Component based approach to real-time and embedded systems

IST-Compare European project

V. Seignole, THALES, France

C. Wigham, Prismtech, UK

A. Radermacher, CEA, France

Who is COMPARE ?

- Collaborative European project funded by European commission
- Running until end of 2006

Outline

- Introduction and motivations
- Considered component approach
- Integration of some real-time techniques with components
- Retargetability of components
- Application to software defined radio
- Conclusions

Motivations

- Traditional difficulties in the real-time and embedded domain
  - Reusability, Modularity, Predictability
  - Scarce computing resources, mature hardware
  - heterogeneous multiprocessor architectures
  - Hard Integration times

- Component orientation for real-time and embedded systems
  - Key to reusability, modularity
  - Key to extensibility

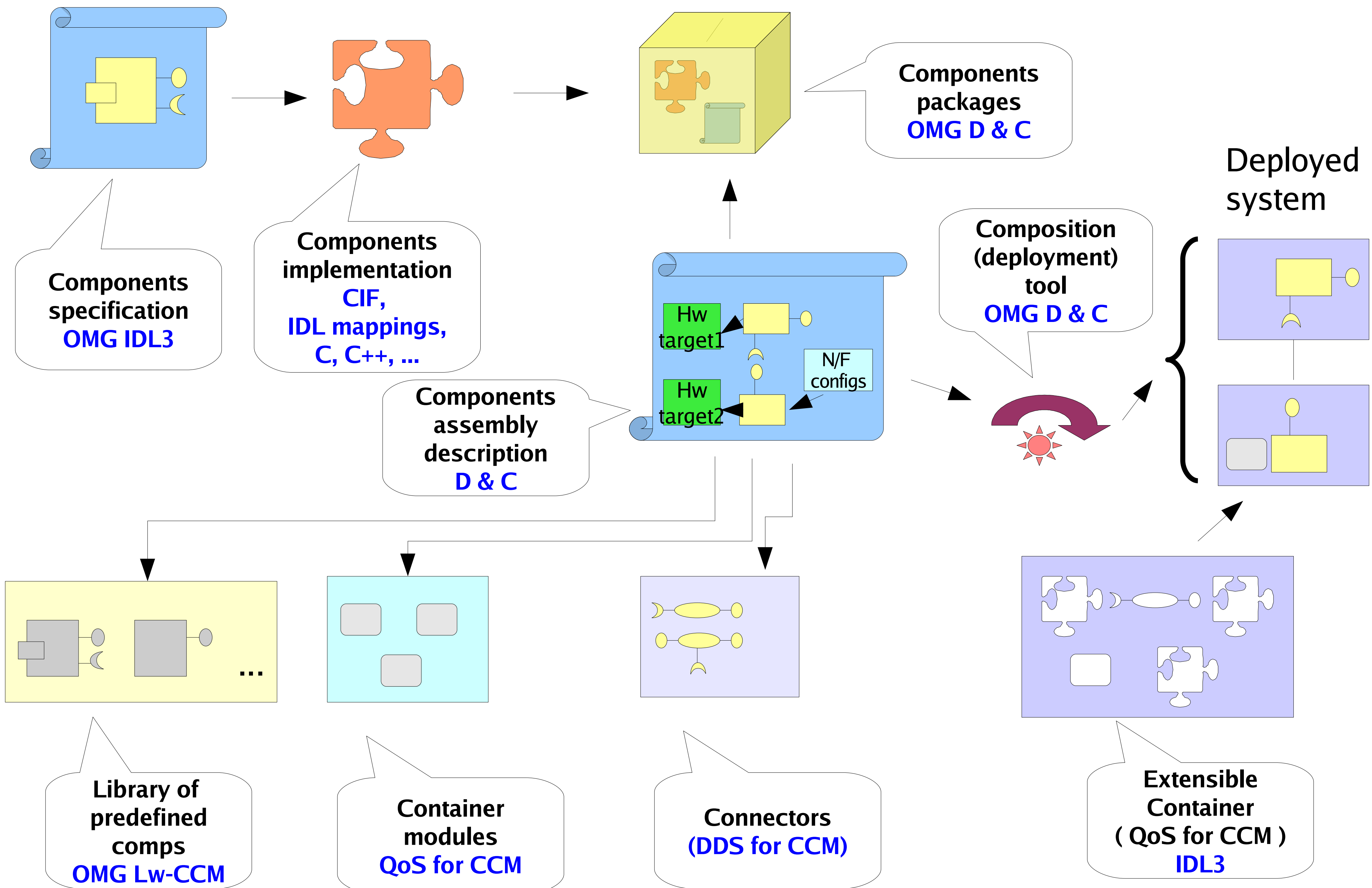
Topics of many studies in software architecture in general.

- Purpose of the talk:
  - Show how open standard component-based approach is extended and applied to fullfill the real-time and embedded domain requirements.

Basic principles on component model

- Component types
  - set of *functional only* provided and used interfaces and attributes
- Mapping implementation rules
  - Define *how to implement* component types
- Composition description
  - Define the system as an *assembly of components*
  - Also contains runtime support *configurations*
- Supporting runtime
  - Allows realisation of the component description, both
    - For composition of *functional* blocks
    - And realisation of *extra-functional* configurations

Technological scope : Lw-CCM and D&C



Modular runtime environment architecture

- The runtime environment (sometimes called container) role is to give life to the hosted component assembly:
  - it manages execution resources
  - supports distribution
  - handles setup of connectors when non-default one is used

- Runtime env modules insert behaviour at « integration points »
  - Corresponding to instants during application lifetime

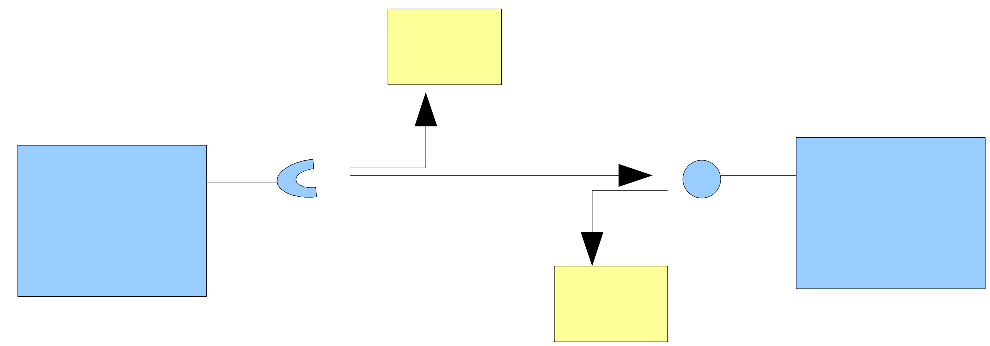
Fairly generic semantics

- Creation of components, Connection of components
- Interaction between components
- Incoming / Outgoing logical flow of control inside components

Runtime integration points principles

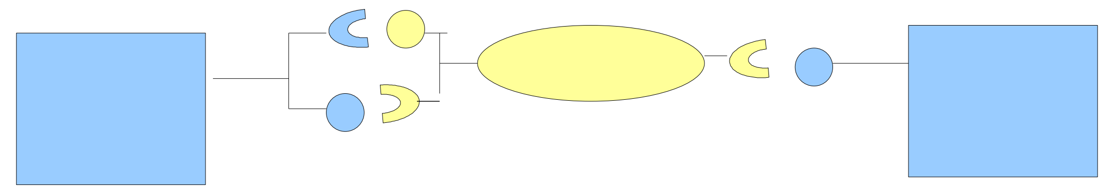
Pre and post Interception

Interceptors:
- can access request context,
- targeted facet ref



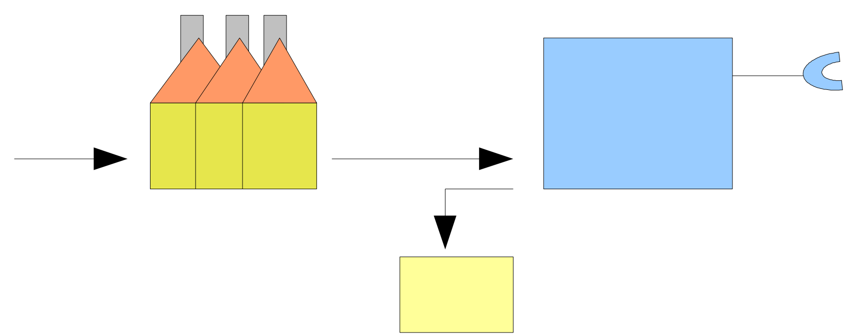
Caller-side pre and callee-side post)

Interactions (connectors)



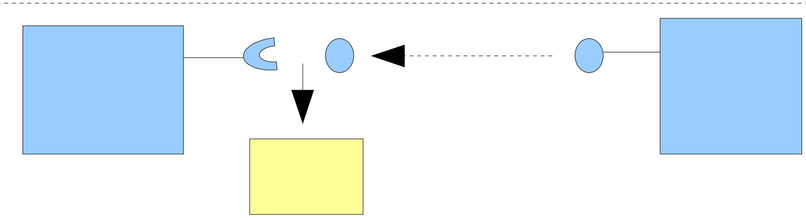
On_creation

(typically used when comp integrated in a framework – e.g. CORBA activation)



On_connection

(typically used to configure transport, or manage per-connection data)

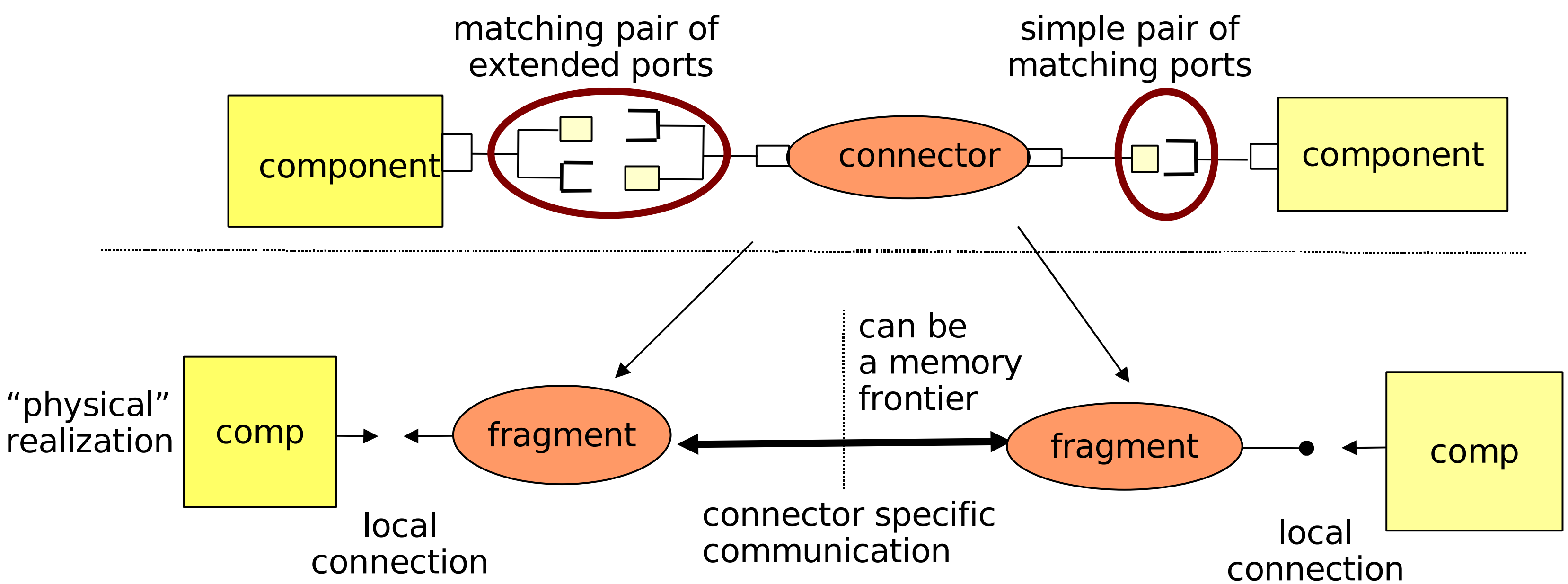


Connectors

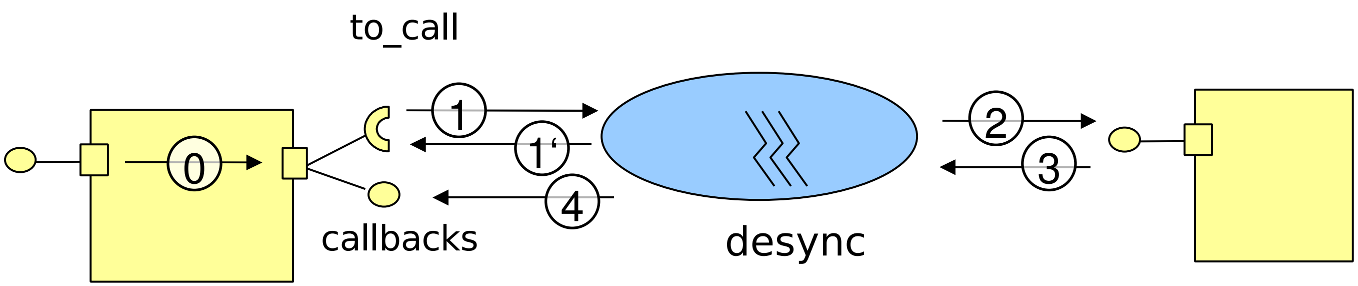
- Component / Connector model
  - Extended port concept
    - Can define the port type needed in case of specific interaction model
    - Can be parameterised by an interface type or event type
    - Extended port as collection of needed / used interfaces (including semantics)
- Component model
  - becomes a little bit closer to UML2 component model
- Connector concept:
  - Stands for interaction entity having some extended ports as well as attributes

Connectors

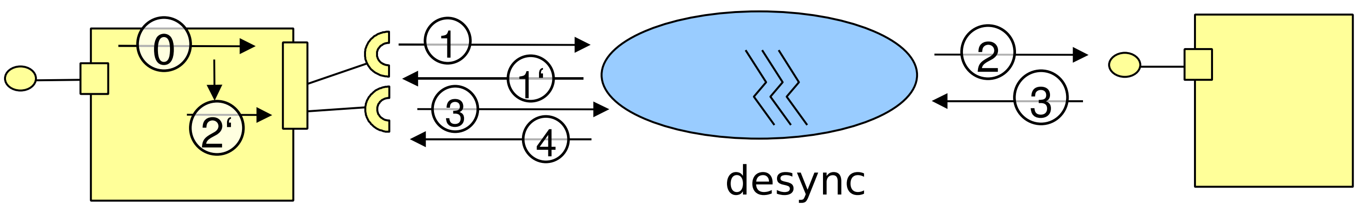
- Allows integration of different architecture patterns, e.g
  - Deferred synchronous method invocation
  - Variants of Pub/Sub
  - Data distribution
  - Streaming
- Also allows specific messaging / transport



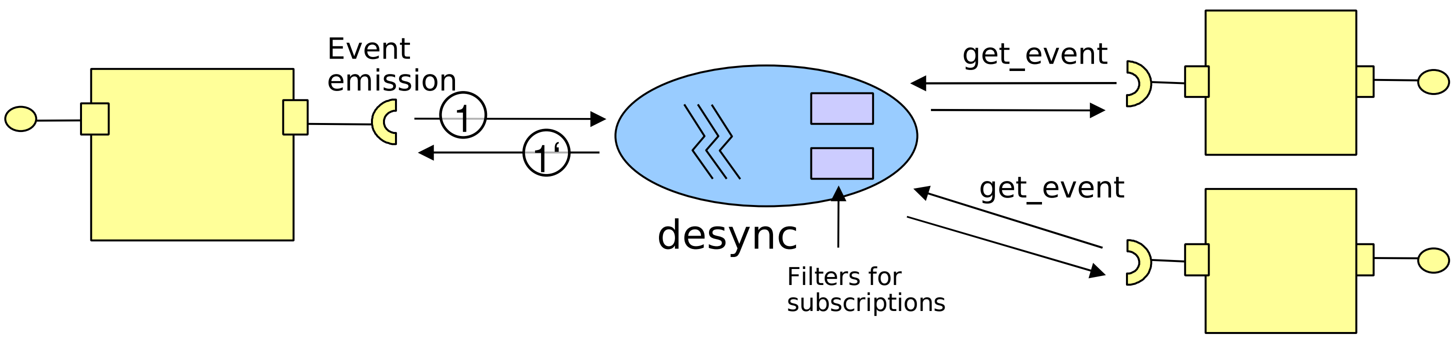
Connectors examples



Deferred synchronous method invocation with Callback



Deferred synchronous method invocation with Polling



Push/Pull event bus with filters

Integrating real-time

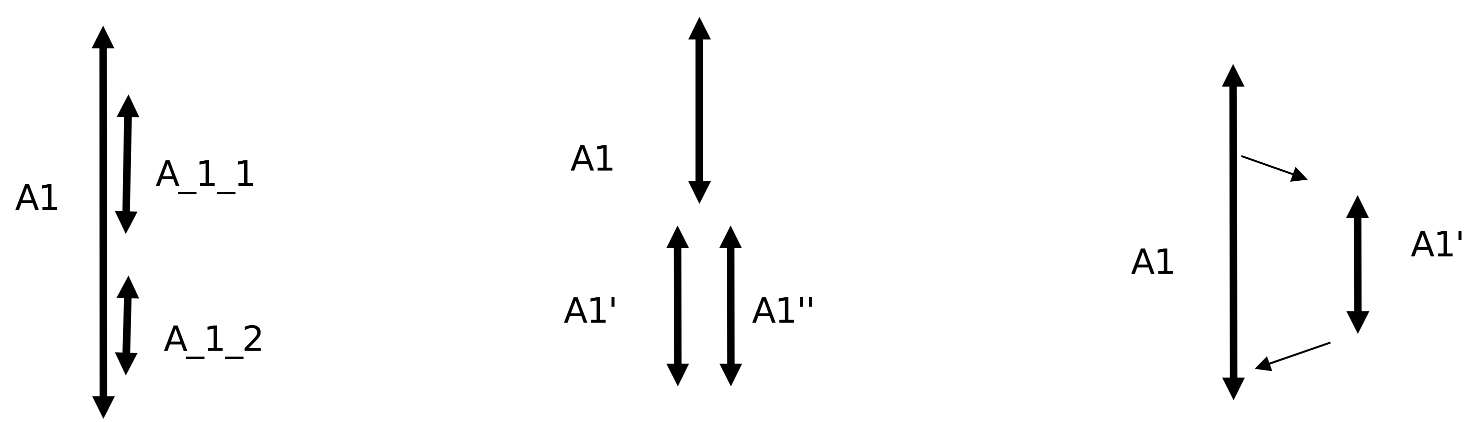
- Key aspects to address:
  - How to make reusable components
    - not tied to real-time execution semantics ?
  - How to make applications based on these components and
    - be able to use different real-time scheduling strategies ?
    - be able to use different concurrency management strategies ?

This is all about finding patterns for separation between real-time techniques and the components, and exploiting modular container to realise them

- Next slides:
  - Mono-processor fixed priority scheduling implementation example
  - Integration of Real-time CORBA capabilities
  - Concurrency management

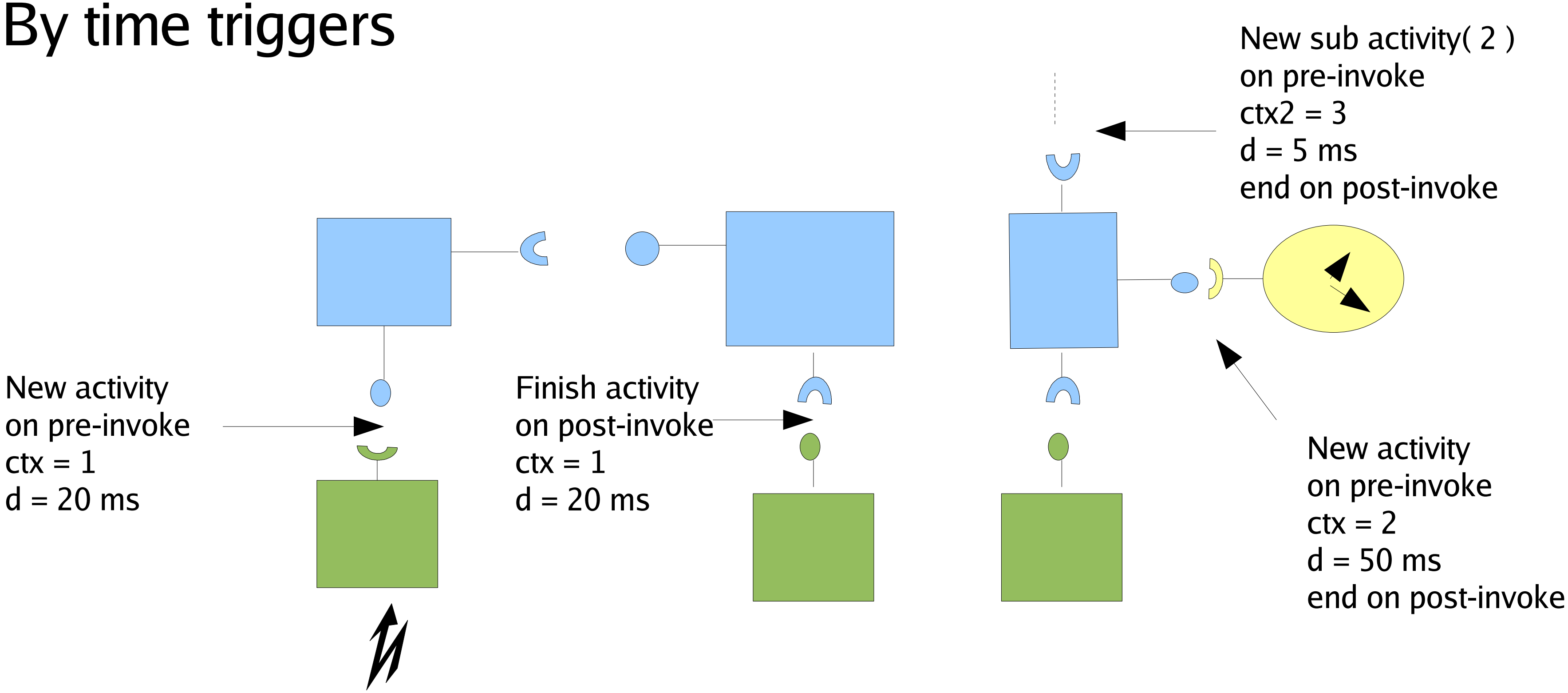
Real-time definitions w.r.t components

- An *activity* is a logical path through a component assembly
- An activity instance is the realisation of a given Activity at a given instant, and under a given context (*e.g* who triggers the activity instance)
- Activity instances can be composed:
  - Nesting of activities instances
  - Chaining of activities instances
  - Forking and joining activities instances



Real-time concepts associated to components

- Activities appear / disappear:
  - typically during process I/O: interaction with predefined comp
  - on functional components interaction time
    - likely to actually be a nested activity
  - By time triggers



Note: the notation used in the example is informal. It is used just to fix concept ideas

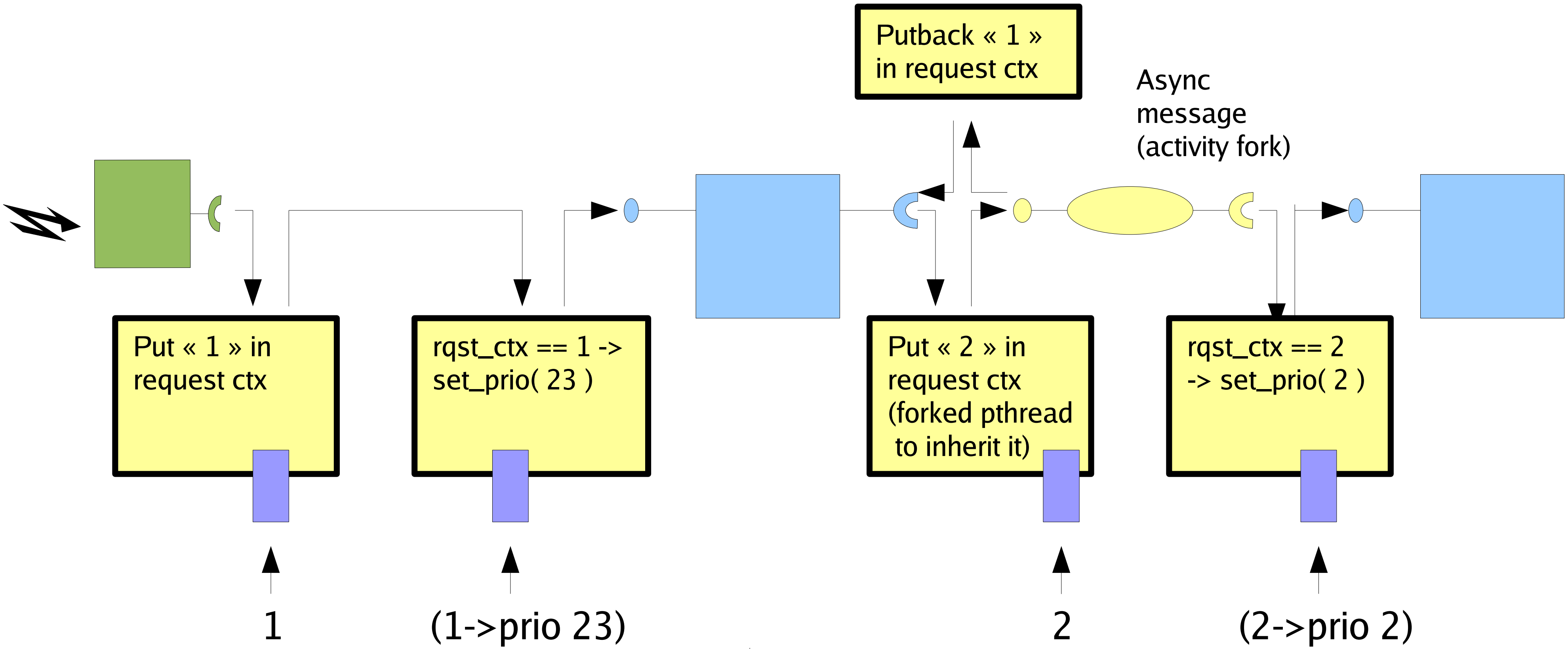
Mono-processor priority scheduling

- Example targeted technology: POSIX Threads
  - Realise activity instances with the following mapping:
    - 1 activity instance == 1 Posix thread
    - Execution threads are managed:
      - by Time triggers, hw encapsulation components, or connectors
    - Activity context propagation done via Posix thread local storage

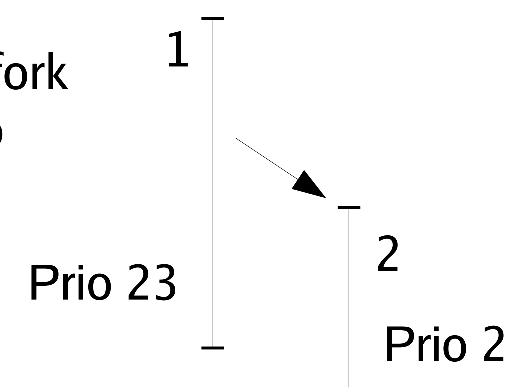
- Result of scheduling analysis:
  - Set of priorities to apply on threads / on threads « segments » under particular context value

- Application of the priorities done through interceptors
  - Based on request context value

Mono-processor priority scheduling (example)



Activity fork scenario



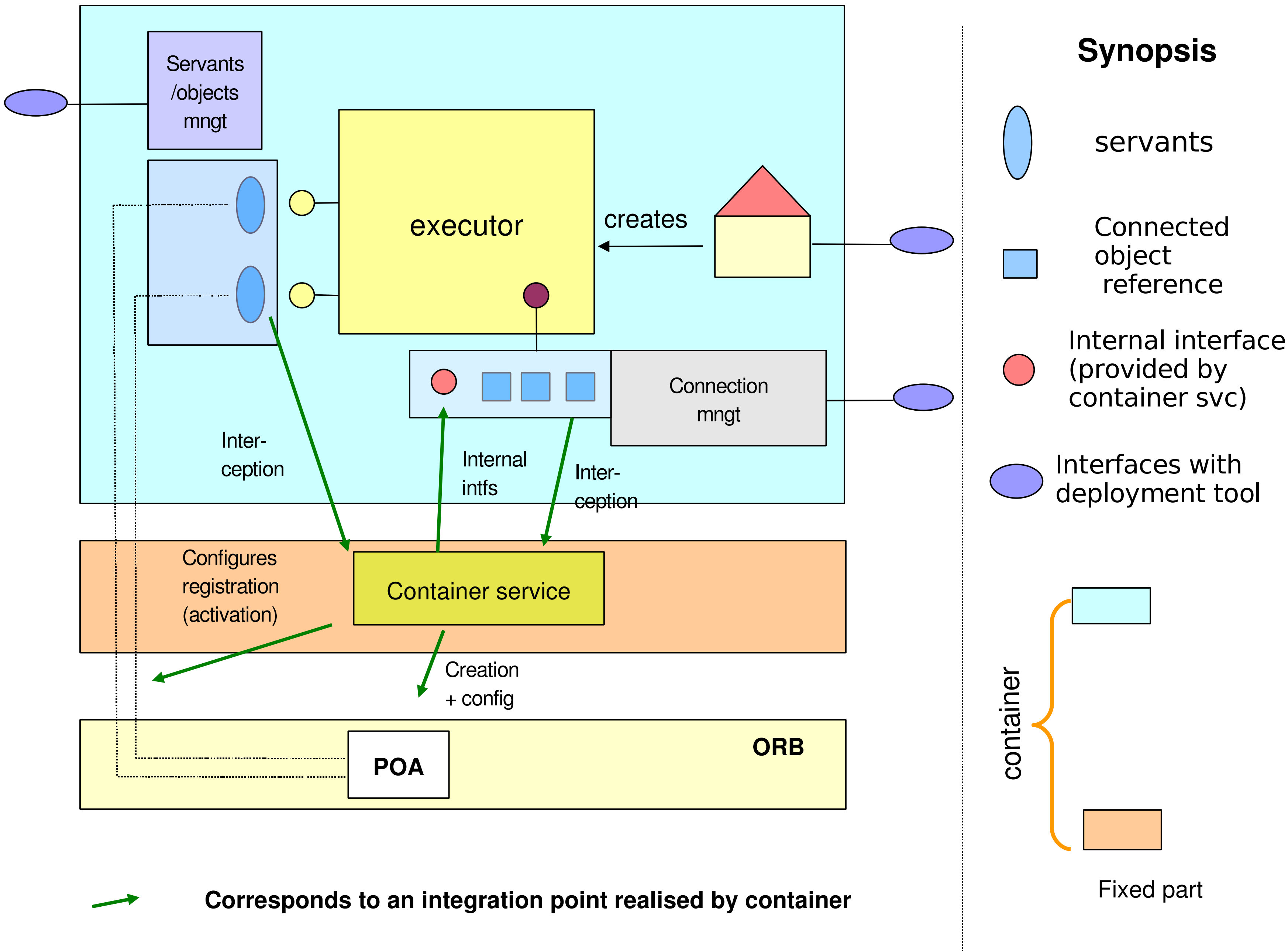
Used POSIX primitives:

```
pthread_create
pthread_key_create /
pthread_set_specific
pthread_attr_getschedparam /
pthread_attr_setschedparam
pthread_getattr
```

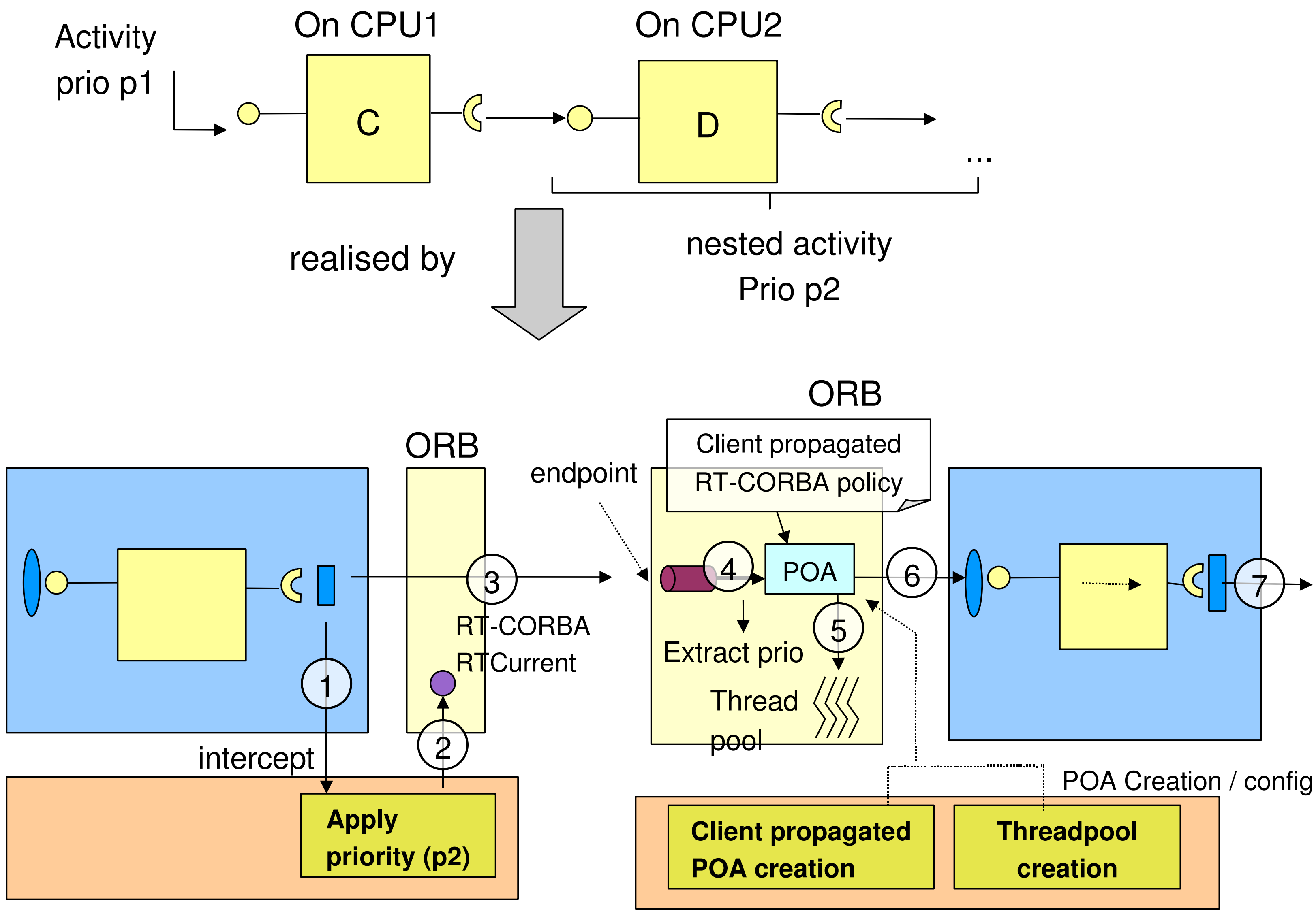
RT-CORBA integration

- Different approach:
  - Runtime resources are managed by the middleware
  - The middleware is already kind of a framework
    - The typical user implements so-called « servants » used as callbacks triggered from the RT-POA
  - The middleware supports some real-time execution semantics
    - Concurrency guarantees (threadpools)
    - Application of priorities (specified at client / server side)
    - Among others, less linked to activity scheduling
      - Transport protocol selection
      - Connections multiplexing, ...

(RT-)CORBA integration : specific code generation



RT-CORBA integration (sketch)



Concurrency management strategies

- RW-Locks associated to component instance
  - Based on method being read or write
  - Specified as accompanying the component implementation
- Software transactional memory
  - The component method as unit of memory transaction
  - Natural composition of memory transactions
  - No priority inversion
  - Clean roll-back to coherent state of component in case in exception is raised

Both techniques implemented via component fwk runtime modules and without impact on components.

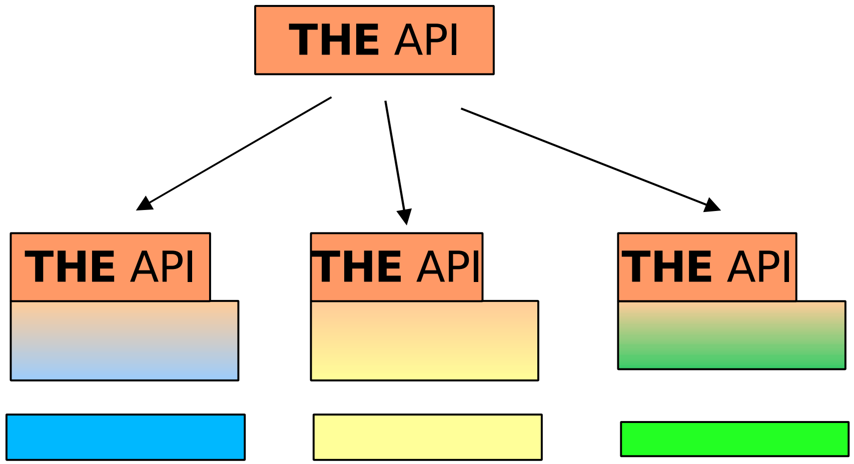
Other added value services

- Periodic triggers
- Performance measurement (WCET exploration) service
- and associated simulation on the host
- Real-time Trace
- Application state management

All implemented as container modules.
Reusable in different applications and integrated in applications
without impact on components

Retargetability of components

Traditional approach for portability



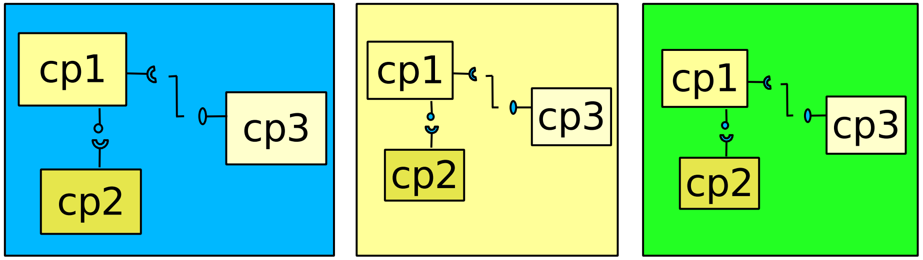
In quest of the definitive API ....

Same interfaces and semantics on different platforms

- Can give headaches to define / implement
- Precludes the usage of specific target (RTOS, mware) added value
- Highly costly to maintain if features coverage is large
- APIs that stay proprietary ... no industrial consensus seems possible

Developed approach

- No portability layer
- puts the focus on integration of technology neutral components
- components embedded in container, implemented with native target mechanisms, and configured in an independent fashion



Realisation of the same component assembly on different target platforms possible via container

Vision: realisation of a native application with components conforming to model

Targetting low-resource processors

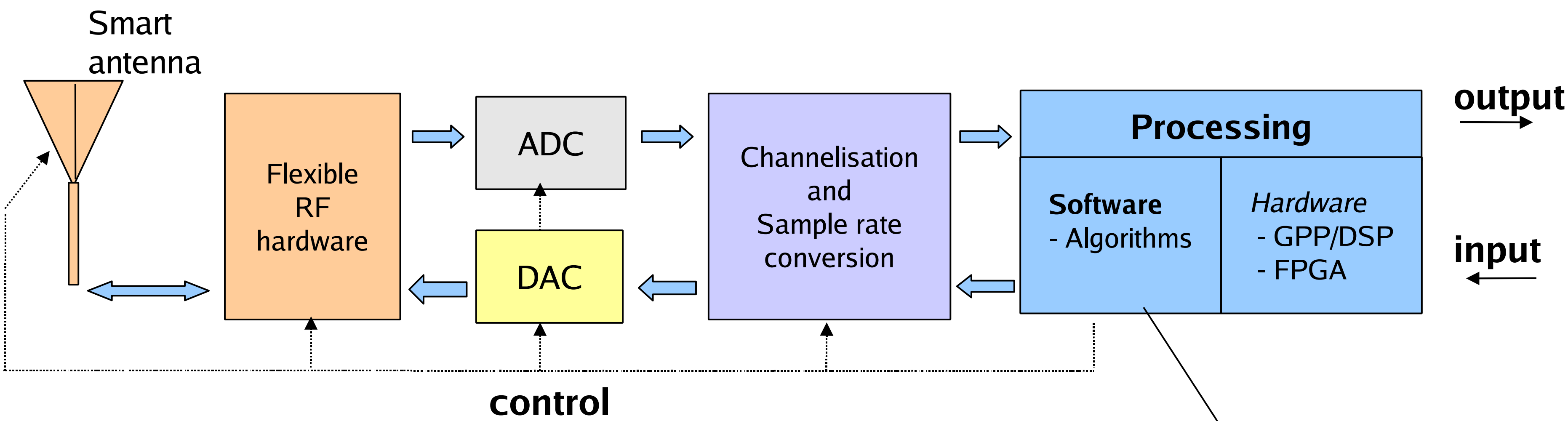
- Tradeoffs considered for very low footprint systems:
  - Dynamic deployment replaced by static deployment
    - Getting rid of generic connection code
    - local (yet compliant) components and connectors as alternative to ORB stubs and skeletons + library
  - application is generated, components statically linked
  - Components following the C mapping are preferably used
    - Natural compactness of the C mapping compared to C++ one
    - Embedded C++ is also used on some targets
  - Modularity of the container is at application build time
  - Smart Inlining techniques can be considered to go further.

Component framework implementation

- Programming languages supported:
  - ISO C++, Embedded C++ and C mapping

- Supported targets combinations:
  - Linux + TAO
  - VxWorks-5.5.1 / PowerPC / Prismtech e*ORB SDR C++
  - OSE-compact kernel / TI C5510 DSP / Prismtech e*ORB SDR C
  - OSEK / ARM7 / orbless / connector on OSEK-Com over CAN
  - OSE epsilon / Coldfire / orbless / mono-processor

Applications – Software defined radio



- Modulation
- Coding
- Medium access control
- protocol layers
- I/Os
- routing functions

- control of (configurable) hardware

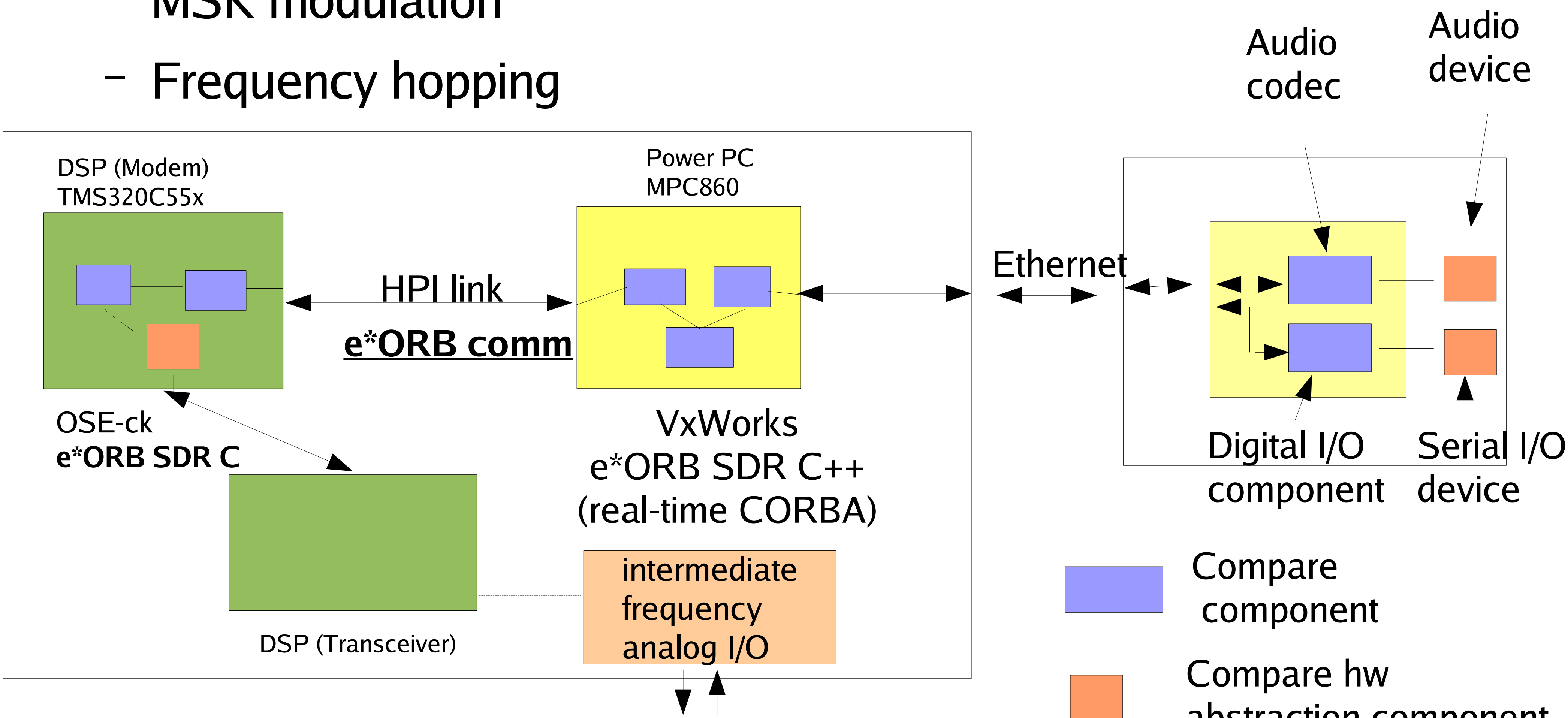
Waveforms

- generic enough hardware
- standard software platform running on the hardware
- ability to control R/F chain in software

Allowing a fully programmable and upgradable radio

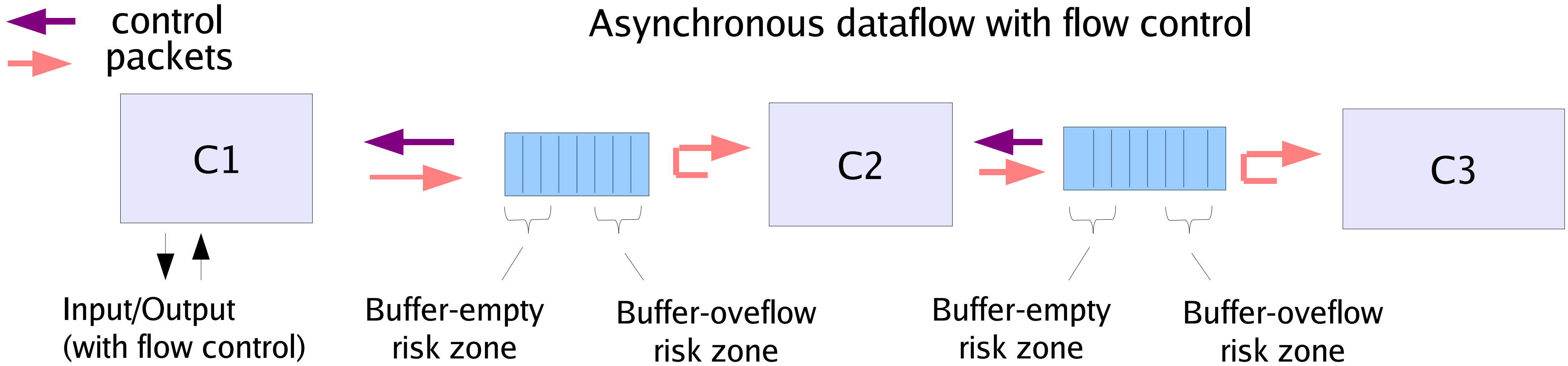
Considered application and targeted platform

- A public test waveform, including
  - Audio, Raw data through Serial I/O
  - Red and Black MAC separated by CSS module
  - MSK modulation
  - Frequency hopping

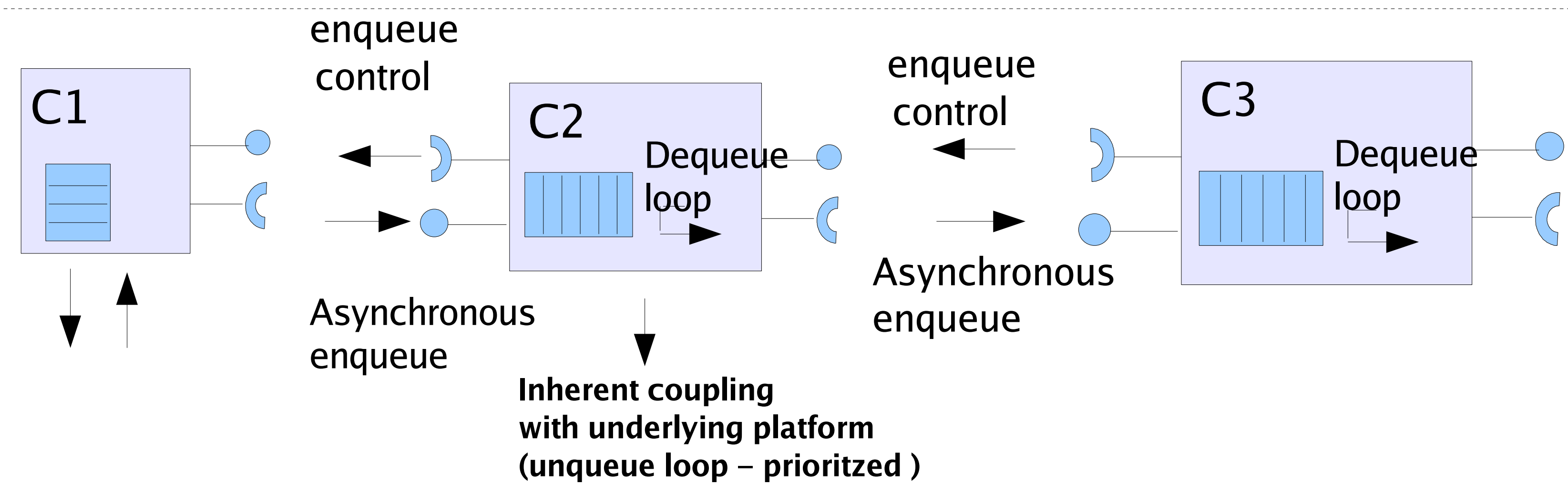


Waveform architecture considerations

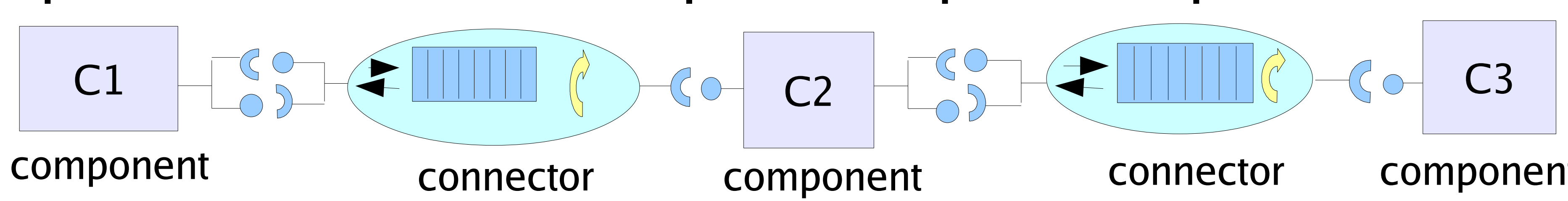
Asynchronous dataflow with flow control



Basic Realisation



Proposed Realisation : reusable and platform independent component



Waveform implementation considerations

- C components used for signal processing (DSP)
- C++ components on the general purpose processors
- Component framework underlying technologies:
  - *e*ORB SDR C* used on the « Modem DSP »
    - Only on-chip RAM of DSP used (160 kb available)
  - *e*ORB SDR C++* used on the GPPs

  Integrated on the board via ETF plugin for HPI
  - Deployment via OMG D&C :
    - Dynamic deployment to the GPPs
    - D&C « Proxy installer » for loading DSP image from GPP side
  - Connector: Asynchronous message connector with flow-control

Conclusions

- Presented an embedded component framework
  - based on open standards (OMG Lw-CCM, D&C)
  - Supporting retargetable components on widely different platforms
  - Allowing components on severely resource-constrained CPUs
  - Providing separation of concerns in the engineering process
  - Natural target platform in the scope of an MDA approach
- *Vision summary:*
  - Synthesis of a component application by various integration techniques (CORBA, Connectors, co-location) fitting nothing-more than just application needs.