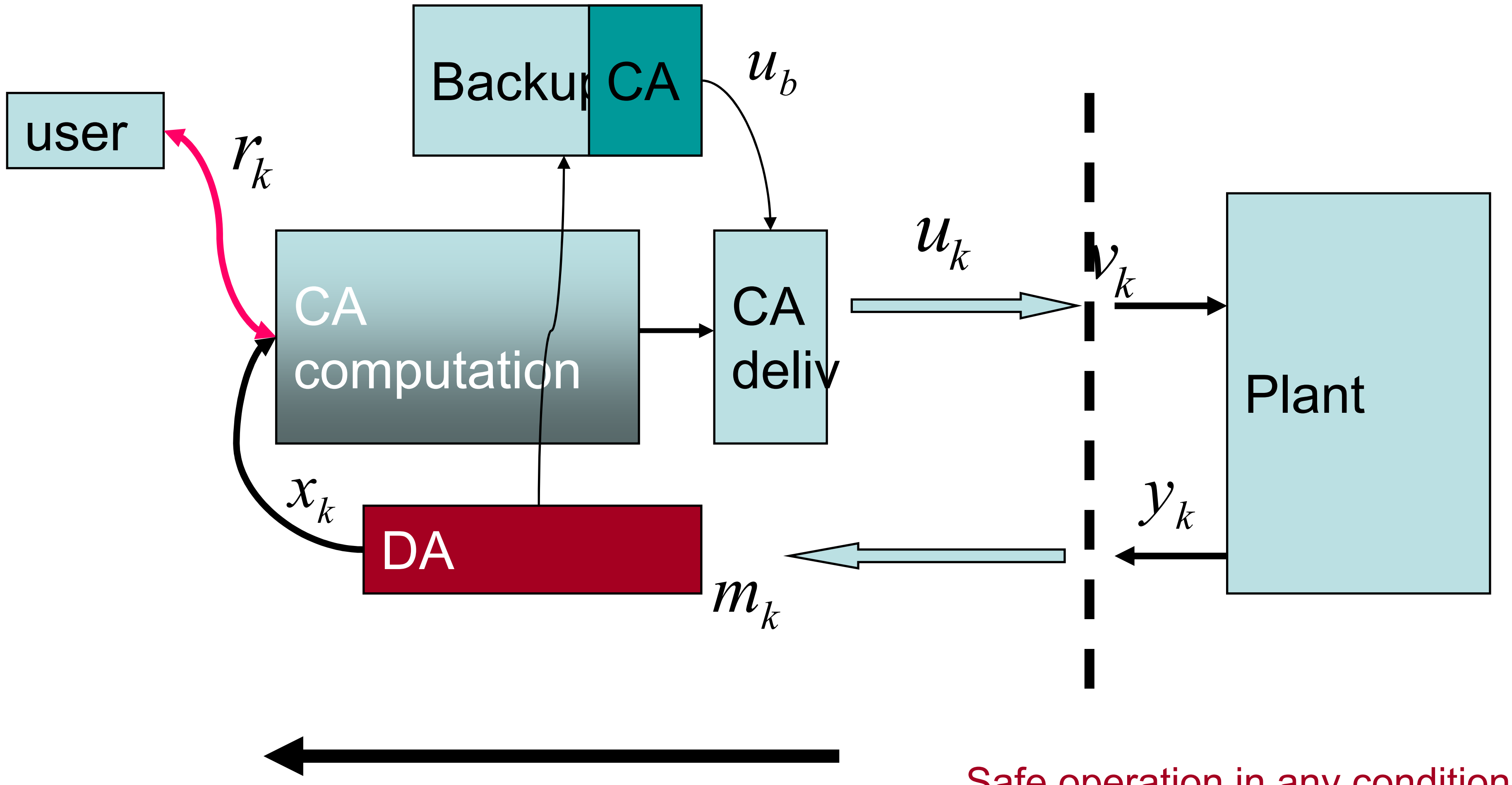
- File management
- Quality of service
- Tracing and debugging

OS Kernel for control

The OS Kernel provides the minimal services that should be included in any embedded **control** system.

- **Fault tolerance**
  - Degrade task activity (when a task does not guarantee some timing constraints, the degraded behavior is executed)
  - Change mode events raised when some faults can not be managed.
- **Mode changes**
  - Mode definition (set of tasks associated to a mode)
  - Mode change events (event to change from one mode to another)
  - Mode change protocol

The control kernel concept



Safe operation in any condition

Control Kernel

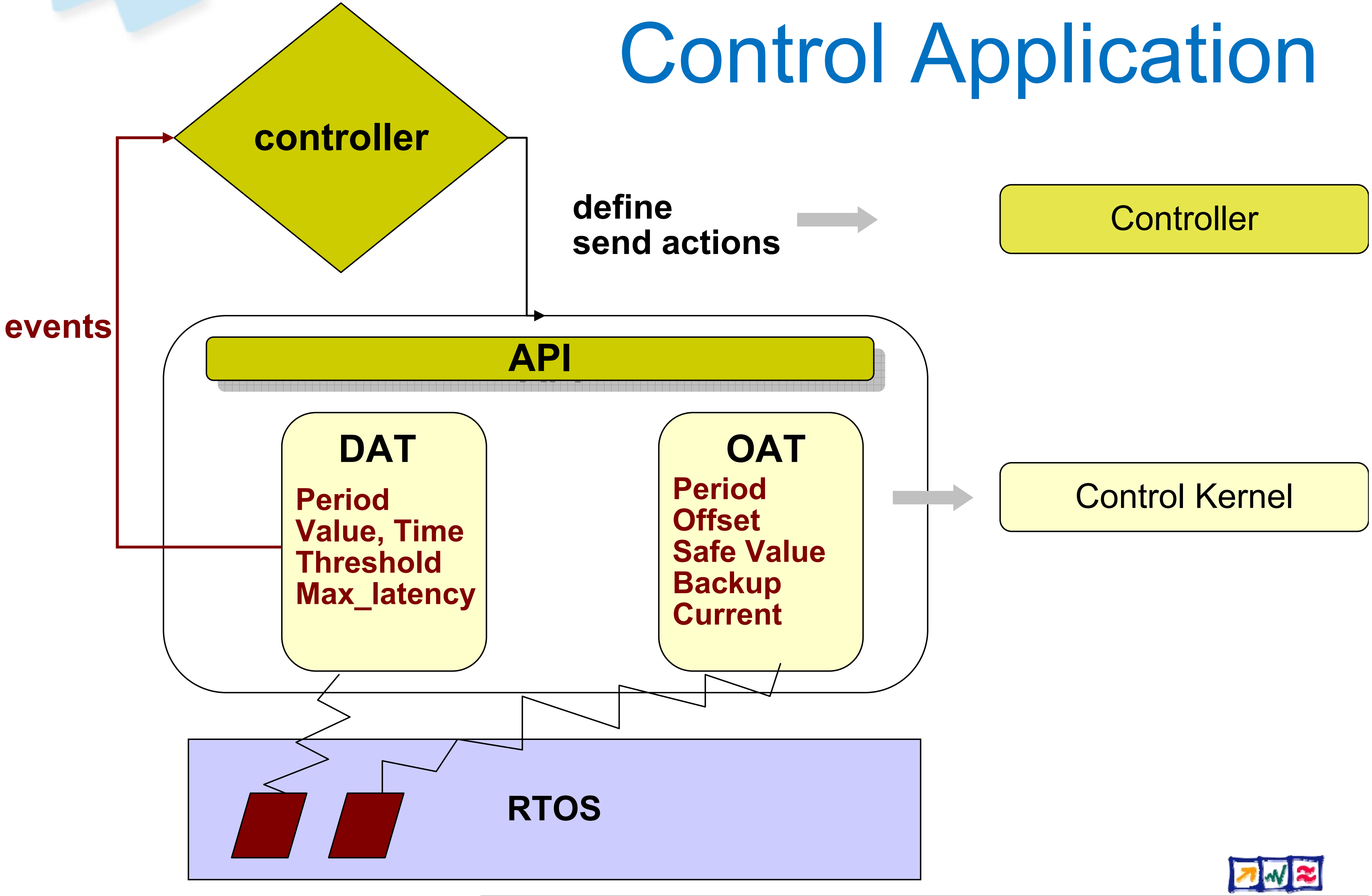
- Ensures control action (CA) delivering
  - Safe (back-up) CA computation
  - Safe CA computation based on previous data
- Data acquisition of major signals
  - Safe CA computation based on current data
- Transfer to new control structure
  - Basic control structure parameters computation
  - CA computation

Control Kernel

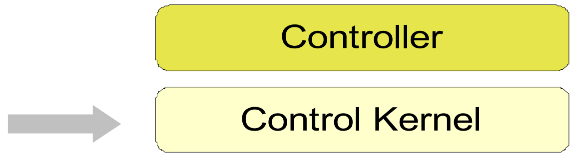
Controller

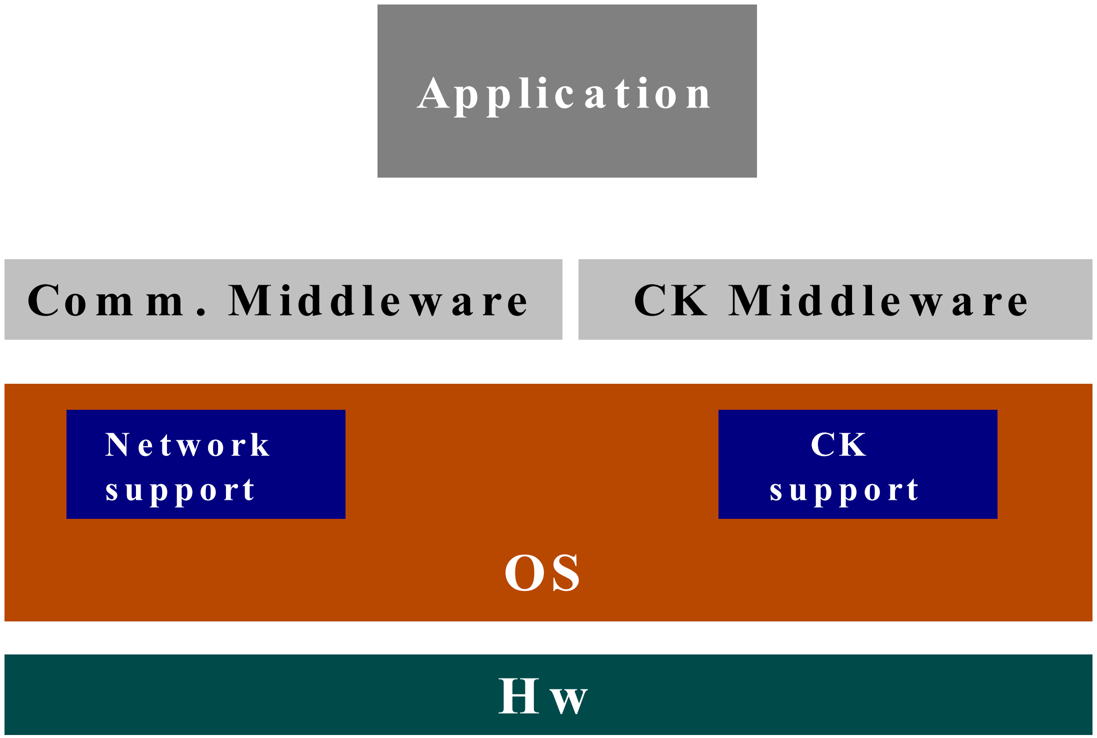
- Full DA
  - Control structures evaluation and selection
  - CA computation (different levels)
- Communication facilities
- Coordination facilities

Control Application

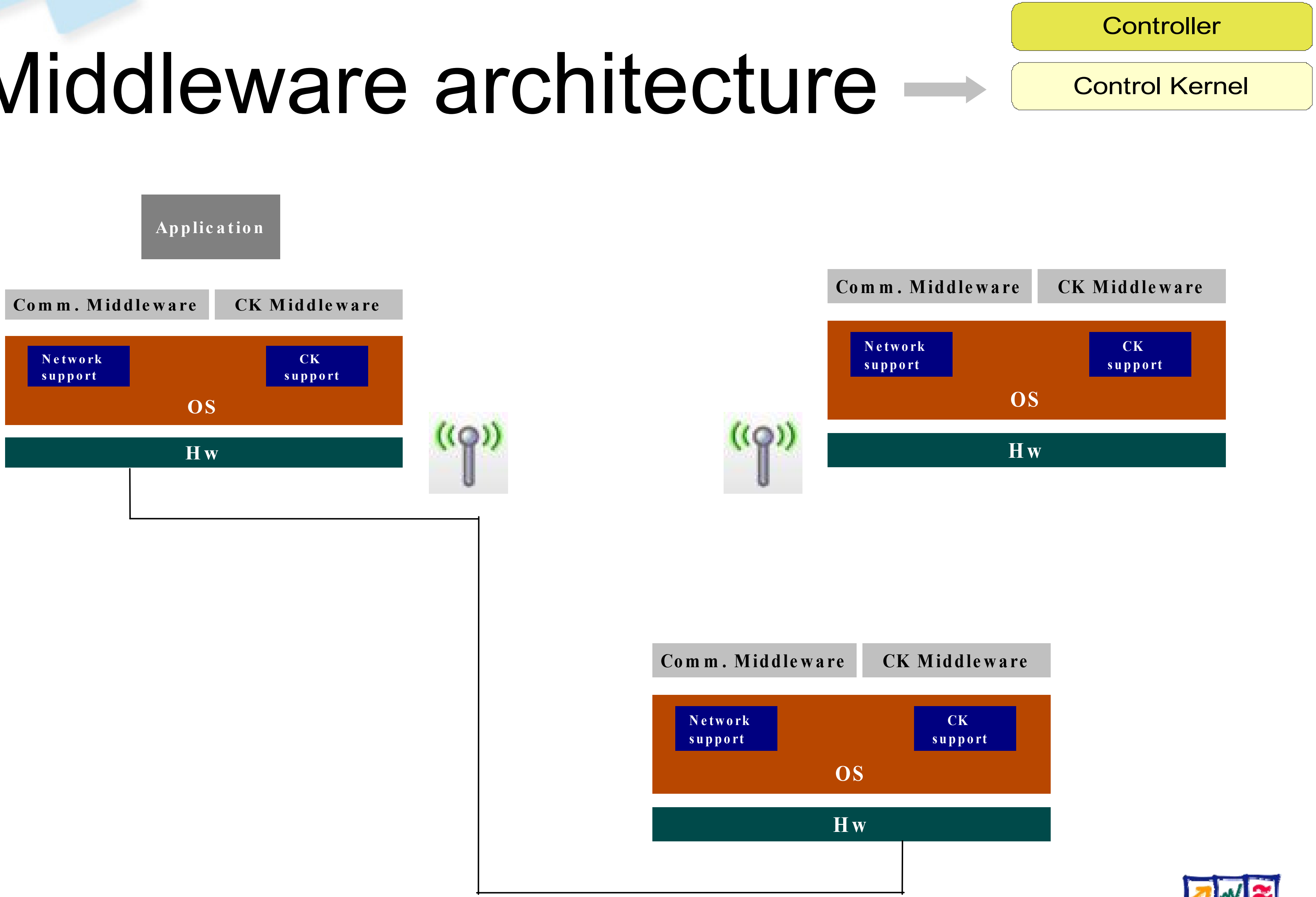


CK Middleware





Middleware architecture



CK Middleware functionality

- Provides object classes for sensors, actuators, controllers
- Remote communication through Comm. Middleware
- Pool of threads at different priority levels (acquisition, data acquisition, basic computation)
- Admission control (negotiation)
- Mode change (task + controllers commutation)

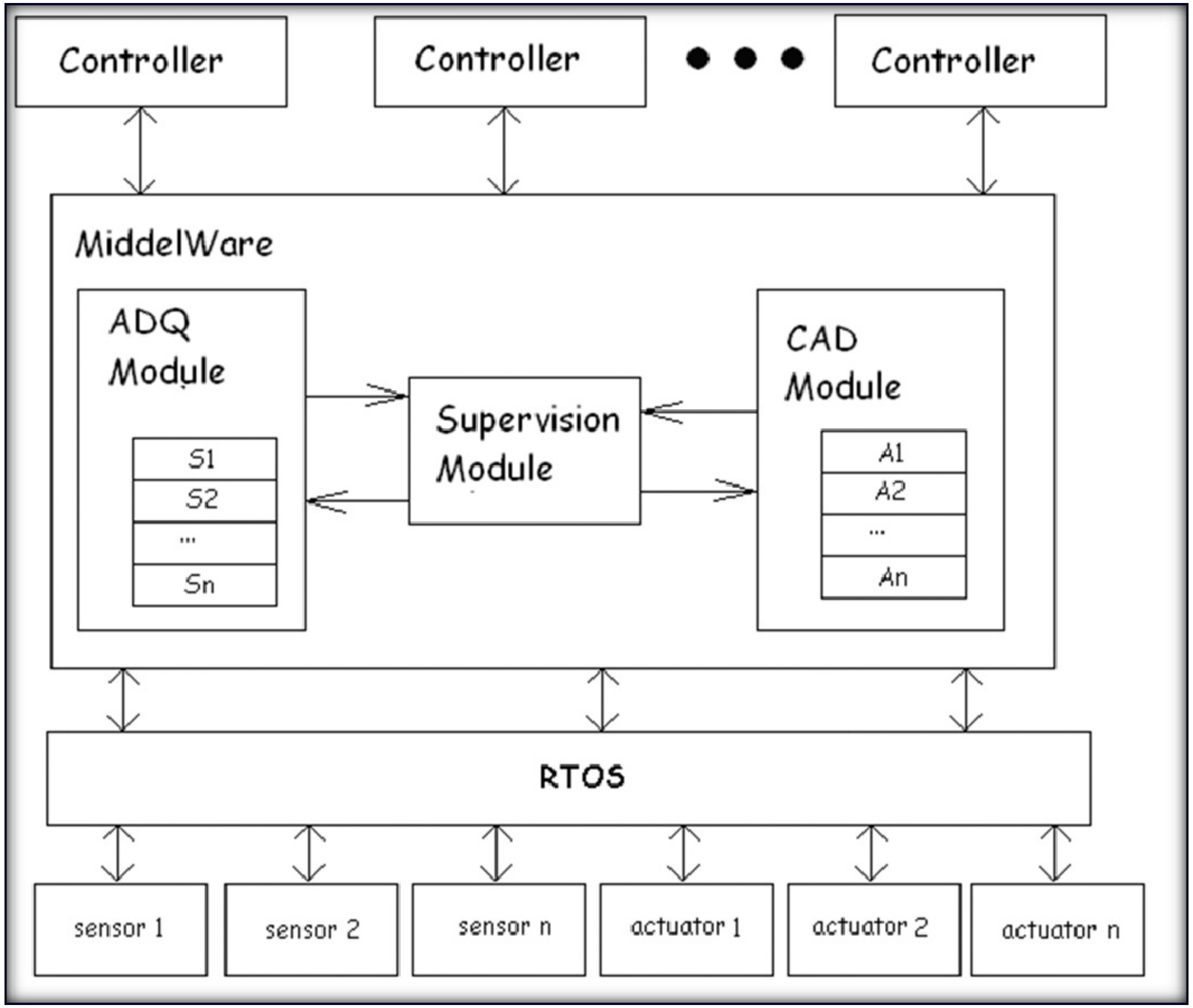
CK Middleware functionality (II)

- Definition of controller parameters:
  - Reduced model controller
  - Backup actuation
  - Sensor characteristics (virtual/real, range, acquisition period, filter, threshold, ...
  - Actuator characteristics
  - Call-back function
- Compute RMController (locally)

CK Middleware structure

Controller

Control Kernel

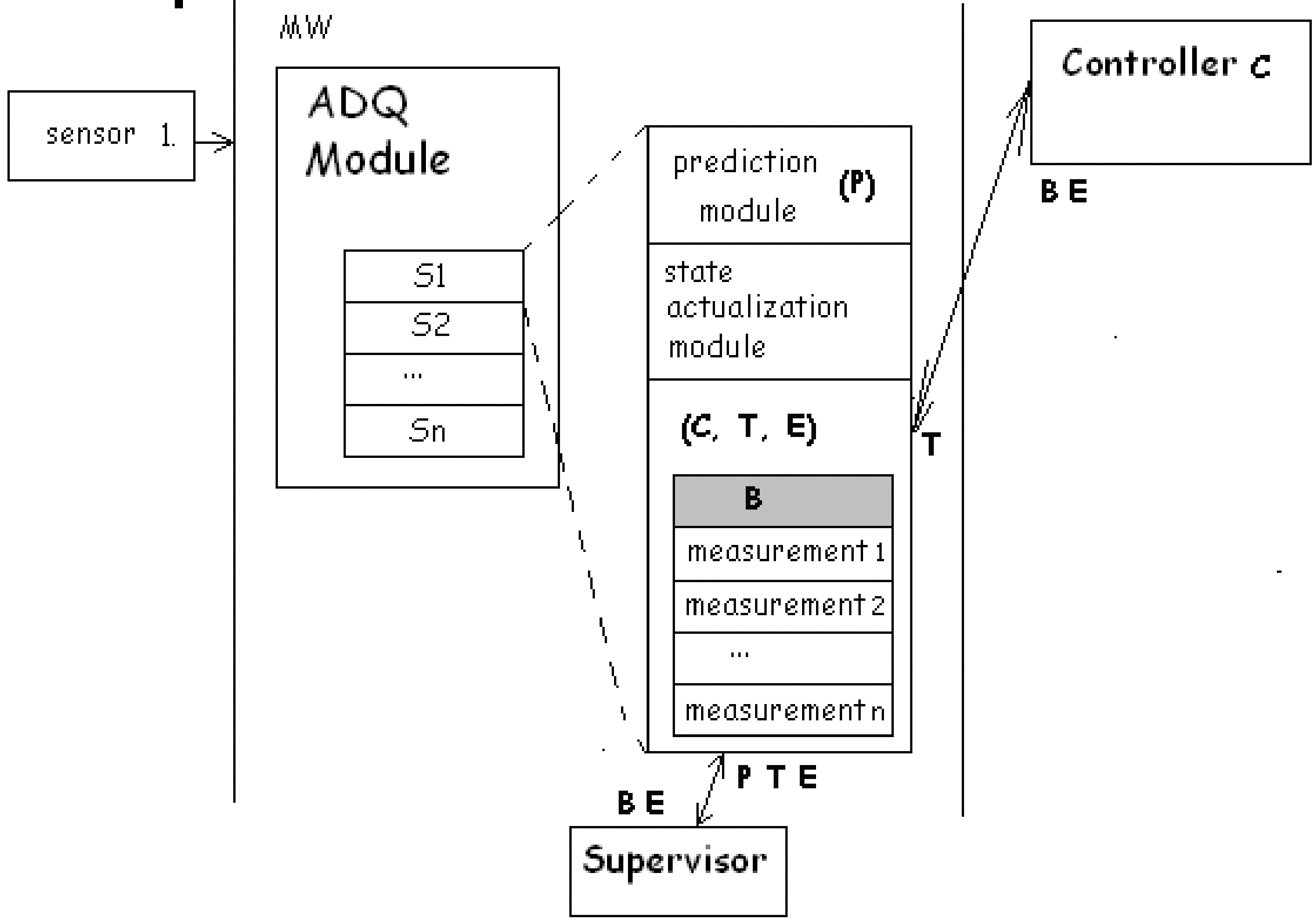


Controller

Control Kernel

CK Middleware structure

Data acquisition



CK Middleware structure

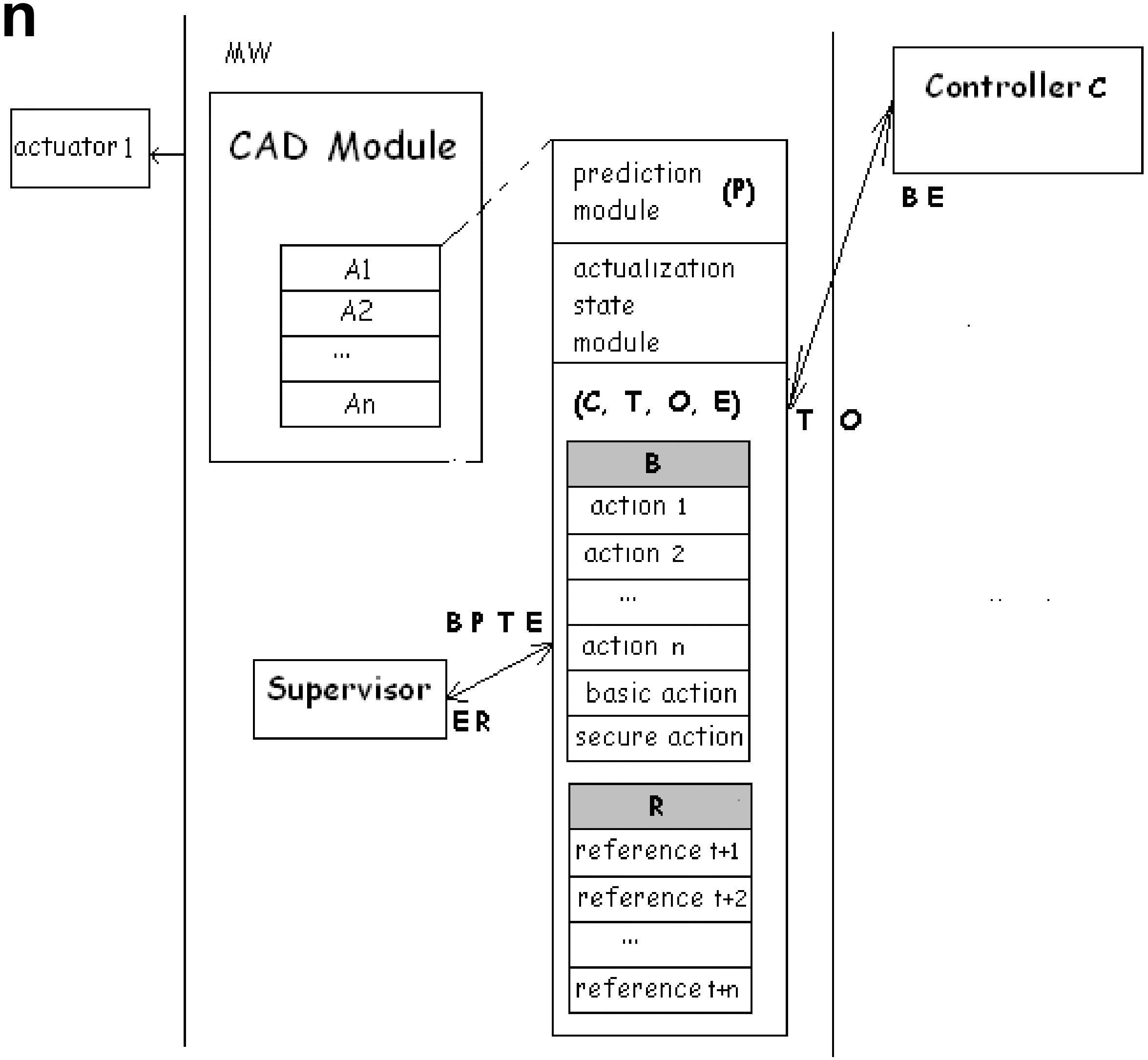
- Each physical sensor S has:
- S = {T, B, C, E},
  - T: sampling period,
  - B: buffer n values,
  - C: the controller function
  - E: the sensor state {fail, event, no_fail}
- Acquisition quality
  - Via data acquisition interval (DAI) concept

Controller

Control Kernel

CK Middleware structure

Actuation

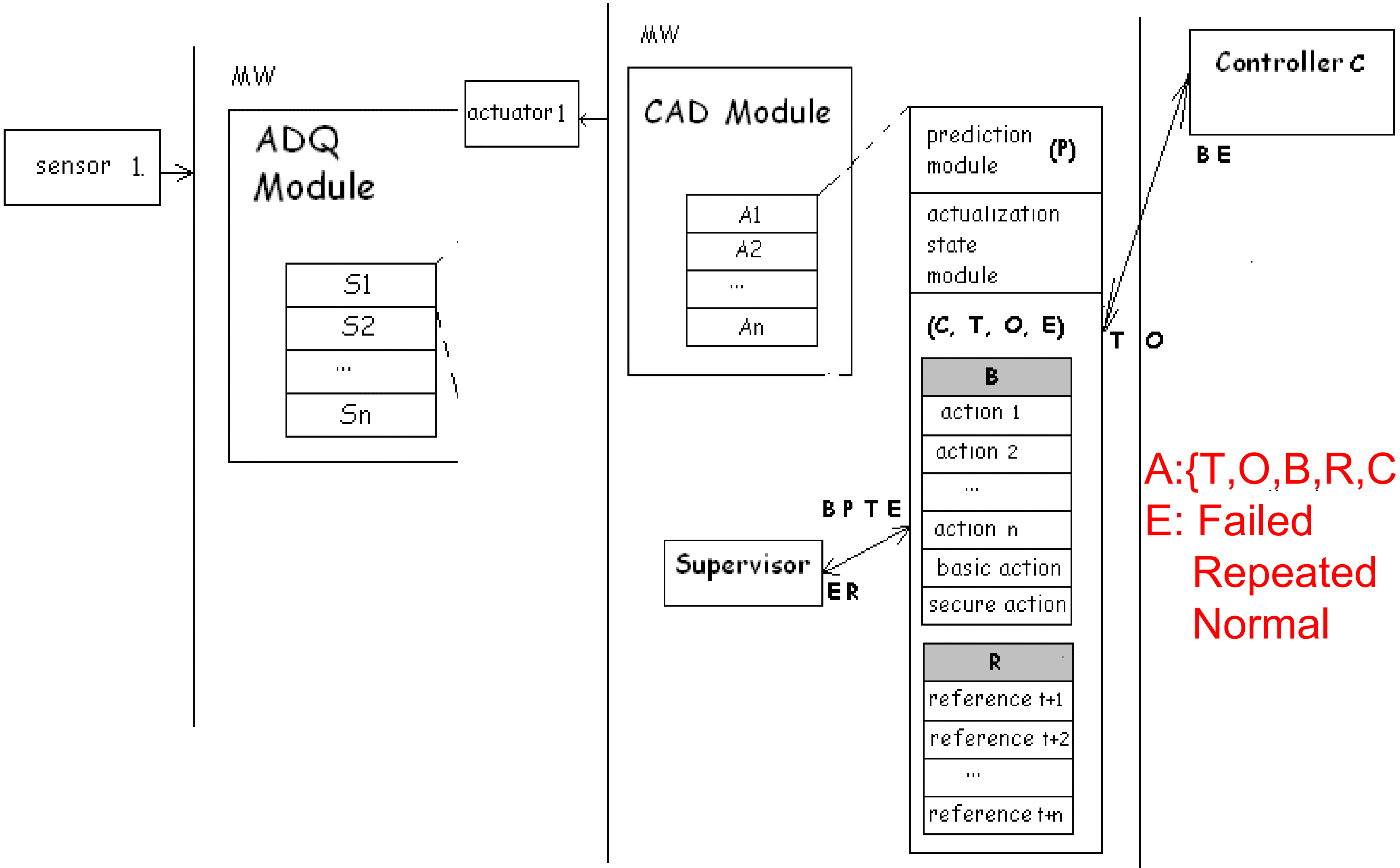


CK Middleware structure

Actuation

- Each physical actuator A has:
- A= {T, O, B, R, C, E}
  - T: sampling period,
  - O: Offset between delivering of the action and acquisition of data.
  - B: buffer n values.
  - R: To store the n future references values.
  - C: the controller function.
  - E: the sensor state {fail, event, no_fail}
- Delivering actions quality
  - Via data acquisition interval (CAI) concept

Control Kernel structure



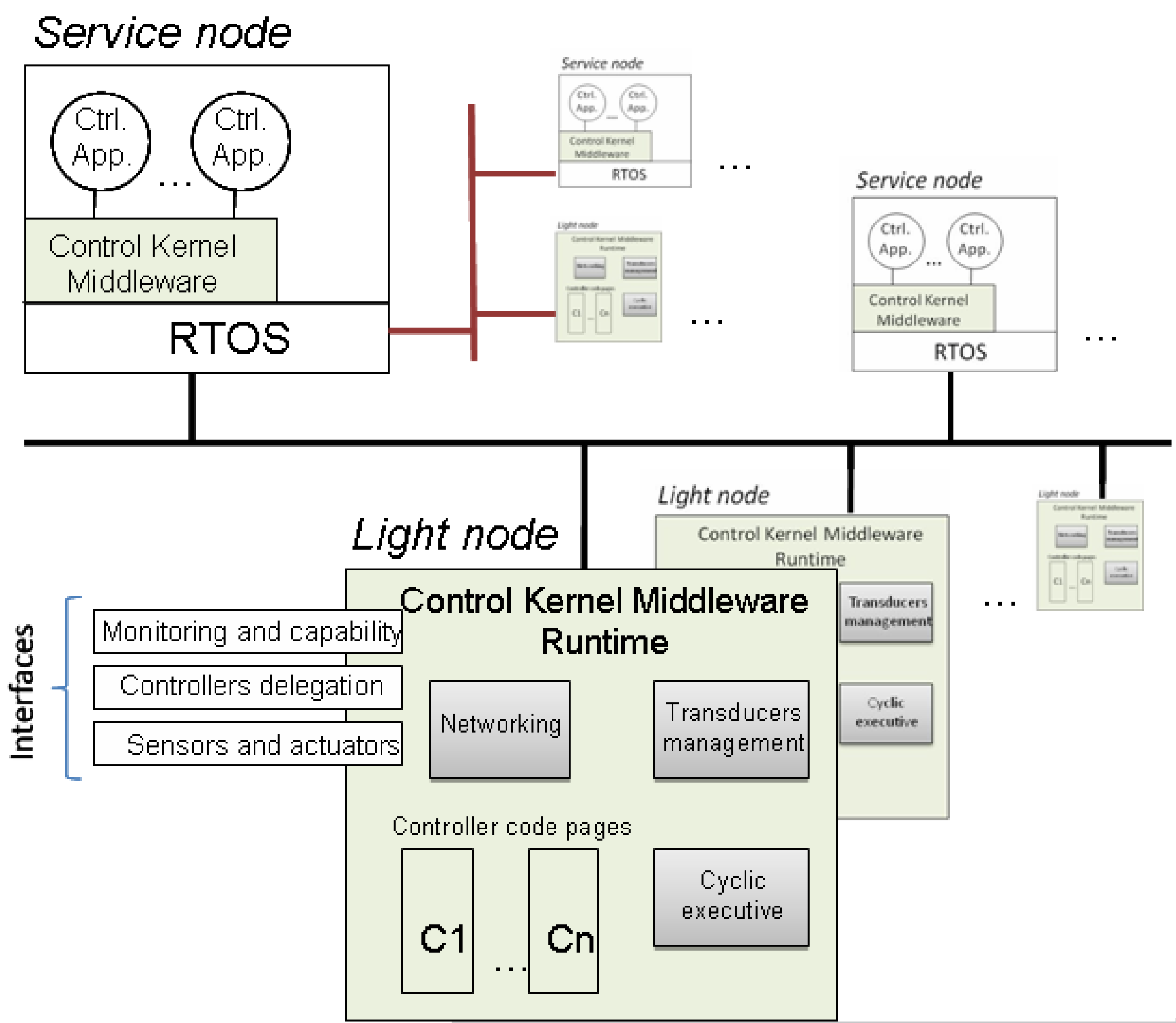
A:{T,O,B,R,C,E}

E: Failed

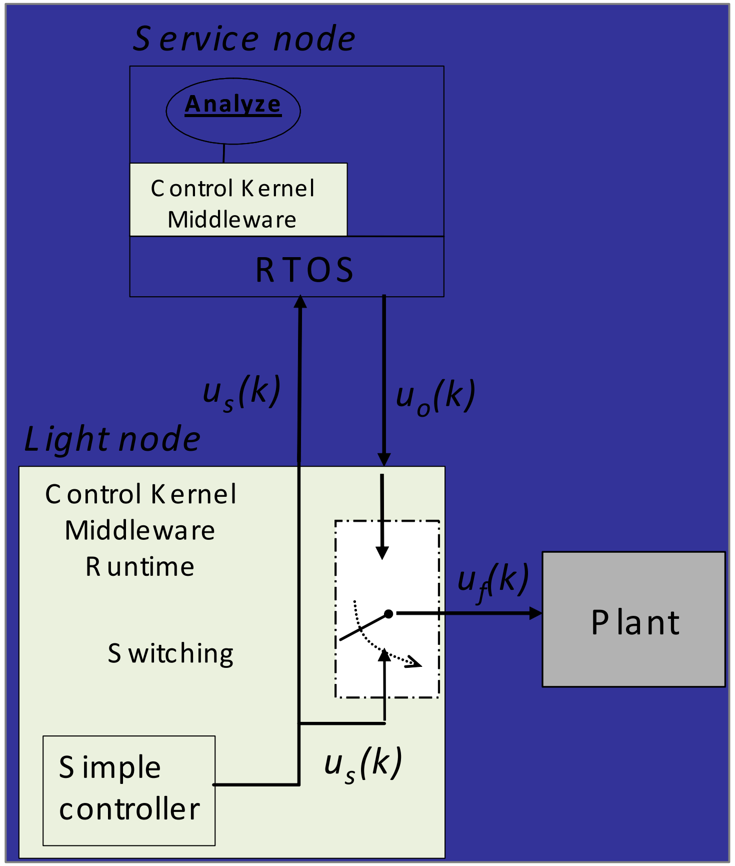
Repeated

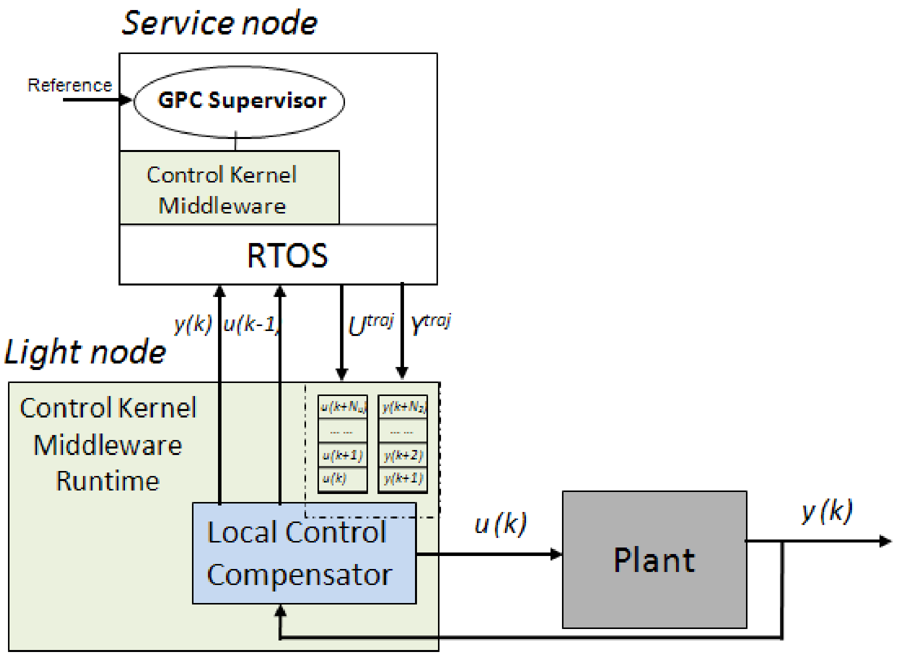
Normal

CK layout


Service node
Ctrl. App.
Ctrl. App.
...
Control Kernel Middleware
RTOS
Service node
RTOS
Light node
Service node
Ctrl. App.
Ctrl. App.
Control Kernel Middleware
RTOS
Light node
Control Kernel Middleware Runtime
Transducers management
Cyclic executive
Light node
Control Kernel Middleware Runtime
Interfaces
Monitoring and capability
Controllers delegation
Sensors and actuators
Networking
Transducers management
Controller code pages
C1
...
Cn
Cyclic executive

CK detail


Service node
Analyze
Control Kernel
Middleware
RTOS
$u_s(k)$
$u_o(k)$
Light node
Control Kernel
Middleware
Runtime
Switching
$u_f(k)$
Plant
Simple
controller
$u_s(k)$


Service node
Reference
GPC Supervisor
Control Kernel
Middleware
RTOS
y(k)
u(k-1)
U^{traj}
Y^{traj}
Light node
Control Kernel
Middleware
Runtime
Local Control
Compensator
u(k)
Plant
y(k)

Scheduling Policies

Controller

Control Kernel

- Several scheduling policies can coexist depending on the thread level.
- Kernel threads (DAThread and OAThread) are executed as part of the RTOS. Both are periodics and serve acquision and delivery actions
- Both have a **queue** were requests are served on deadline basis.
- Values are written/read to/from control kernel middleware.

Controller

Control Kernel

Programming

- The application has to define the elements involved in a controller:

```
procedure configuration {
//allocating resources and failure handlers
// definition of the objects......

    idq1=register_sensor(t,p1,buffersize);
    idq2=register_sensor(t,p2,buffersize);
    register_fail_detection(idq1,twindow, thrsh, nthresh, maxInc, sc, p , r);
    register_fail_detection(idq2,wt, h, cd, d, sc, p , r);
    idu=register_actuator(t,o,p1,buffersize);
    register_controller(user_controller, [idq1 idq2] ,idu);
// start control
   start_controllers();
}
procedure finish {
// free the resources allocated
   stop_controllers();
}
```

Implementation

- Current version of the CK Middleware has been implemented in C
- The RTOS used is PartiKle and open-source rtos which is the new core of RTLinux_GPL

- It can be executed in x86 or ARM processors
- Different execution platforms

Conclusions about the implementation

- **Control kernel extracts** the **basic control services** and defines a middleware for control purposes.
- The scheduling scheme that support this architecture permits the execution of the control kernel and control loops.
- Several scheduling policies are applied to the different control levels based on EDF and DM.
- Also, the control kernel obtains an improvement with respect to the CPU use when the system is in stable conditions.
- An implementation of the control kernel in a rtos has been experimented

Fixed-Point Arithmetic

Arithmetic in Embedded Control

- Microcontrollers used in small embedded systems and sensor networks typically do not have hardware support for floating-point arithmetic
- Options
  - Software emulation of floating-point arithmetic
    - Compiler/library supported → easy
    - Large code size, slow
  - Fixed-point arithmetic
    - Often manual implementation → hard
    - Fast and compact
    - Fallen out of most curricula

Fixed-Point Arithmetic

- Idea: Represent all numbers (signals, parameters) using **integers**

- Use **scaling** to make all numbers fit into one of the integer data types, e.g.,
  - 8 bits (char, int8_t): [-128, 127]
  - 16 bits (short, int16_t): [-32768, 32767]
  - 32 bits (long, int32_t): [-2147483648, 2147483647]

Challenges

- Must select data types to get sufficient numerical precision
- Must know (or estimate) the minimum and maximum value of every variable in order to select appropriate scaling factors
- Must keep track of the scaling factors in all arithmetic operations
- Must handle potential arithmetic overflows
- Must choose a good controller realization

Fixed-Point Representation

In fixed-point representation, a real number x is represented by an integer X with N = QI + QF + 1 bits, where

- N is the word length
- QI is the number of integer bits (excluding the sign bit)
- QF is the number of fractional bits

Q-format: X is called a Q[QI].[QF] number

Conversion from real to fixed-point number:

$$X := \mathrm{round}(x \cdot 2^{QF})$$

Conversion from fixed-point to real number:

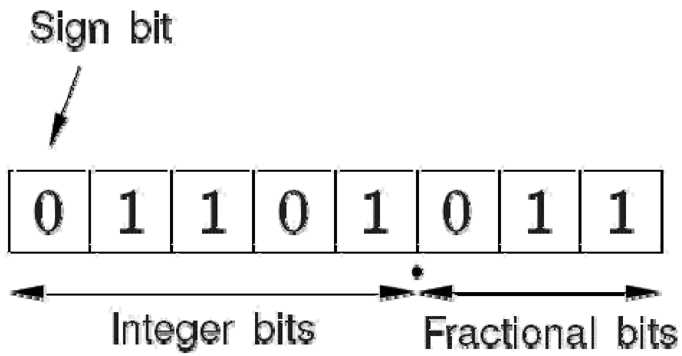
$$x := X \cdot 2^{-QF}$$

Example

Convert the real number x = 13.4 to a fixed-point number
X □ Q4.3

$$X = \mathrm{round}(13.4 \cdot 2^3) = \mathrm{round}(107.2) = 107$$

In the computer, 107_{10} will be stored as 01101011_2:



- Ordinary two's complement is used for negative numbers
- Converting back, we get x := 107 × 2^{-3} = 13.375 (quantization)

Fixed-Point Addition/Subtraction

- Two fixed-point numbers with the same number of fractional bits can be added or subtracted directly
- The result will have the same number of fractional bits

$$z = x + y \quad \Leftrightarrow \quad Z = X + Y$$

$$z = x - y \quad \Leftrightarrow \quad Z = X - Y$$

(If the variables have different numbers of fractional bits, scaling must be performed before the operation)

Fixed-Point Multiplication and Division

- If both the operands and the result are in the same Q-format, multiplication and division are done as

$$z = x \cdot y \quad \Leftrightarrow \quad Z = (X \cdot Y)/2^{QF}$$

$$z = x/y \quad \Leftrightarrow \quad Z = (X \cdot 2^{QF})/Y$$

- Double word length is needed for the intermediate result
- Multiplication and division by 2^{QF} is implemented as a left-shift and right-shift by QF bits

If the operands have different Q-formats additional scaling is needed

Example: Multiplication

- Q5.3 operands and result

```
#define QF 3                  /* number of fractional bits  */
int8_t X, Y, Z;               /* Q5.3 operands and result   */
int16_t temp;                 /* Q10.6 intermediate result  */
temp = (int16_t)X * Y;        /* cast operands to 16 bits   */
Z = temp >> QF;               /* divide by 2^QF             */
```

Numerical Errors

- Addition and subtraction are error-free as long as there are no overflows
- Multiplication and division involve quantization
  - truncation or rounding

Overflow

- All fixed-point operations are prone to overflow
- Because of the internal 2-complement's representation, unexpected results can appear - **wraparound**
- Example: Two numbers in Q4.3 format are added:

$$x = 12.25 \quad \Rightarrow \quad X = 98$$

$$y = 14.75 \quad \Rightarrow \quad Y = 118$$

$$Z = X + Y = 216$$

- This number is out of range and will be interpreted as

$$216 - 256 = -40 \quad \Rightarrow \quad z = -5.0$$

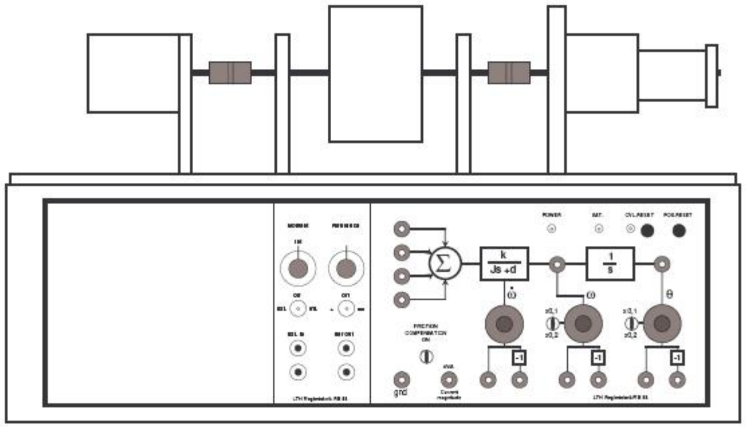
Saturation

- Some DSPs automatically saturate on overflow
- On ordinary micro-controllers, saturation must be implemented manually.
- Example (multiplication):

```
#define N 8                  /* word length                  */
#define QF 3                 /* number of fractional bits    */
int8_t X, Y, Z;              /* Q5.3 operands and result     */
int16_t temp;                /* Q10.6 intermediate result    */
temp = (int16_t)X * Y;
if (temp < -(1<<(N-1))       /* check lower saturation       */
   temp = -(1<<(N-1));       /* smallest possible number     */
else if (temp >= 1>>(N-1))   /* check upper saturation       */
   temp = (1>>(N-1))-1;      /* largest possible number      */
Z = temp >> QF;              /* divide by 2^QF               */
```

Evaluation of Execution Time and Code Size

- Atmel AVR ATmega16(L) micro-controller @14.7 MHz with 16K ROM controlling a rotating DC servo



- C program that implements simple state feedback controllers
  - **velocity control (one state is measured)**
  - **position control (two states are measured)**

- Position controller with integral action

$$u(k) = l_1 y_1(k) + l_2 y_2(k) + l_3 I(k)$$
$$I(k+1) = I(k) + r(k) - y_2(k)$$

  - Three multiplications + some summations (accumulations)
  - As simple as it can get ☺

Measurements

Software-emulated floating-point using `float` data types:

- Velocity control: 835 μs
- Position control: 980 μs
- Code size: 13708 bytes

Fixed-point using 16-bit integers (Q 3.12):

- Velocity control: 15 μs
- Position control: 40 μs
- Code size: 3748 bytes

- Speedup of 25-50 times!
- Floating-point math library takes 10k (out of 16k)!

Coefficient Quantization

- General problem when implementing controllers and filters using fixed-point arithmetic
  – Poles and zeros end up somewhere else
  – The magnitude of the problem is very much dependent on the controller realization

Realizations

A digital controller

$$u(k) = H(q^{-1})y(k) = \frac{b_0 + b_1 q^{-1} + \cdots + b_m q^{-m}}{1 + a_1 q^{-1} + a_2 q^{-2} + \cdots + a_n q^{-n}} y(k)$$

can be realized in a number of different ways with the same input-output behaviour, e.g.

- Direct form
- Companion (canonical) form
- Series (cascade) or parallel form

Direct form

$$u(k) = \sum_{i=0}^{m} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i)$$

- Nonminimal (all old inputs and outputs are used as states) and hence consume RAM memory
- Very sensitive to coefficient roundoff
- Avoid!

Companion Forms

E.g. controllable canonical form:

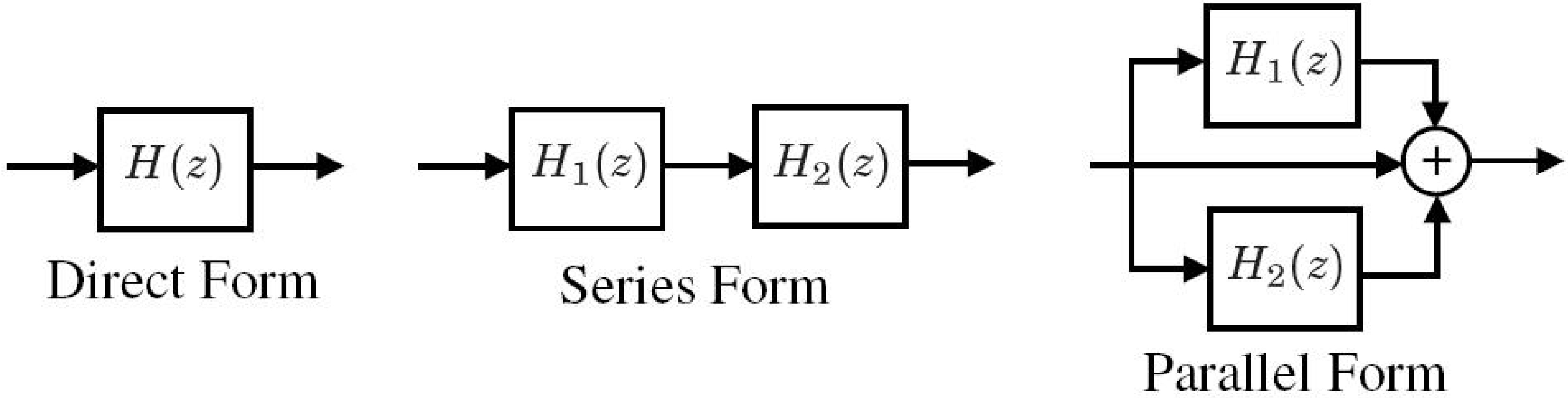
$$x(k+1) = \begin{pmatrix} -a_1 & -a_2 & \cdots & -a_{n-1} & -a_n \\ 1 & 0 & & 0 & 0 \\ 0 & 1 & & 0 & 0 \\ \vdots & & & & \\ 0 & 0 & & 1 & 0 \end{pmatrix} x(k) + \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} y(k)$$

$$u(k) = \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix} x(k)$$

- Very sensitive to coefficient roundoff
- Avoid!

Series and Parallell Forms

Divide the controller into a number of first- or second-order subsystems:

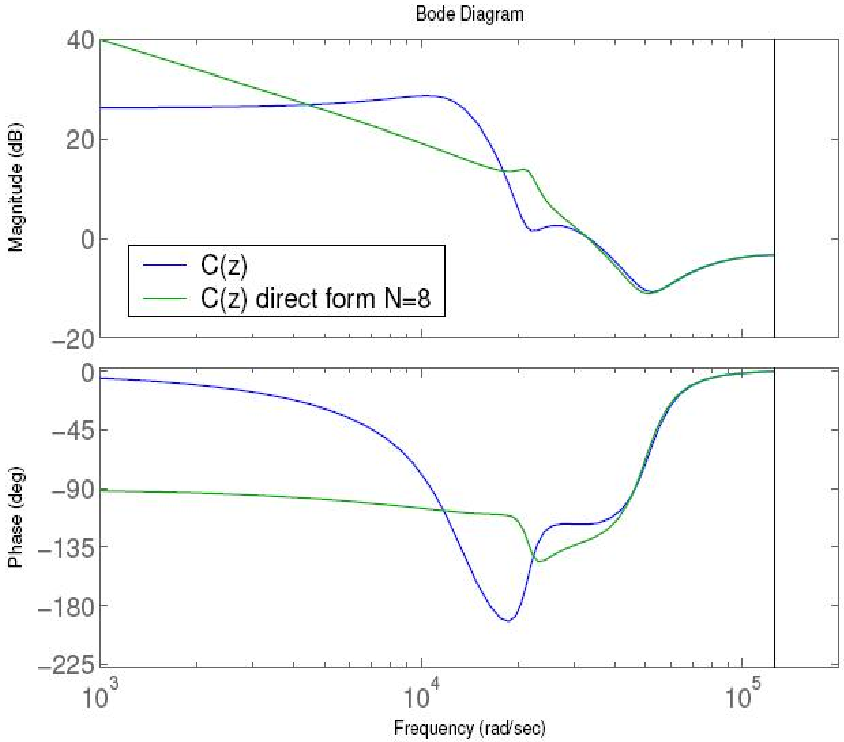


- Try to balance the gains such that each subsystem has about the same gain
- Good numerical properties!

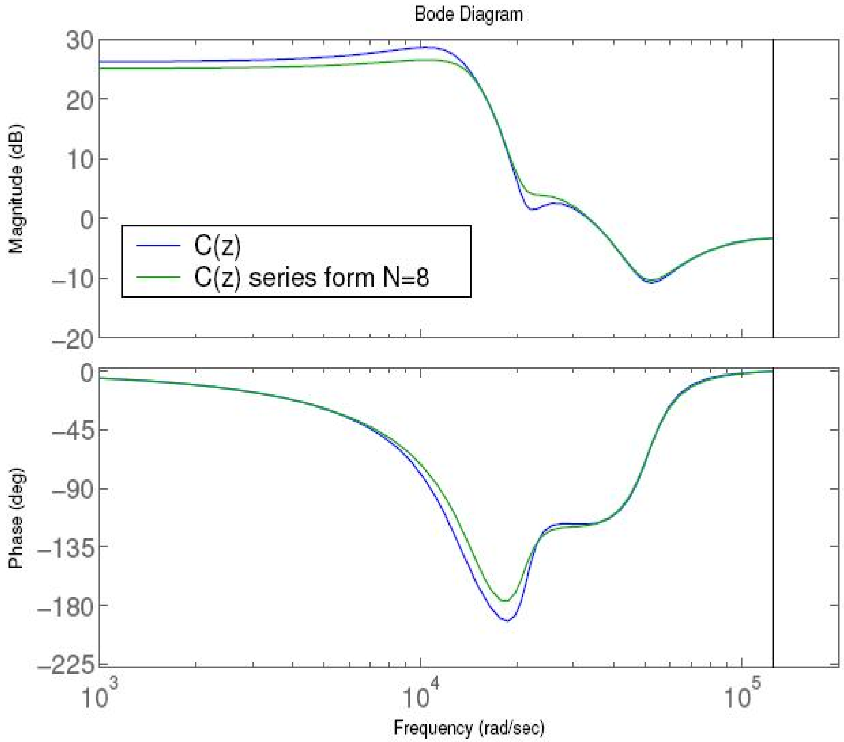
Example

$$H(z) = \frac{z^4 - 2.13z^3 + 2.351z^2 - 1.493z + 0.5776}{z^4 - 3.2z^3 + 3.997z^2 - 2.301z + 0.5184}$$

Direct form



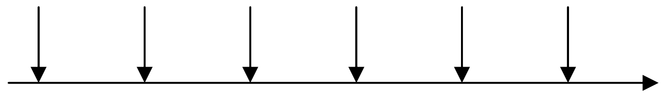
Series form



Event-Based Control

Motivation for Event-Based Control

- Control almost always assumes and requires **periodic** sampling
  - Well-developed theory



- Cornerstone for large part of real-time systems

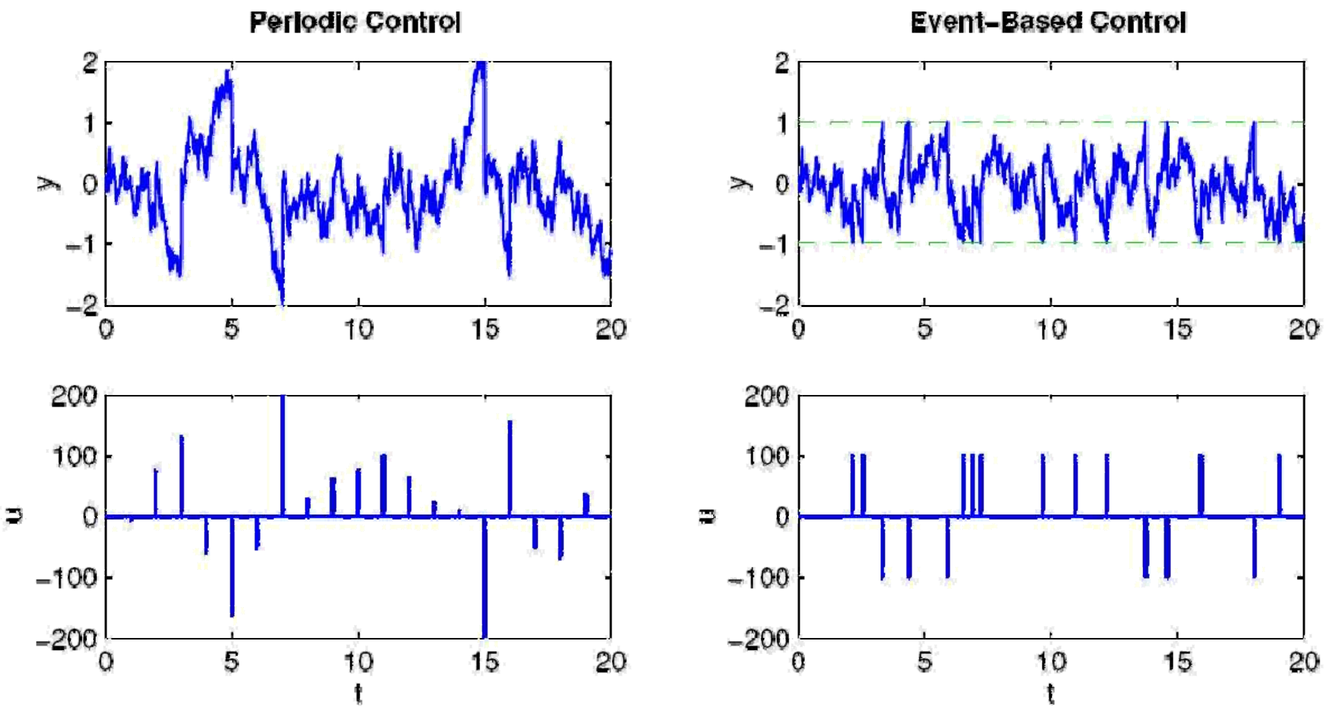
- However, what if we relax this assumption?
  - Sample and control only when an **event** has occurred, e.g., a threshold crossing
  - Most likely closer to how nature performs feedback
  - Several practical observations have reported that event-based control can perform as good or better than time-based control
  - But, very very little theory

- Applications where sampling and computation are costly
  - Networked embedded control systems with limited bandwidth
  - …

Aperiodic Control

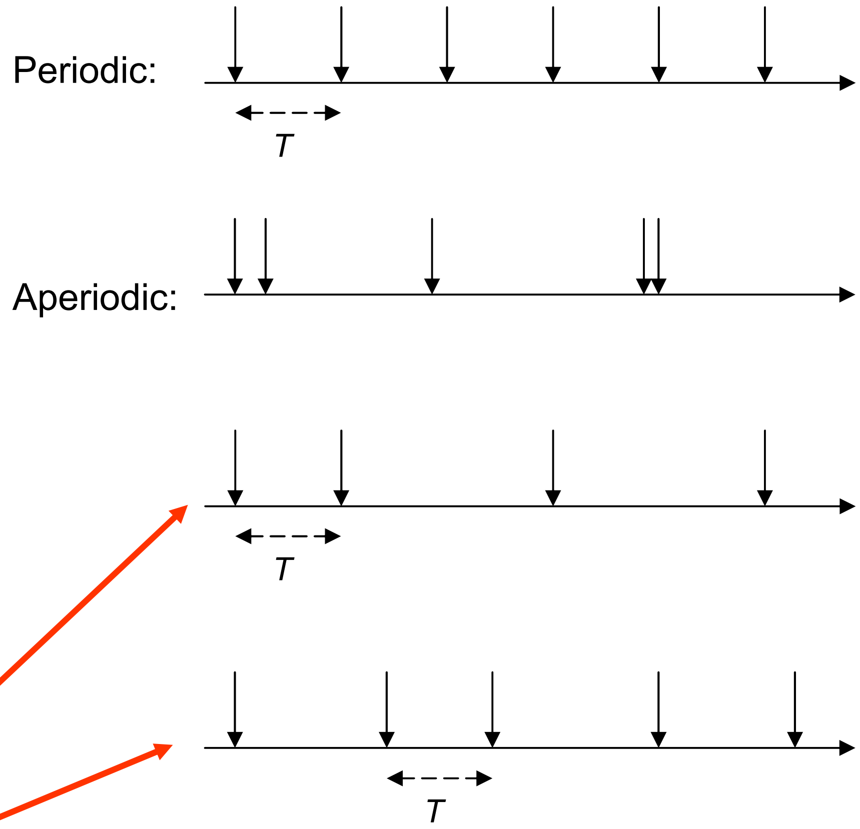
[Åström and Bernhardsson, 1999]

– Aperiodic event-based control of first-order stochastic systems
– Reduction of variance by a factor 3 for an integrator systems, assuming the same average sampling interval

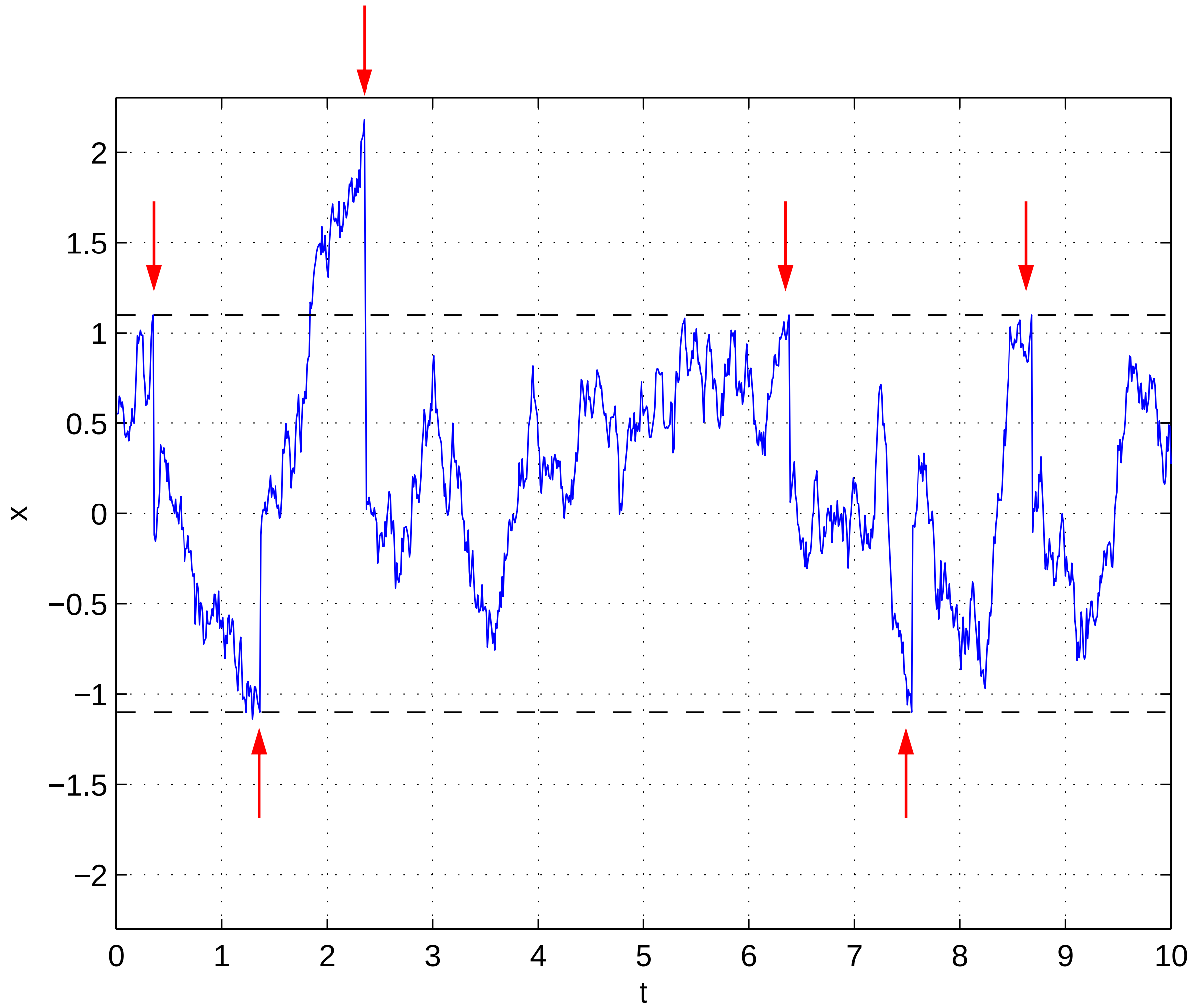


Sporadic Event-Based Control

- Recent work by Johannesson, Henningsson and Cervin, Lund
- Apply impulse control when |x| > r
  - State is reset to zero
  - Dirac-pulse control signal
- Sporadic Event-Based Control
  - Minimum inter-event time → finite utilization factor
- Two versions:
  - Discrete-time measurements
  - Continuous-time measurements



Example: Sporadic Control



Cost Function

$$J = \underbrace{\lim_{t\to\infty} \frac{1}{t} \int_0^t x^2(\tau)\, d\tau}_{J_x} + \rho \underbrace{\lim_{t\to\infty} \frac{1}{t} N_u(0,t)}_{J_u}$$

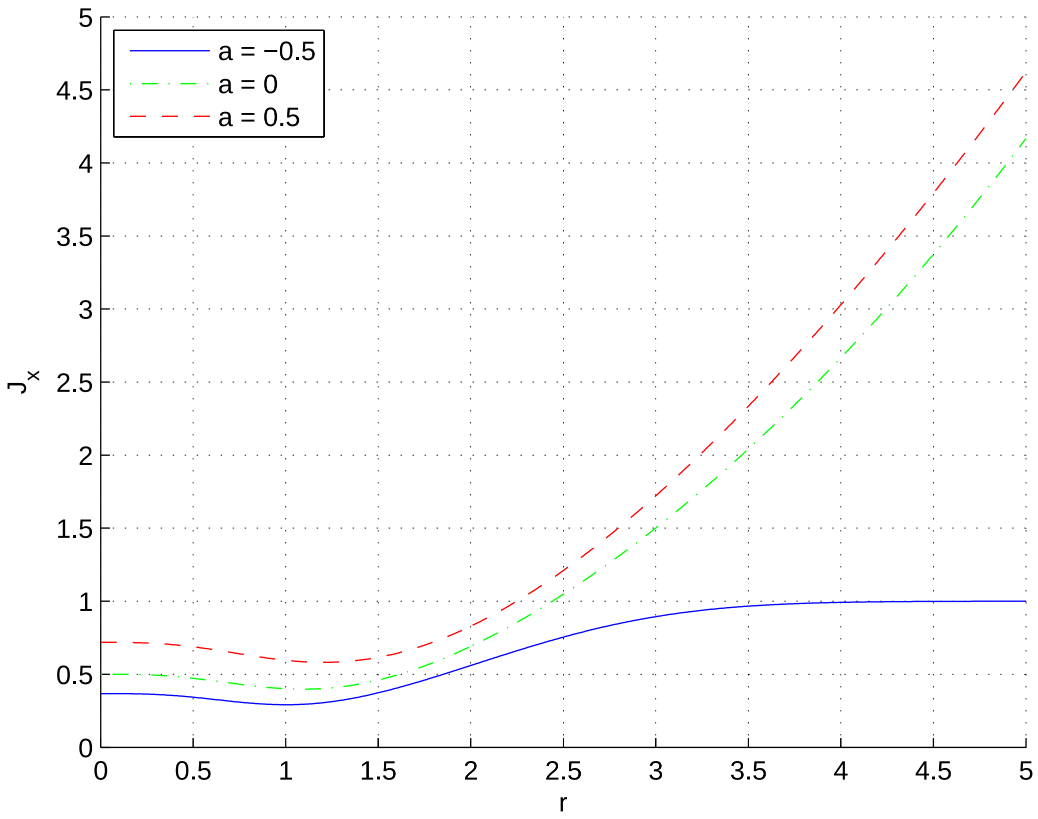
J_x = State cost

- Stationary process variance

J_u = Control cost

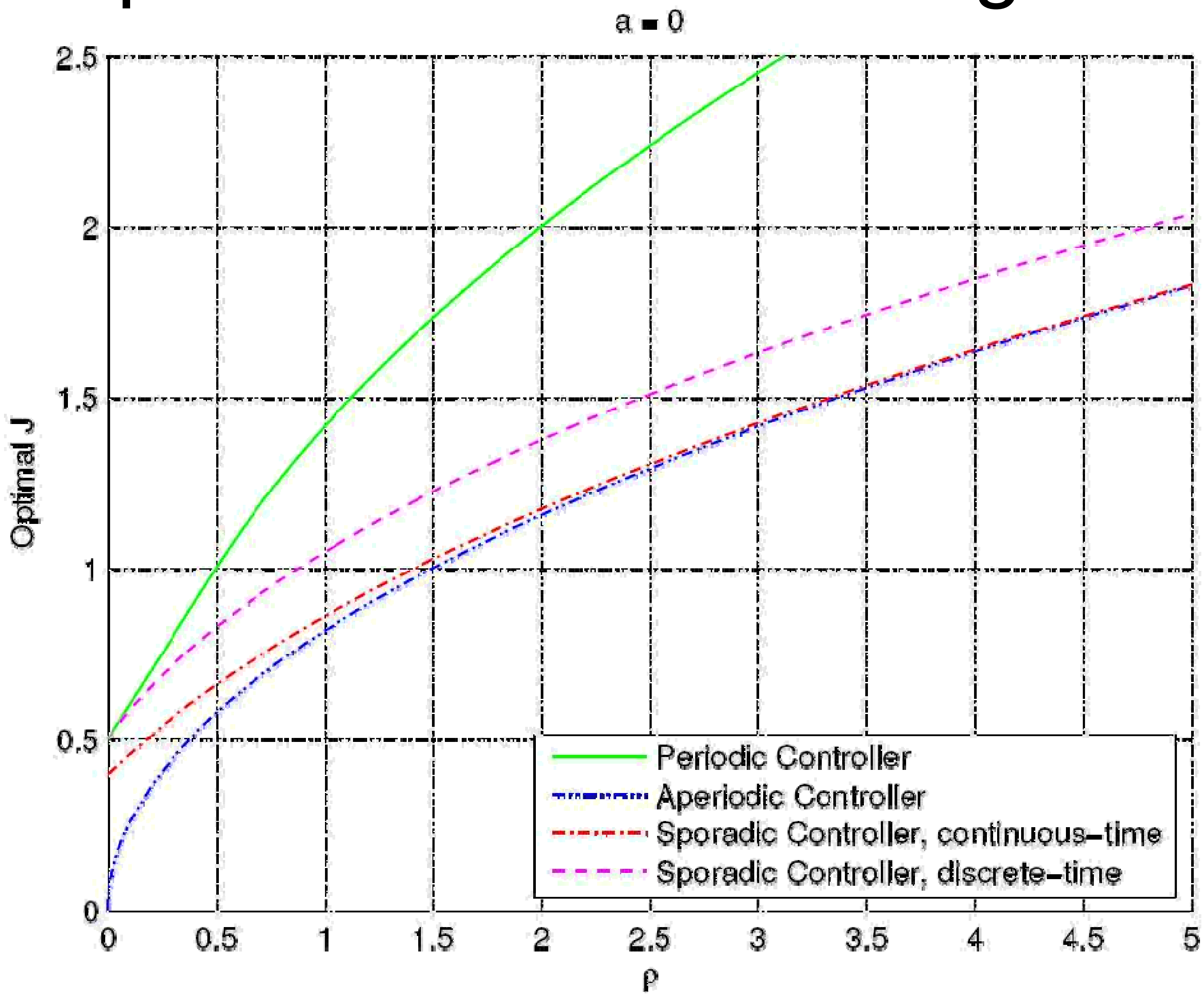
- Average number of events / time unit

Optimal Choice of Threshold under Sporadic Control



- Local minimum for some r > 0
  - For small errors, it is better to wait than to control!

Optimal Cost for the Integrator



Conclusions

- Sporadic control can reduce the process variance and/or the control frequency compared to periodic control
- More realistic than aperiodic control
- Many interesting further problems for higher-order systems
  - When to generate events?
  - What control actions to apply?
  - Event-based observers

Feedback Scheduling

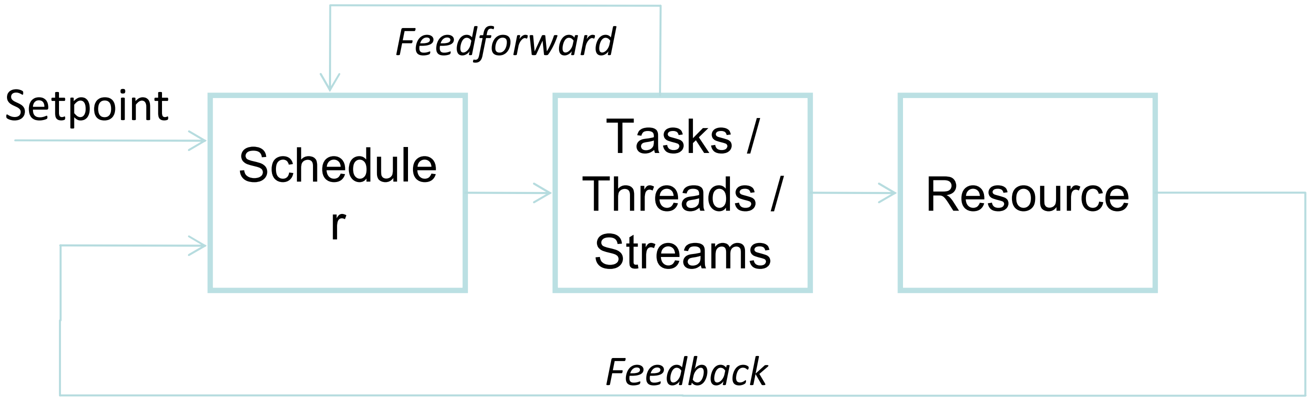
Feedback Scheduling

- Feedback as a technique to manage uncertainty and achieve performance and robustness in computer and communication systems.
- Feedback-based resource management
  - Adaptive or flexible scheduling
  - Dynamic allocation based on
    - measurements of actual resource consumption and comparison with the available resources , or
    - measurements of quality-of-service (QoS) and a comparison with the desired QoS

Why?

- Many applications have time-varying or unknown resource demands
- Static allocation leads to problems
  - Pessimistic → under-utilization (expensive)
  - Optimistic → overload
- Current hardware development makes static worst-case design increasingly difficult
  - Shared memory multi-cores (> 10% of all embedded systems already today)
  - Sub-40 nm chip technology

Structure

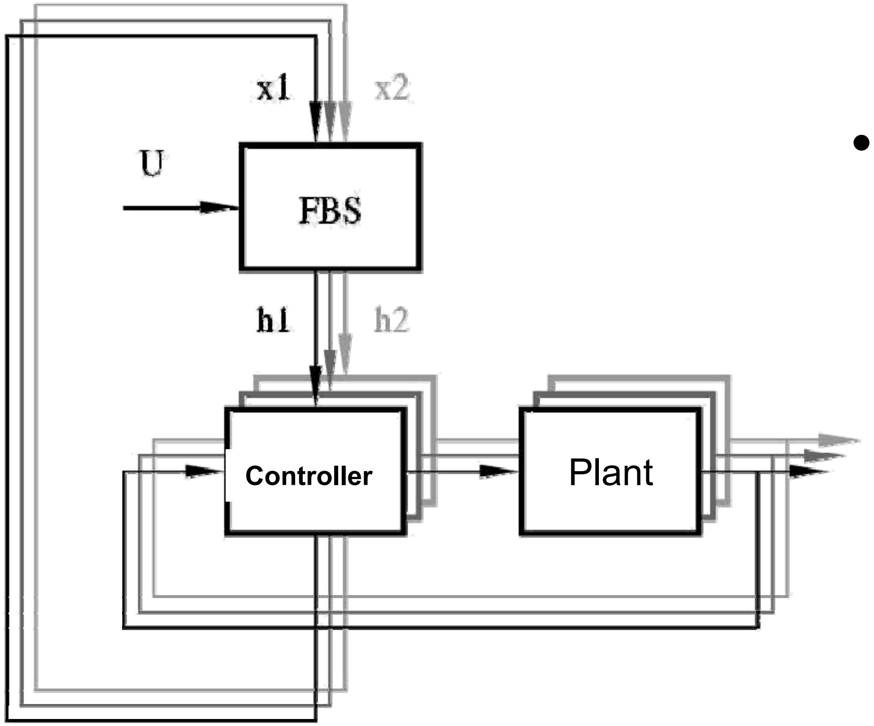


- Feedback to handle uncertainties and disturbances
  - Unknown worst-case resource utilization
  - Load variations
- Feedforward to handle known changes in resource utilization

Potential Problems

- More parameters to tune
  - Bad tuning might lead to stability problem
- Harder to prove anything about the system
- May be harder to program
- The feedback itself consumes resources
- Sensors and actuators not always available
- Which type of models to use?
- Feedback implies errors → Not for hard real-time applications. Or ???
  - But hard RT applications are used to implement feedback systems!

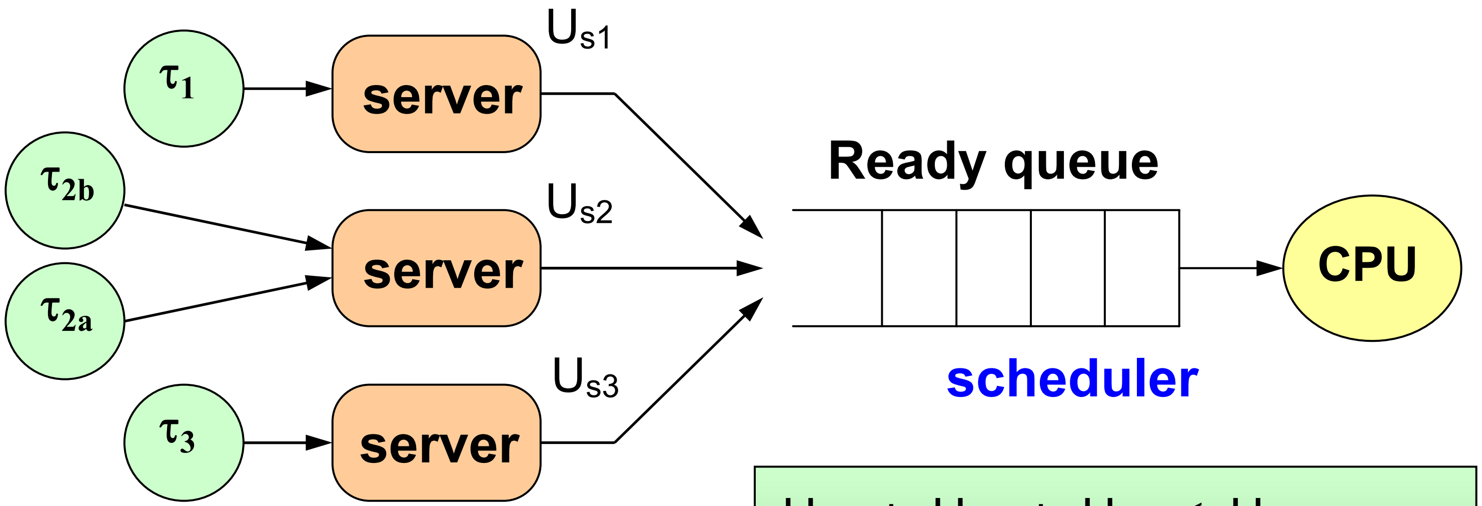
Example: Feedback Control of Control Loops



- A number of control loops in one CPU
- Feedback mechanism to adjust the sampling rates of the controller tasks, so that the global performance is maximized subject to a schedulability constraint on the total CPU utilization

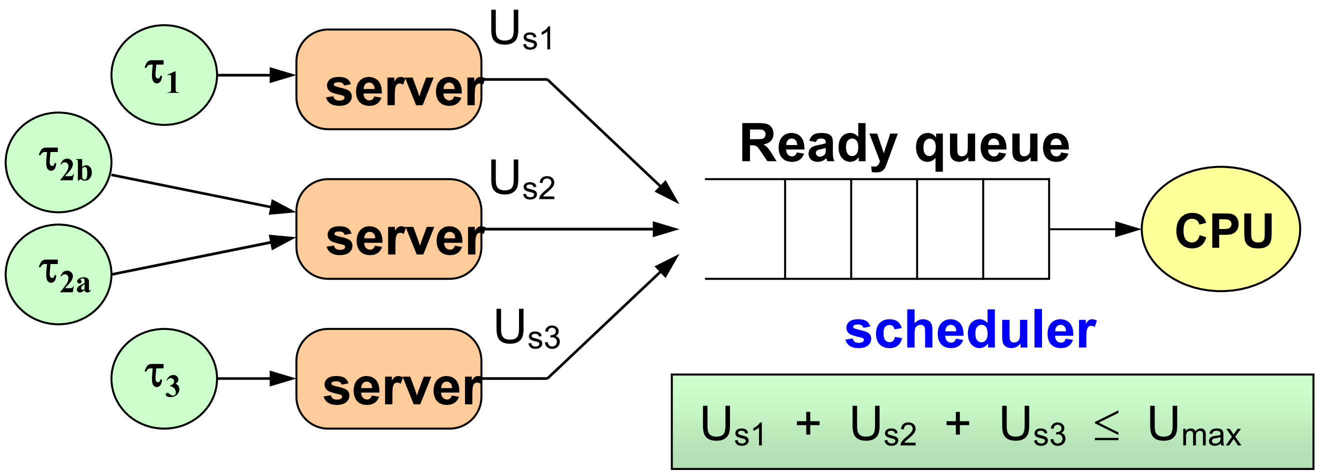
Example: Reservation-Based Resource Management

- Bandwidth server techniques that provide temporal isolation between tasks sharing the same CPU
  - Each reservation is guaranteed a minimum resource budget (e.g. 40%) over a certain time period



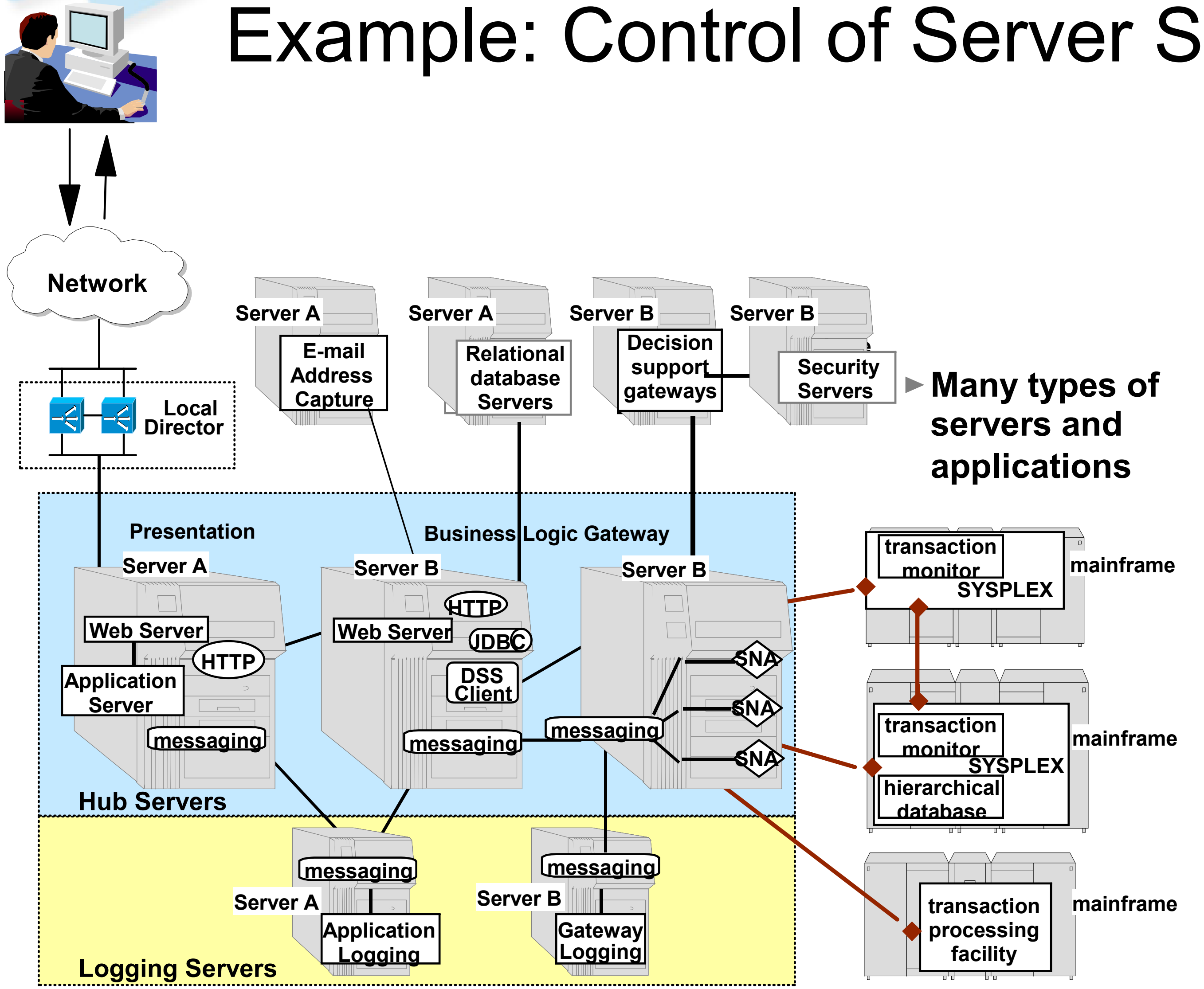
$$U_{s1} + U_{s2} + U_{s3} \leq U_{max}$$

Example: Adaptive Reservations



- Optimization-based global allocation of resources
- Feedback-based adjustment of the budgets of the reservations
- Feedback-based adjustment of the resource utilization within each reservation to meet the allocated resources
- The EU Projects FRESCOR and ACTORS

Example: Control of Server Systems



Front end for online customer service

Back-end Systems

Multi-tier systems of
Web browsers, business logic and databases

Feedback at various levels

Queue Control

IBM, HP, Microsoft, Amazon, ….

Challenges:

- Modeling formalisms (DES, ODEs, queuing theory, …)
- Design of software and computing systems for controllability

Conclusions

Conclusions

- Successful embedded control systems require both control and embedded/real-time competence
- Separations-based or integration-based design approaches
  - Temporal determinism and temporal robustness crucial issues
  - New tools necessary
- The control kernel - an implementation platform tailored for embedded control systems
- Fixed-point implementations can give large savings in small embedded systems and sensor network applications
- Event-based control is a challenging area
- Feedback scheduling for flexibility and adaptivity

Thank you for your attention.

Questions?