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

Contract-based resource reservation and scheduling

Lecturer: Michael González Harbour
Professor
Universidad de Cantabria, Spain

Information Society Technologies

General assumptions of real-time theory

WCETs are measured and enforced

All timing requirements are hard

Industry is familiar with the details of real-time theory

Industry has real-time analysis tools integrated into their design processes

Operating systems provide adequate scheduling services

Is real-time theory useful?

What industry does in reality

- no WCET estimation
- mixture of requirements: hard real-time part is small
- maximum use of the available resources
- no protection or fault detection due to added complexity
- no real-time analysis
  - independently developed components make it difficult
  - timing requirements “proven” by testing
  - “develop and hope for the best” methodology
  - hard real-time analysis is too pessimistic

Is real-time theory useful?

Composition of independently-developed modules makes analysis difficult

Hard real-time scheduling techniques are extremely pessimistic for components with high execution time variability

- for instance, multimedia components

It is necessary to use techniques that let the system resources be fully utilized

- to achieve the highest possible quality

Real-time scheduling theory regarded as "the solution to the wrong problem"

Yes: real-time theory is useful!

Vision:

- real-time scheduling theory is the right solution to the problem,
- but needs proper abstractions
- and needs to be integrated into the design process

Vision: Requirements-based scheduling

Instead of asking the developers to map the application requirements into:

- fixed priorities
- EDF
- aperiodic servers
- timers
- execution-time timers
- complex analysis techniques

Just ask them to specify the application requirements

Vision: Requirements-based scheduling

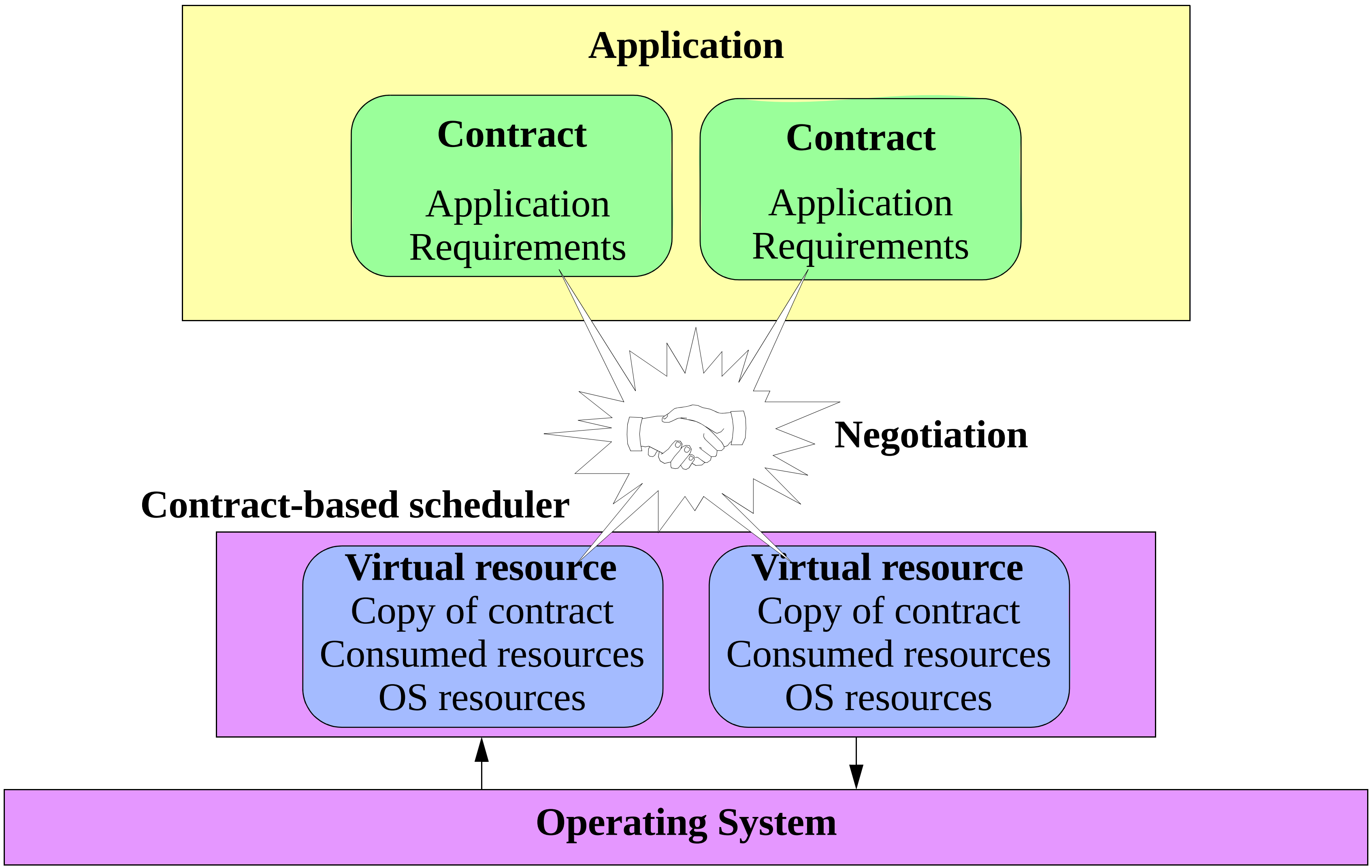
Application developer:

- "tell me what you need"
  - platform independent
- integrated with a component-based design methodology
  - support the composability of independently developed components

System:

- uses the most advanced scheduling methods in real-time theory
  - built-in analysis
  - tells you if the minimum requirements can be guaranteed
  - handles overload in a safe way
- distributes spare capacity to maximize quality
  - high-level quality of service management
- resource reservation and protection
  - processors, networks, memory, disk bandwidth, power, ...

Solution: service contracts



Service contracts (cont'd)

Contract-based scheduling

- Contract specifies:
  - minimum requirements for a given resource
  - how to make use of any spare capacity
  - minimum contract duration
  - minimum stability times
- Online or offline acceptance test
- Spare resources are distributed according to *importance* and *weight*
  - statically,
  - and dynamically
- Renegotiation possible
  - dynamic reconfiguration

Service contracts (cont'd)

Contracts support the following features

- Coverage of application requirements
  - mixture of hard and soft real-time
- Platform independent API
  - independent of OS
  - independent of underlying scheduler
- Support for multiple resources
  - processors, networks
  - memory, energy, disk bandwidth, ...
- Ease of building advanced real-time applications
  - by having time and timing requirements in the API

Contract-based resource reservation and scheduling

2. The FRESCOR approach

FRESCOR: EU IST project in FP6

- Framework for Real-time Embedded Systems based on COntRacts

http://frescor.org

FRESCOR objectives

- Define a contract model that specifies application requirements
  - required to be guaranteed
  - usable to increase quality of service
- Build and underlying implementation manages & enforces contracts: *FRSH*
  - integrated resources (processor, network, power, disk bandwidth, multiprocessor, reconfigurable hardware)
- Adaptive QoS Manager
- Distributed transaction manager
- Performance analysis via simulation
- Component-based framework bridges the gap with design methods
  - tools allow independent analysis
  - tools calculate contract parameters
  - tools obtain timing properties of the overall system
- Test & evaluate on three application domains
  - software-defined radio, multimedia application on a mobile phone, video surveillance

3. Resource Reservation Contracts

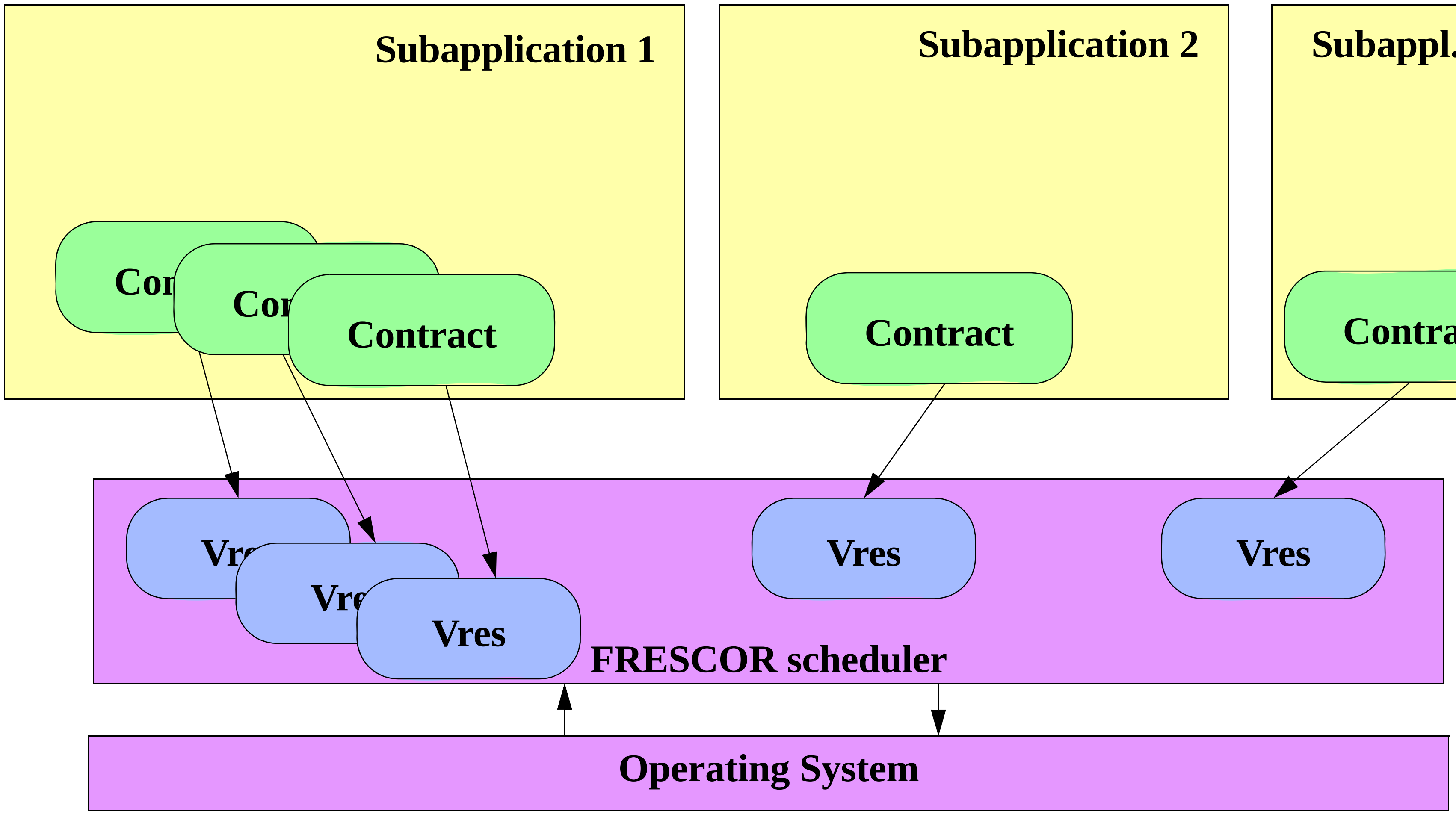
- Contract specifies:
  - minimum requirements: *budget* and *period*
  - how to make use of any spare capacity: other *budgets* and *periods*
- QoS Parameters
  - spare resources are distributed according to *importance* and *weight*
- Deadlines
  - Max interval to receive the reserved budget
- Support for multiple resources through a resource type
  - CPU
  - network
  - memory (special type of contract)
  - disk bandwidth
- Resource usage under different power levels

Resource Reservation Contracts

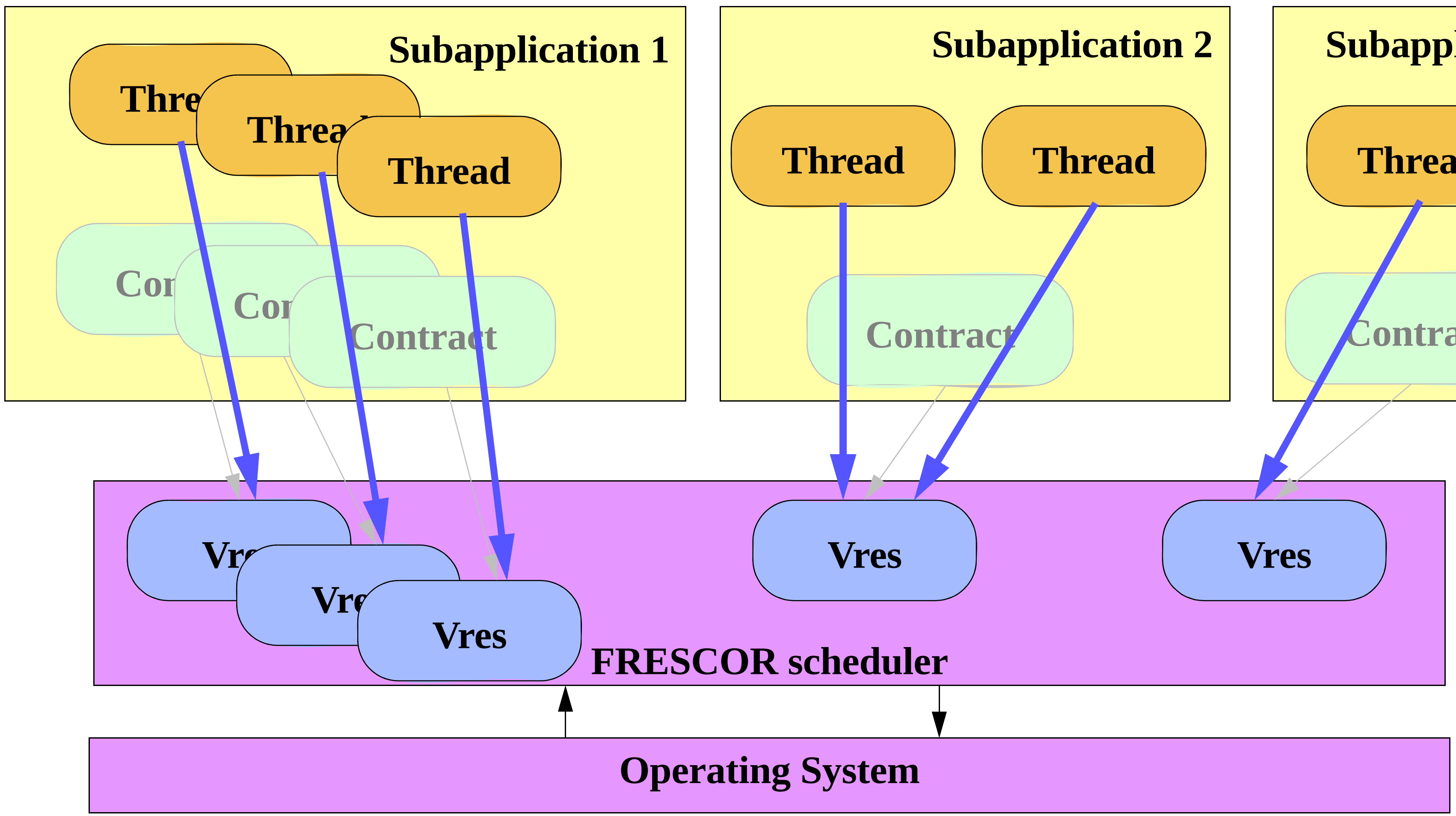
Features

- API is independent of underlying scheduler
- Ease of building advanced real-time applications
  - by having time and timing requirements in the API
- Renegotiation possible

The application model: negotiation



The application model: binding threads to vres



FRESCOR modules

- Core
- Dynamic reclamation
- Implementation Specific
- Shared resources
- Memory management
- Hierarchical scheduling
- Spare capacity sharing
- Energy management
- Distributed
- Disk bandwidth
- Feedback control

4. Core services

Minimum resource reservation

- Minimum budget
- Maximum period
- Vres deadline

Workload type, specifying different job models

- *indeterminate* (may use all extra time assigned)
- *bounded*: stream of jobs (may return unused capacity)
- *synchronized* with the virtual resource period

Different contract types

- *regular*, for tasks with resource requirements
- *background*, for non-real-time tasks
- *dummy*, to account for overheads

Core Attributes

Attribute	Data
Label	global id (string)
Resource type	processor, network, memory
Resource id	number
Minimum budget	$C_{i,min}$
Maximum period	$T_{i,max}$
Workload	Bounded / Indeterminate /Synchronized
D=T	Yes / No
Deadline	vres D_i
Budget overrun signal	Signal number and data
Deadline miss signal	Signal number and data
Type of contract	regular, background, dummy

FRESCOR API: Core

Contract Creation and Initialization

contract_init
contract_set(get)_basic_params
contract_set(get)_resource_and_label
contract_set(get)_timing_reqs

Negotiation and Binding Functions

contract_negotiate
contract_cancel
contract_renegotiate (synch, asynch)
vres_get_renegotiation_status
vres_get_contract
thread_create_and_bind
thread_create_in_background
thread_bind
thread_unbind
thread_get_vres_id

Obtaining information from a Vres

vres_get_usage
vres_get_remaining_budget
vres_get_budget_and_period
vres_get_job_usage

Group Contract Negotiation

group_negotiate
group_change_mode (synch, asynch)

General Management

init
service_thread_set(get)_data
config_is_admission_test_enabled
resource_get_vres_from_label

FRESCOR API: Core (cont'd)

Synchronization objects

synchobj_create
synchobj_destroy
synchobj_signal
synchobj_wait
synchobj_wait_with_timeout
timed_wait
vresperiod_wait
vres_get_period

Example: Periodic task with budget and deadline control

With OS API

```
set priority
create budget signal handler
create deadline signal handler
create budget timer
create deadline timer

while (true) {
   reset deadline timer
   set budget timer
   do useful things
   reset budget timer
   set deadline timer
   wait for next period
}
```

With FRSH API

```
create contract with {C,T}
negotiate the contract
while (true) {
   do useful things
   frsh_timed_wait
}
```

5. Advanced services
5.1 Shared objects

Two kinds of shared objects

- *unprotected*: trusted, guaranteed WCET for critical sections
  - no monitoring mechanism, analysis assumes WCETS are OK
- *protected*: good estimates, but no guaranteed WCET for critical sections
  - monitoring mechanism
  - rollback mechanism when budget is overrun

Critical sections can be

- *unchecked*: WCET not monitored, but used for analysis
- *read*: WCET enforced; can be interrupted with no consequences
- *write*: WCET enforced; they require a rollback mechanism
  - save the part of the object that will be written
  - restore it if necessary

Shared objects attributes

Attribute	Data
List of critical sections	Set of {Critical_Section}

A critical section may have:

- reference to shared object
- WCET
- kind of operation
- memory areas to be saved and restored

Shared objects API

Shared Objects	Critical sections
contract_set(get)_csects sharedobj_init sharedobj_get_handle sharedobj_get_mutex sharedobj_get_obj_kind sharedobj_remove	csect_init csect_destroy csect_get_sharedobj_handle csect_get_wcet csect_register(get)_read_op csect_register(get)_write_op csect_get_op_kind csect_invoke csect_get_blocking_time csect_register_thread csect_deregister_thread

5.2. Spare capacity and dynamic reclamation

Granularity: two ways of specifying how to make use of extra resources available

- continuous range between a minimum and a maximum, for budget and period
- discrete sets of budgets, periods and deadlines

Stability times

- minimum interval during which assigned resources must be guaranteed
  - to avoid fast interactions with control loops
  - to avoid fast changes in the perception of quality obtained by the user

Spare capacity attributes

Attribute	Data
Granularity	Continuous or Discrete
Maximum budget	$C_{i,max}$
Minimum period	$T_{i,min}$
Discrete Utilization Set	Set of {C,T,D} ordered by utilization
Weight	W_i (relative weight)
Importance	I_i (absolute importance)
Stability time	Minimum stability time for assigned resources

Spare Capacity API

Spare Capacity

contract_set(get)_reclamation_params
resource_get_capacity
resource_get_total_weight
vres_set_stability_time
vres_get_remaining_stability_time
vres_drecrease capacity

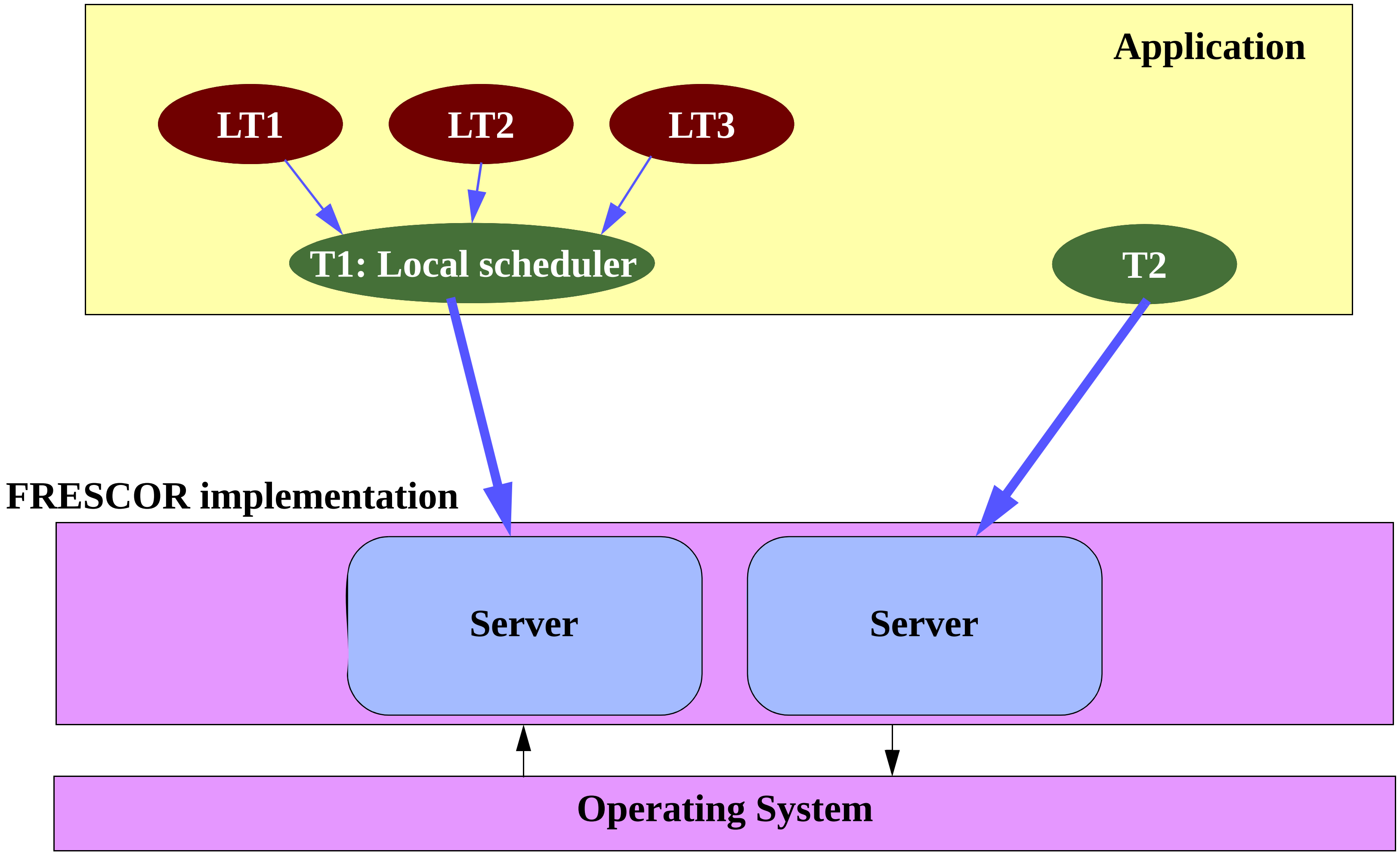
Example: 3-version algorithm

```
create contract with
   {discrete {C1,T},{C2,T},{C3,T}}
negotiate the contract
while (1) {
   if (current_budget<C2) {
         do_version_1
   } else if (current_budget<C3) {
         do_version_2
   } else {
         do_version_3
   }
   frsh_timed_wait
}
```

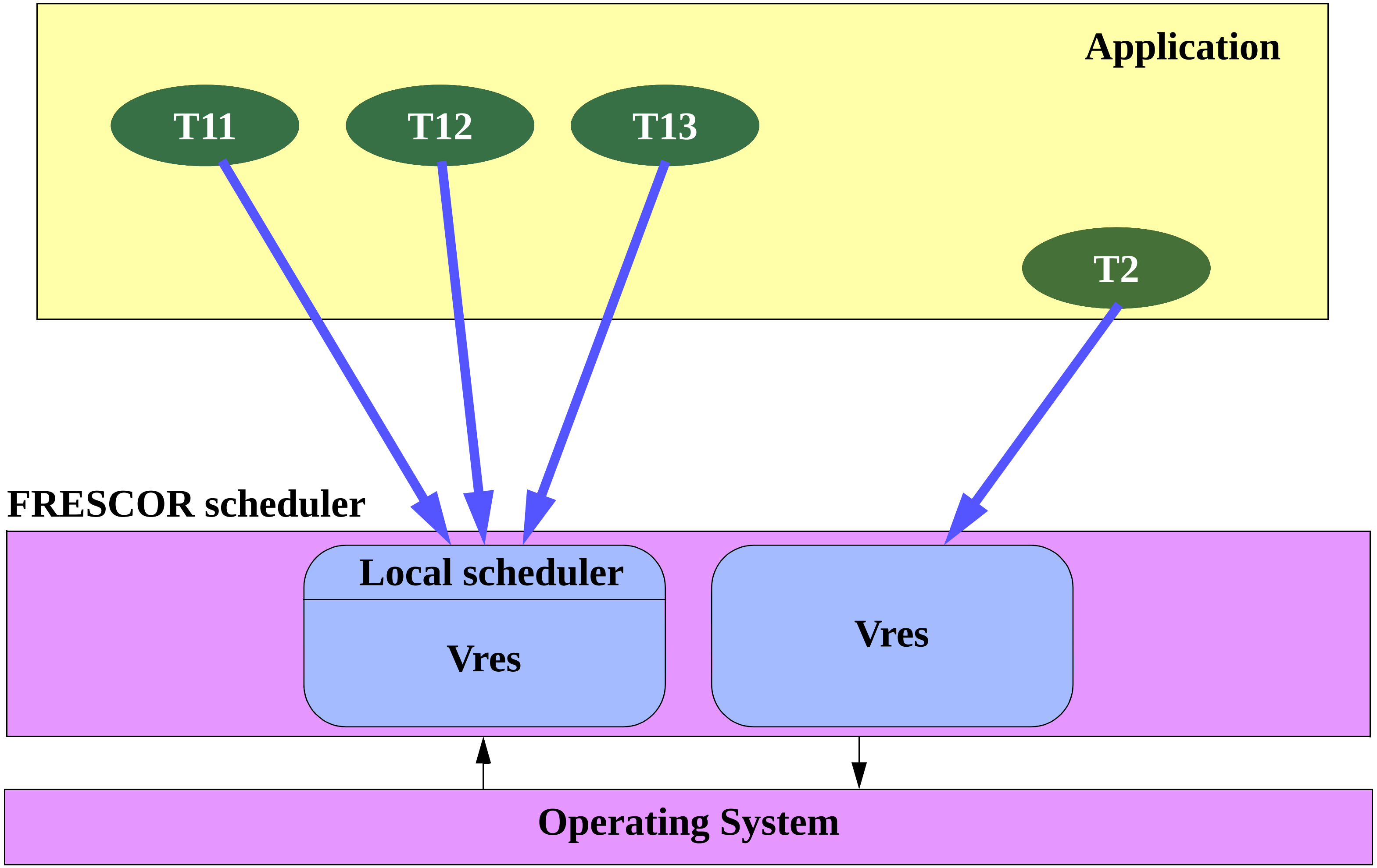
Example: anytime algorithm

```
create contract with
   {continuous {Cmin,T},{Cmax,T}}
negotiate the contract
while (1) {
   while (current_budget enough for one loop) do {
         refine solution
   }
   output result
   frsh_timed_wait
}
```

5.3. Hierarchical scheduling: library level



5.4 Hierarchical scheduling: "kernel" threads



Hierarchical scheduling attributes

Attribute	Data
Scheduling policy	none, fixed_priority, EDF, table_driven
Scheduler init info	for table driven: table with the schedule

Hierarchical scheduling API

Hierarchical scheduling
contract_set(get)_sched_policy local_scheduler_init thread_create_local thread_set(get)_local_sched_params thread_bind_local

5.5. Feedback control

Contracts represent "slow" resource reservations

- negotiation is complex
- a more dynamic mechanism can be used additionally

QoS manager uses a feedback control algorithm to reallocate budgets assigned to virtual resources

- the objective is to maximize the quality perceived by the user

A percentage of utilization is allocated to the QoS manager

- through a regular contract: the spare bucket

The QoS manager may allocate additional budget to its virtual resources

- according to their specified desired budget

Feedback control API

No specific parameters

Reservation of a spare bucket through a regular contract

Control policies and parameters through specific API

Operations:

Feedback control
feedback_set(get)_spare feedback_set(get)_desired_budget feedback_get_actual_budget

5.6. Memory management

Memory is a scarce resource that can be shared and influences quality of service

A memory contract can specify a minimum memory requirement

Spare capacity distribution can be applied to memory resources

- using the FRESCOR spare capacity parameters
  - importance and weight
  - stability times
- the resource is specified as min-max memory
  - not budget & period

Memory management API

Attribute	Data
Label	global id (string)
Resource type	processor, network, memory
Resource id	number
Minimum memory	**Mmin**
Maximum memory	M_{max}
Weight	W_i (relative weight)
Importance	I_i (absolute importance)
Stability time	Minimum stability time for assigned resources

Memory management API

Memory contract management

contract_set(get)_min_memory
contract_set(get)_max_memory

Allocate memory from a vres

vres_get_memory_reqs
vres_memalloc
vres_memfree

5.7. Energy management

Energy is a global resource associated with a particular platform

- initial focus is on execution platforms

Discrete power levels

- as worst-case execution times do not scale linearly

For each power level we need to specify in the contract

- budgets
- worst-case duration of critical sections

An API can be used to switch to a new power level

- the request may be rejected; spare capacity may be reassigned

Battery duration as another resource

- minimum battery expiration time is part of contract

Energy management API

Attribute	Data
Minimum expiration	time
Minimum budget per power level	array[power level] of budget
Maximum budget per power level	array[power level] of budget
Utilization set per power level	array[1-N] of utilization levels

A utilization level is a triple

- {array[power level] of budget, period, deadline}

Energy management API

Contract parameters

contract_set(get)_min_duration
contract_set(get)_min_budget_pow
contract_set(get)_max_budget_pow
contract_set(get)_utilizations_pow

Critical section parameters

csect_set(get)_wcet_pow

Battery duration

resource_get_battery_expiration

Managing the power level

resource_set(get)_power_level
resource_num_power_levels

5.8 Hard disk bandwidth

Initial attempt to use FRESCOR contracts to reserve disk bandwidth

- changes to disk access functions in the OS
- new type of resource: **FRSH_RT_DISK**
- no new APIs

Preliminary work being done

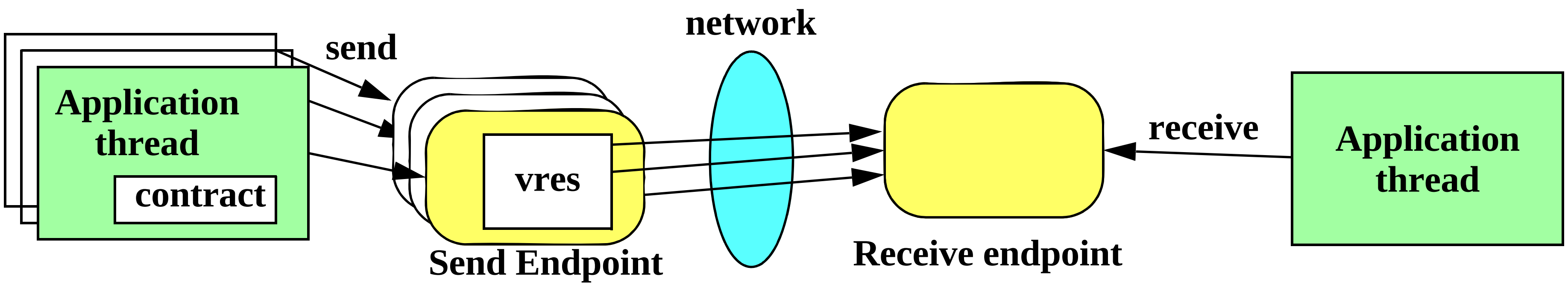
- The objective is to gain experience

6. Distribution

A network contract is specified for a resource of type network

Define specific communication mechanism

- send endpoints
  - they are bound to a network vres
  - they keep track of consumed bandwidth
  - they are connected to one or more receive endpoints, through the destination_id & stream_id
- receive endpoints



Send and receive endpoints

Send endpoint

- Object used to send messages of a particular stream id, through a given network, to a given destination id
- It is bound to a network vres
- Attributes
  - queue-size
  - rejection policy: new, oldest, next-newest

Receive endpoint

- Object used to receive messages of a particular stream id, through a given network
- Attributes
  - queue-size
  - rejection policy: new, oldest, next-newest

Distributed attributes

Attribute	Data
protocol-dependent information	parameters used to negotiate the contract for a particular network protocol
queuing info	Size and rejection policy (oldest, newcomer) of queue used to send messages

Distributed API

Distribution: basic

contract_set(get)_protocol_info
contract_set(get)_queueing_info

Distribution: receive endpoints

receive_endpoint_create
receive_endpoint_destroy
receive_endpoint_get_status
receive_endpoint_get_params

Distribution: send endpoints

send_endpoint_create
send_endpoint_get_params
send_endpoint_get_status
send_endpoint_destroy
send_endpoint_bind
send_endpoint_unbind
send_endpoint_get_vres

Distribution: information

network_bytes_to_budget
network_budget_to_bytes
network_get_message_max_size
network_get_min_effective_budget

Distributed API (cont'd)

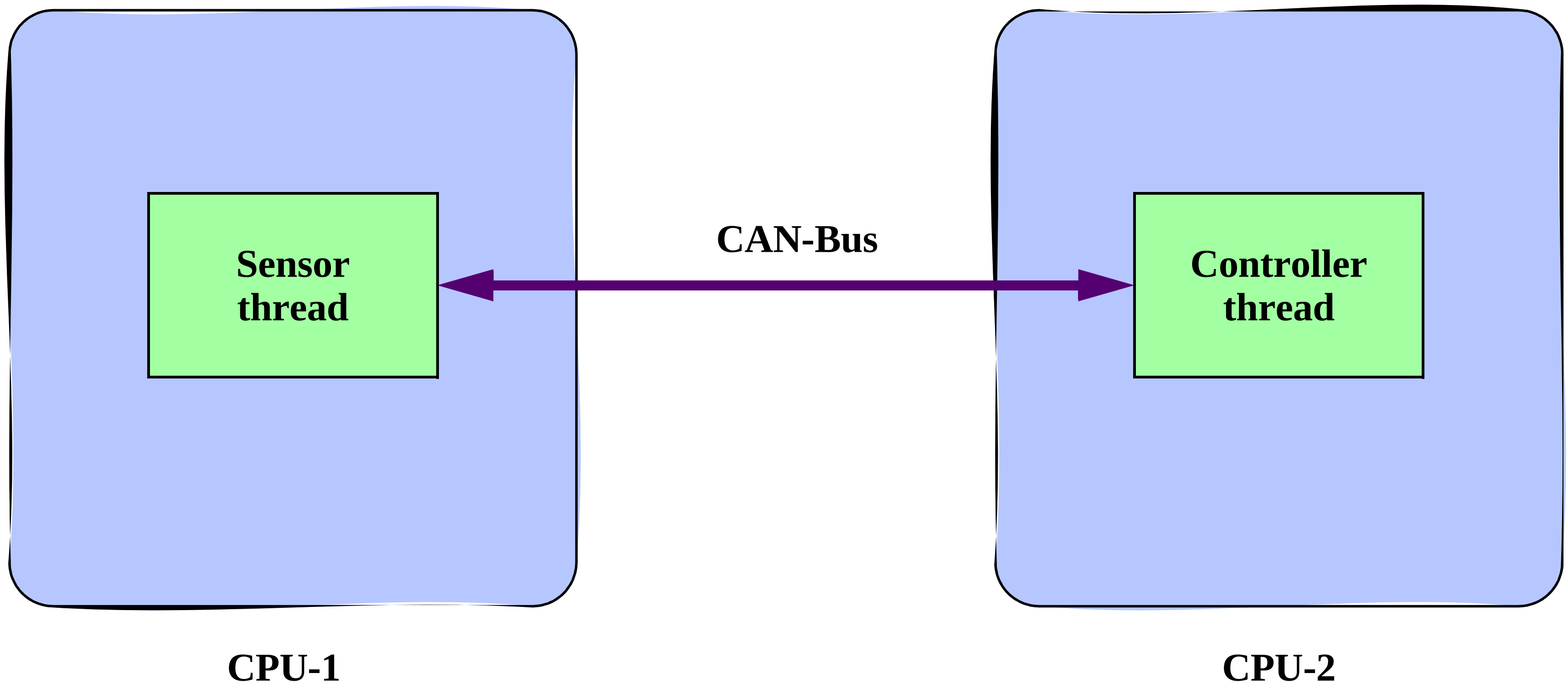
Two-step contract negotiation

contract_negotiate_reservation
vres_commit_reservation

Distribution: send & receive

send_async
send_sync
receive_async
receive_sync

Example: Distributed sensor



Example: implementation

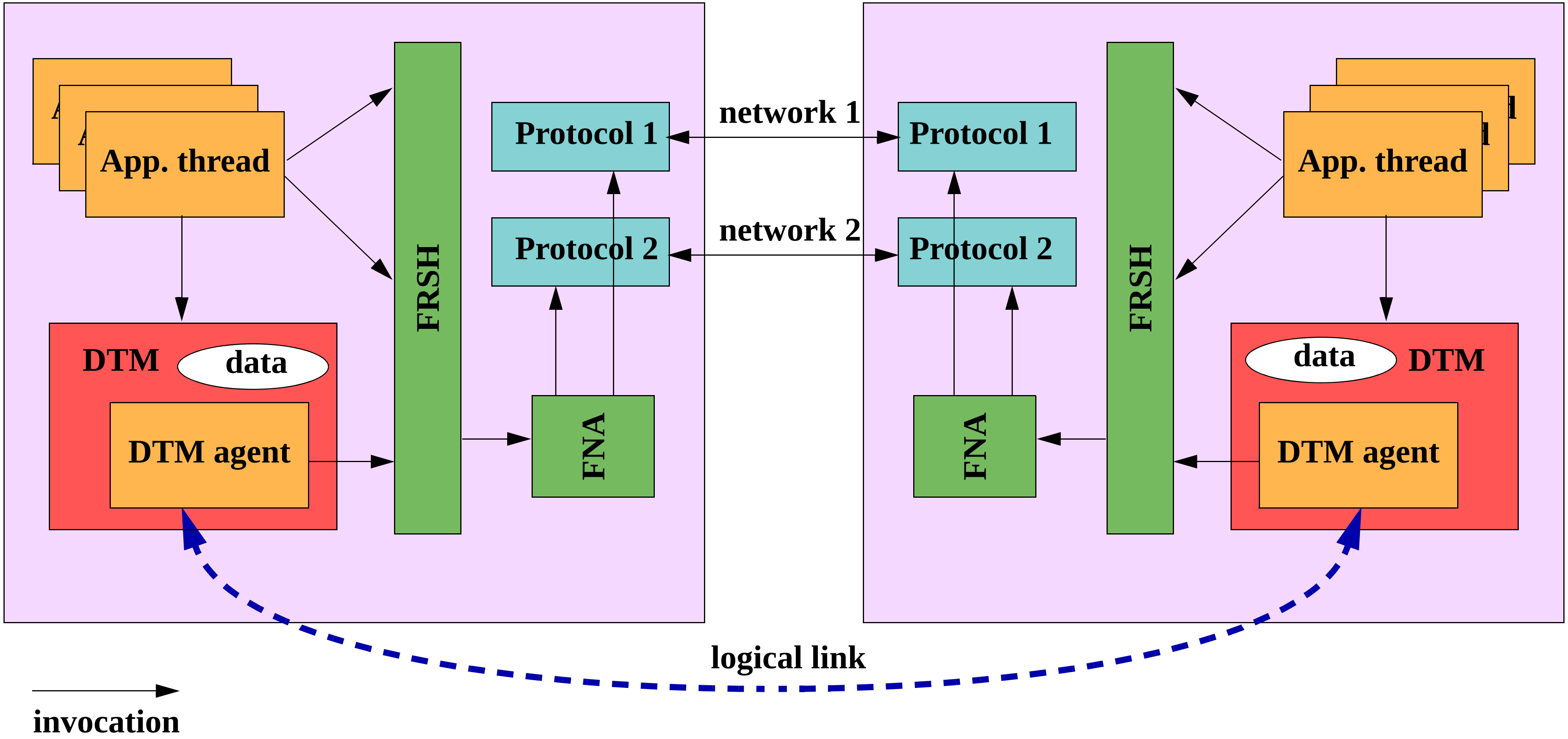
Sensor thread

```
create CPU contract
negotiate the CPU contract
create Network contract
negotiate the Network contract
create send_endpoint
bind server to send_endpoint

while (1) {
   read sensor
   send message
   frsh_timed_wait
}
```

Controller thread

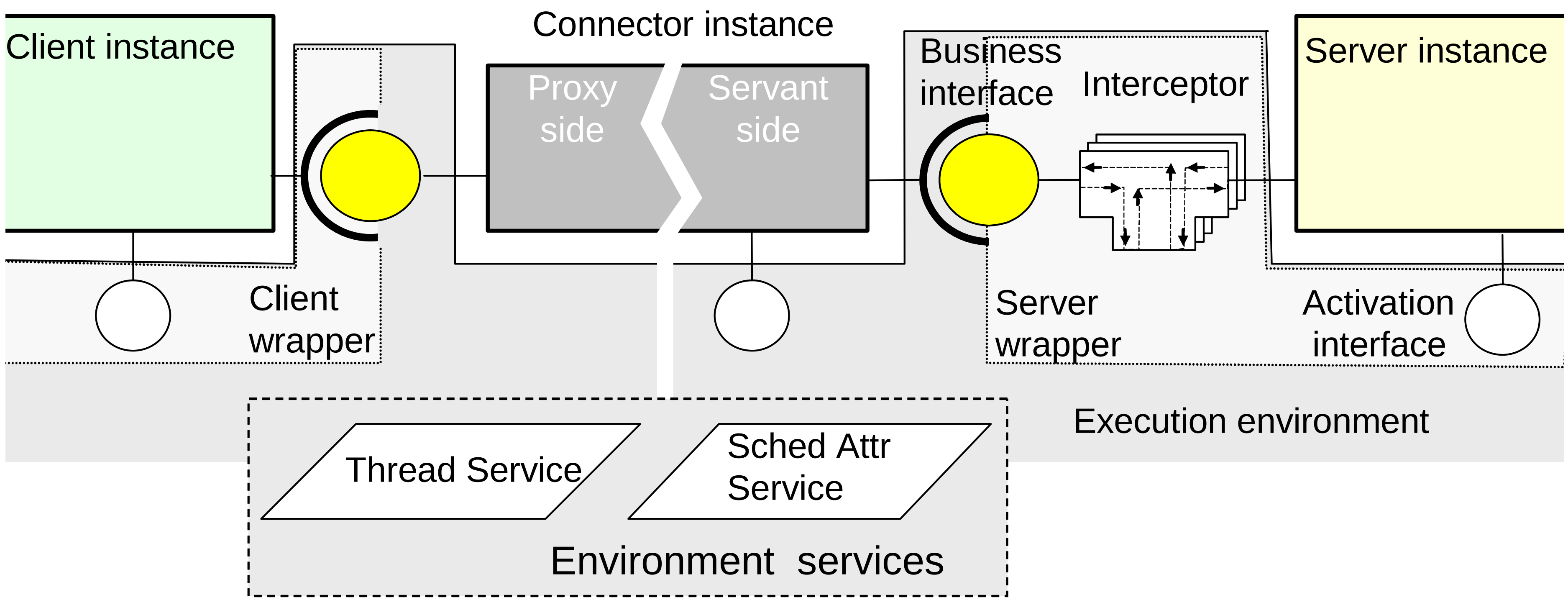
```
create contract
negotiate the contract
create receive endpoint

while (1) {
   read message
   process message
}
```

Distributed transaction manager



7. Container-Component framework (CCM)

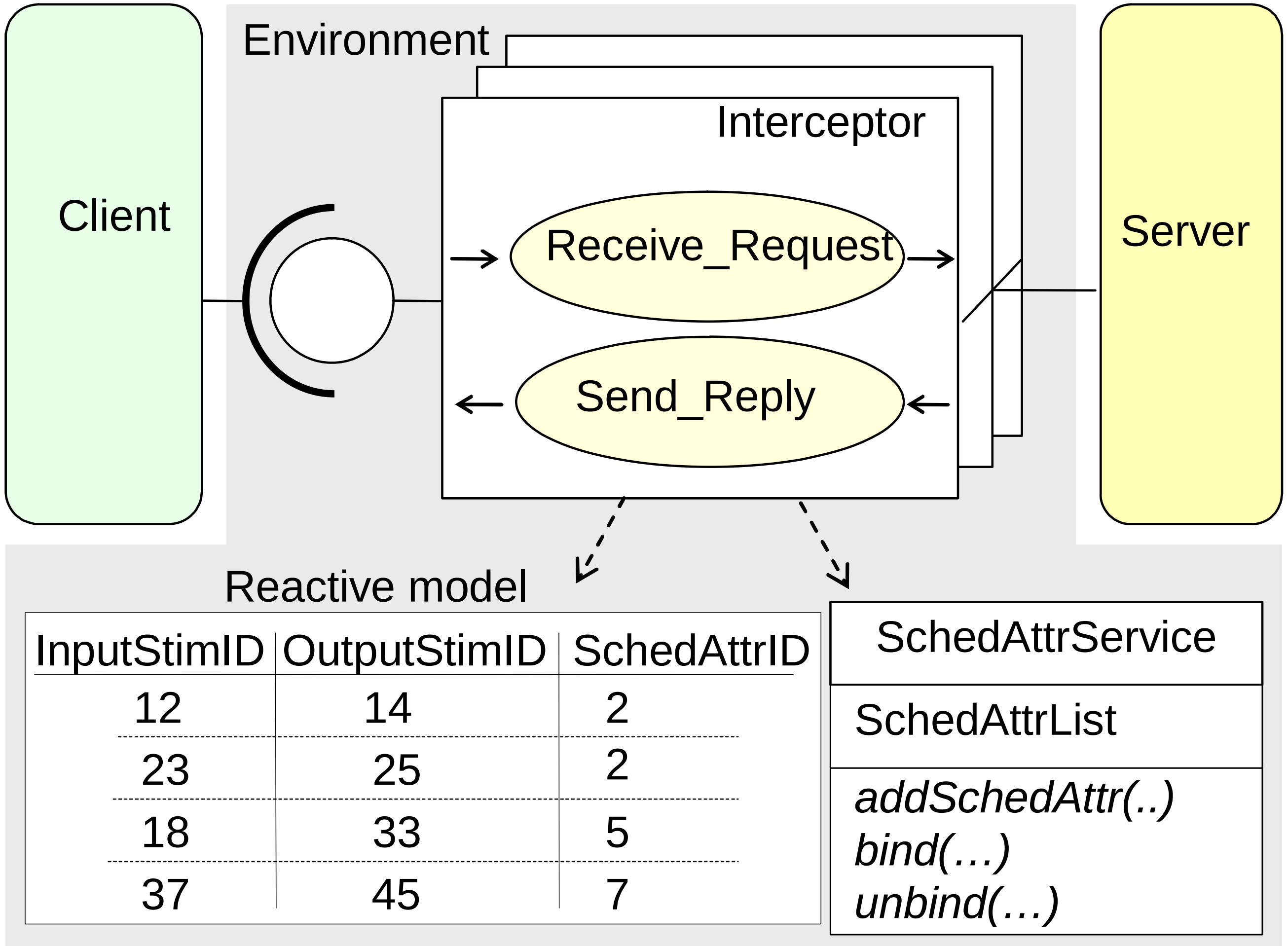
- Reusable components with passive operations
- Threads for executing the operations offered and managed by the container
- Connectors used for communication management
- FRESCOR management achieved by interception



Interceptors

Support the real-time model:

- Assign scheduling attributes to invoking threads
- Differentiates invocations based on global activities



InputStimID	OutputStimID	SchedAttrID
12	14	2
23	25	2
18	33	5
37	45	7

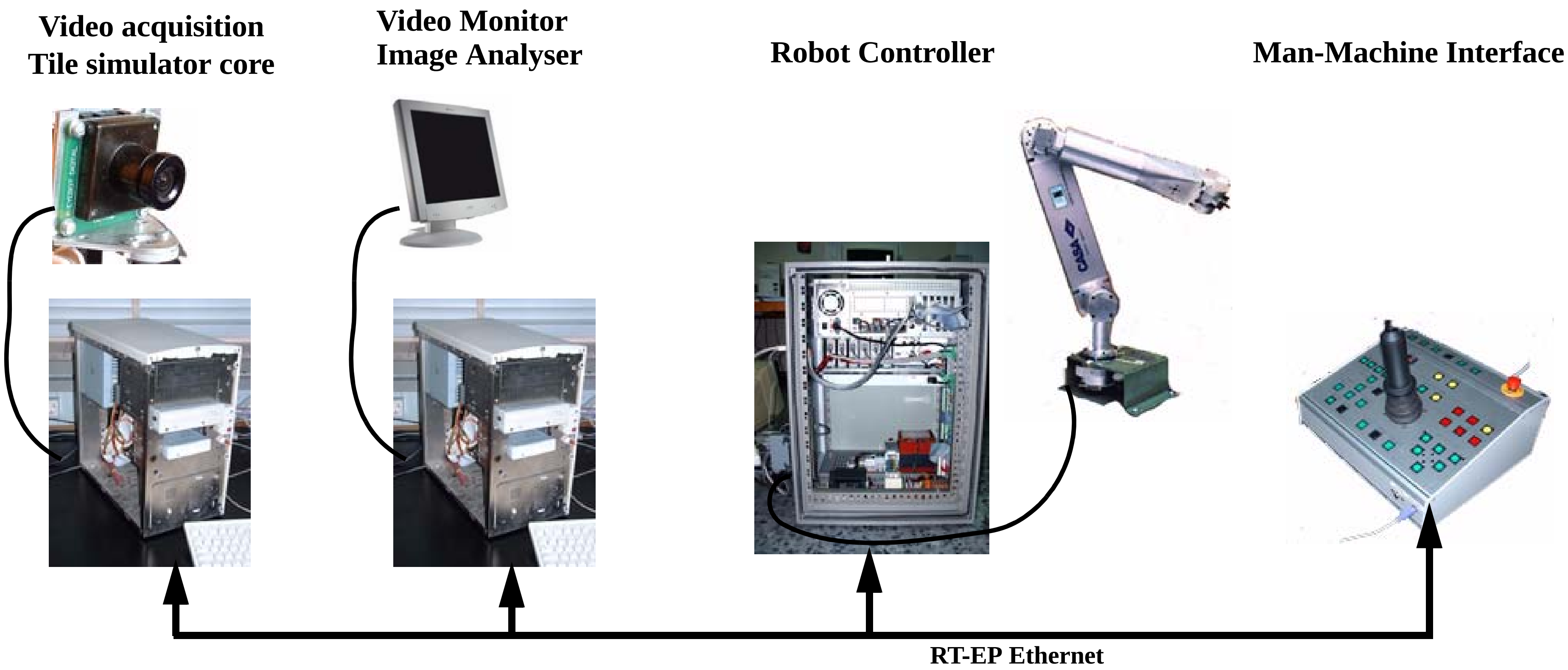
Automatically generated according to deployment data

8. Implementation
Implementation on RTOS

FRESCOR independent of underlying OS

- but underlying FRSH implementation is not
- preliminary implementations:
  - prototype based on fixed priorities + immediate priority ceilings (MaRTE OS)
  - prototype based on EDF + bandwidth inheritance (Shark)

Case study: distributed robot controller



Scheduling services required from OS

- RT scheduling (threads, mutexes, condition variables, ...)
- Notification mechanism (signals)
- execution time budgeting
- general purpose timers
- long jumps
  - asynchronous notification mechanism, to abort a sequence of statements
- user-defined scheduling
  - with hooks to operations when a thread gets blocked, and when a thread gets ready

Current FRSH implementations

Main implementation based on fixed priorities and immediate priority ceilings

Defined a POSIX-like OS adaptation layer: FOSA

- make the implementation independent of underlying fixed-priority OS
- requires OS adaptation (user-defined scheduling)

Current implementations of FOSA

- Partikle/ RT-Linux GPL
- OSE
- MaRTE OS native
- MaRTE OS as a user linux process
- RapiTime simulator

A second implementation exists on Linux/AQUOSA

Implementation on networks

RT-EP

- token passing fixed priority on standard ethernet
- uses fixed priorities & sporadic servers

CAN bus

- using fixed priorities & sporadic servers

WiFi

- using reduced set of priorities

Switched ethernet & TCP/IP

- using industrial switches that support priorities and traffic shaping

9. Conclusion

Contract based scheduling brings flexibility and resource reservations

The contracts provide independence among the different components of the application

They help the application developer by raising the level of abstraction of the real-time scheduling services.

Coexistence and cooperation of diverse real-time scheduling schemes

- hard real-time
- quality of service

Temporal encapsulation of subsystems

- support the composability of independently developed components
- reusability legacy subsystems

Learn more

http://frescor.org

Documents available

First public release expected within a year