# A02-4. *Execution View*

2014

Sungwon Kang

| Solution Design | Implementation Preparation | Implementation |
|---|---|---|
| Conceptual View | Repartitioning | |
| | Module View | |
| Execution View | | |
| | Code View | |

Components contain functionality,

Connectors contain control and interaction

Modules contain everything

# Module View First or Runtime View First ?

- Is the programming language fixed?
- Does the language support the module concept?
- Will your conceptual module be the same as the module of the language?

------------------------------ (conceptual module vs. language module)

- What are your conceptual runtime entities?
- What  are the runtime entities that your programming language supports ?

-------------------------------- (conceptual vs. language runtime entity)

☛ If mapping to runtime entities of the programming language can be postponed to Code View, then Module View can be designed first.

☛ However, if it cannot, then Runtime View must be designed first.

-------------------------------- (conceptual runtime entity should come first,
then conceptual or language module next)

# A. Purpose

- Describe the structure of a system in terms of its runtime platform elements (hardware platform and software platform; e.g. OS tasks, processes, threads, address spaces)
- Captures
  - How the system's functionality is assigned to platform elements
  - How the resulting runtime instances communicate
  - How physical resources are allocated to the runtime instances
  - The location, migration and replication of the runtime instances
- Addresses
  - Performance
  - Distribution

    E.g.) In the client-server architecture, in module view the client API is part of the server. But at runtime, it is part of the application process.
  - Replication

    E.g.) Multiple client APIs for multiple client applications are instantiated

# B. Context

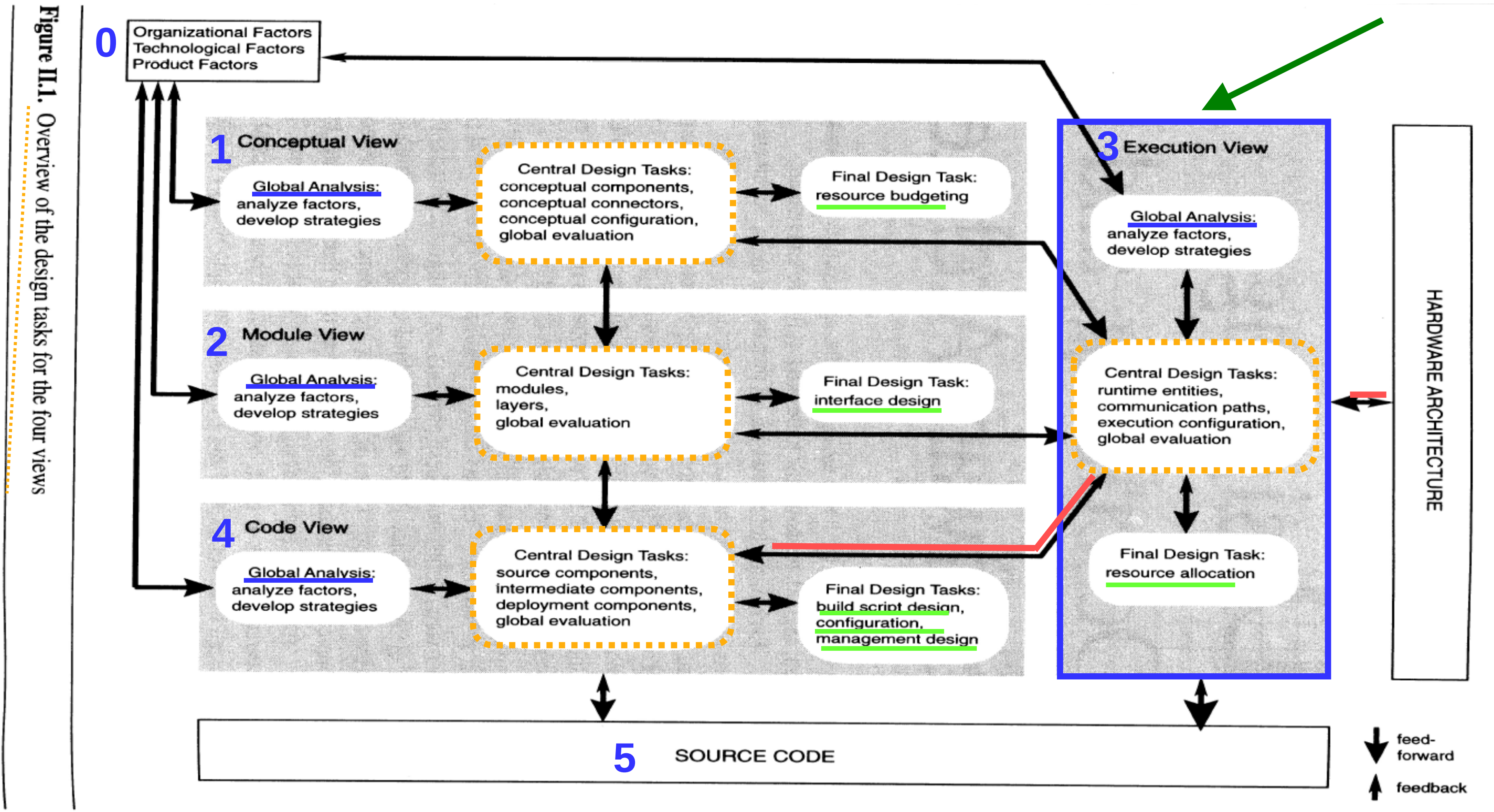


Figure II.1. Overview of the design tasks for the four views
0
Organizational Factors
Technological Factors
Product Factors
1 Conceptual View
Global Analysis:
analyze factors,
develop strategies
Central Design Tasks:
conceptual components,
conceptual connectors,
conceptual configuration,
global evaluation
Final Design Task:
resource budgeting
2 Module View
Global Analysis:
analyze factors,
develop strategies
Central Design Tasks:
modules,
layers,
global evaluation
Final Design Task:
interface design
3 Execution View
Global Analysis:
analyze factors,
develop strategies
Central Design Tasks:
runtime entities,
communication paths,
execution configuration,
global evaluation
Final Design Task:
resource allocation
4 Code View
Global Analysis:
analyze factors,
develop strategies
Central Design Tasks:
source components,
intermediate components,
deployment components,
global evaluation
Final Design Tasks:
build script design,
configuration,
management design
HARDWARE ARCHITECTURE
5 SOURCE CODE
feed-forward
feedback

# C. Global Analysis (1/2)

- Begin by reviewing the analysis for the conceptual and module views.
  - Identify the factors that affect execution view

    e.g. performance requirements, communication mechanisms

- Perform an analysis of the hardware platform and the software platform
  - Hardware platform
    - Know a list of components and their interconnection
    - Know parts to change, likelihood of change, when change will occur
  - Software platform
    - Identify infrastructure software

      E.g. OS, networking software, middleware, DBMS etc.
    - List platform elements to use

      E.g. UNIX process, NT thread, queue, file, etc.
    - Determine parts to change, likelihood of change, when change will occur

# C. Global Analysis (2/2)

- Notes
  - Software platform may have to change due to change in hardware platform
  - New factors may be obtained
  - Look for new issues related to performance and dependability
  - Develop strategies for the issues

    E.g. resource sharing, scheduling policies etc.

# D. Central Design Tasks (1/2)

1. Runtime Entities
   - Modules are assigned to runtime entities
   - Identify platform elements (OS tasks, processes, threads, address spaces)
   - Decide how to map components and modules to the platform elements

2. Communication paths
   - Identify the expected and/or allowable communication paths between them
     - Mechanisms: IPC, RPC, DCOM, etc.
       - May use platform elements: mailbox, queue, buffer, files
     - Resources

# D. Central Design Tasks (2/2)

3. Execution Configuration
   - Describe the system's runtime topology
   - Characterize the instances of the runtime entities and how they are interconnected
   - Determine each runtime instance and its attributes

     E.g. runtime entity, host name, info about resource allocation, info about its creation and termination
4. Global Evaluation (ksw: Can be done as Final Design.)
   - Balance guidelines and restrictions

     E.g. strategies for performance and dependability

     Conceptual view: concurrency

     => Execution view must support this
   - May perform performance experiment or simulation
   - Based on the results, adjust or refine the architecture

# E. Final Design Tasks

- Perform resource allocation
  - Allocate runtime instances and budgets to hardware devices
    - ✓ Assign values to the budgeted attributes (e.g. by setting process priorities)
    - ✓ Decision examples:
      - Processes are assigned 256K of shared memory
      - Rate monotonic scheduling is used to assign priorities

- If there are not enough resources, then revisit decisions made during central design tasks

# [1] Global Analysis

# New Issue: High Throughput

- Relevant strategies from the previous analysis
  - Issue 2: Skill Deficiencies
    - S2A: Avoid use of multiple threads
    - S2B: Encapsulate multiprocess support facilities

- New Issue: High Throughput

  <= Performance is important because of the very high data rate of the probe hardware
    - For low cost => use a single CPU, 64 MB
    - Realtime requirement => use high-end Pentium processor
      - If a single CPU is not enough,

        Use a new strategy S8B: *Use an additional CPU*
        - 1st CPU : realtime processing
        - 2nd CPU : UNIX
  - To achieve higher performance, need concurrency
    - New strategy S8A: *Map independent threads of control to processes*

# 8. High Throughput

The system has high-performance probe hardware with a very high data rate, higher than for previous products. The processing rate must keep up with the data rate from the probe hardware, at least up to the point at which data is recoverable. Common techniques to achieve higher performance include the use of multiple threads and multiple processes. However, the development team is deficient in the necessary skills.

**Influencing Factors**

O2.3: There is only one developer with expertise in multithreading.

O2.4 There are only two developers with expertise in using multiple processes.

P7.1: The budget for the product is limited and there is very little flexibility in changing it.

T1.2: We don't know whether one CPU will be sufficient to meet system performance needs when fully loaded. It is possible to enhance the system performance by adding a CPU. However, this may exceed the budget for the product.

T3.2: The cost of creating/destroying operating system processes is low.

**Solution**

We know from experience that to achieve adequate performance we must maximize the use of the processor by maximizing concurrency. We need an approach for achieving this, given the skill set of the development team. If one processor is not sufficient to handle peak system load, there are a couple of options. We could add another processor running the same real-time operating system or a general-purpose operating system like UNIX. If additional processing power is needed, we must then determine what is technically feasible, the impact it will have on the cost of the unit, and how it affects the design.

**Strategy:** ***Map independent threads of control to processes.***

To increase performance, take advantage of the low cost of process creation/destruction and map independent threads of control to processes. This strategy complements the strategy *Avoid use of multiple threads.*

**Strategy:** ***Use an additional CPU.***

Perform experiments to determine whether one CPU is sufficient. If the processor load is too high, use a standard real-time operating system and consider a dedicated "real-time CPU." This further isolates the real-time requirements and allows a more general processor with more flexibility for the nonreal-time portion.

**Related Strategies**

Related strategies are *Encapsulate multiprocess support facilities* and *Avoid use of multiple threads* (issue, Skills Deficiencies).

# Real-time Acquisition Performance

- IS2000 realtime performance requirements:
  - Maximum signal data rate for acquiring data
    - Rate at which the probe control can acquire data
  - Acquisition performance
    - Size and number of images
    - End-to-end acquisition response time
- Performance estimation
  - S9D: *Use Rate Monotonic Analysis (RMA) to predict performance*

☛ For execution view, we have

1) Processes are the basic unit for execution
2) May need a second CPU

- Still may have to adjust process boundaries during implementation

  => Strategies to reduce process adjustment cost:

  - S9C: *Use flexible allocation of modules to processes*
  - S9B: *Develop guidelines for module behavior*

**Revisit**

## 9. Real-time Acquisition Performance

Meeting real-time performance requirements is critical to the success of the product. There is no separate source code for meeting the real-time performance requirements directly. The source code that implements functional processing must also meet the performance constraints.

**Influencing Factors**

T1: General-purpose hardware

T3: Operating system, operating system processes, and database management system

P3.1: Maximum signal data rate

P3.2: Acquisition performance

**Solution**

Partition the system into separate components for algorithms, communication, and control to provide the flexibility to implement several different strategies. Use analysis techniques to predict performance to help in the early identification of performance bottlenecks.

**Strategy:** ***Separate time-critical components from nontime-critical components.***

To isolate the effects of change in the performance requirements, partition the system into components (and modules) that participate in time-critical processing and those that do not. This requires careful consideration at the interface between the real-time and nonreal-time sides of the system.

**Strategy:** ***Develop guidelines for module behavior.***

Impose a set of guidelines on module behavior to help eliminate performance bottlenecks and to support correct behavior. For example, ensure that modules have a single thread of execution, are reentrant, and are nonblocking.

**Strategy:** ***Use flexible allocation of modules to processes.***

Make it easy to change the module-to-process allocation so that the system can be tuned to achieve the required performance. This flexibility can also be used to group modules or threads with similar deadlines, periods, or frequencies, then assign the group to the same process to reduce scheduling and switching overhead.

**Strategy:** ***Use rate monotonic analysis (RMA) to predict performance.***

Use RMA to make sure the project is on track for fulfilling performance requirements.

---

**Related Strategies**

See also *Separate components and modules along dimensions of concern* (issue, Skills Deficiencies) and *Encapsulate multiprocess support facilities* (issue, Easy Addition and Removal of Features).

# Resource Limitations

- Driven by budget and technological factors

- To support realtime processing requirements, use QNX
  - Use only POSIX compliant features

    => The OS can be replaced with another POSIX compliant OS

  Question: what factors necessitate this?

- Budget limitation –> Memory limitation
  - S5A: *Limit the number of active processes*

# New Issue

## 5. Resource Limitations

To provide support for meeting the real-time processing requirements, a UNIX-like operating system that supports real-time processes is selected. The platform elements are processes, timers, shared memory buffers, and queues. It is relatively inexpensive to create and to destroy processes. Also, there are a fixed number of resources, such as sockets and timers.

The architecture design must cope with the limitations of these hardware and software resources. The strategies should provide guidance for making design choices that cope with resource limitations and make it easy to adapt the system when these limitations change.

**Influencing Factors**

T1.3: The size of the memory is limited. It is not likely to change drastically due to budget limitations.

T3.2: Operating system processes also consume software resources such as memory. Too many active processes may degrade system performance. However, it is relatively inexpensive to create and to destroy processes on the selected operating system.

**Solution**

Use a flexible approach for the usage of limited resources.

**Strategy:** ***Limit the number of active processes.***

If memory requirements of active processes cause performance degradation, consider limiting the number of active processes that can run at the same time. We need to terminate and restart processes in this case. This is acceptable due to the low cost of process creation and destruction.

# [2] Central Design Tasks:

## Runtime Entities, Comm Paths and Configuration

# A. Begin Defining Runtime Entities

[D31] Begin by associating each high-level conceptual component with a set of execution elements

- We avoid multi-thread processes and instead put each thread in its own process

  => Figure 6.5

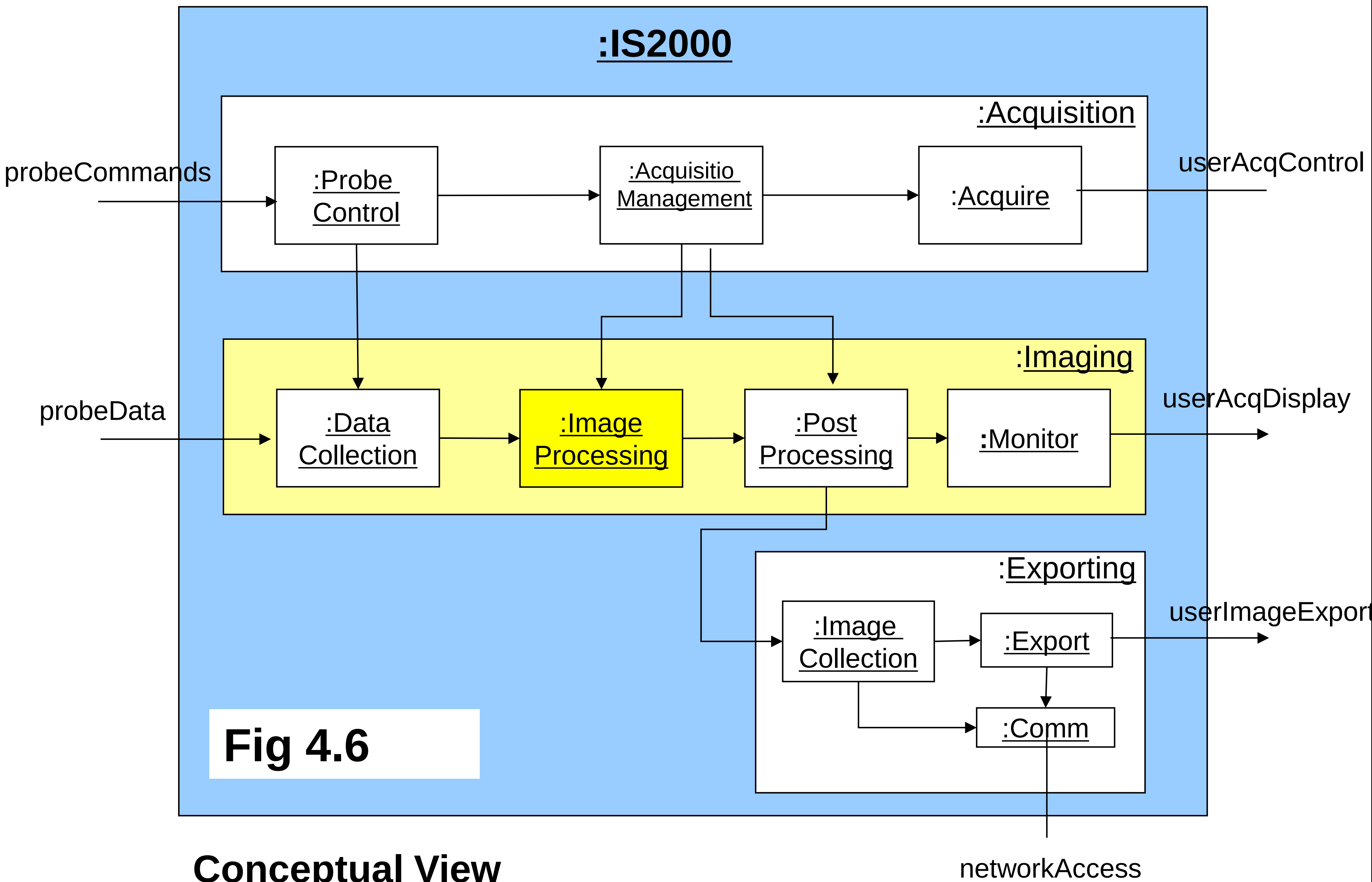
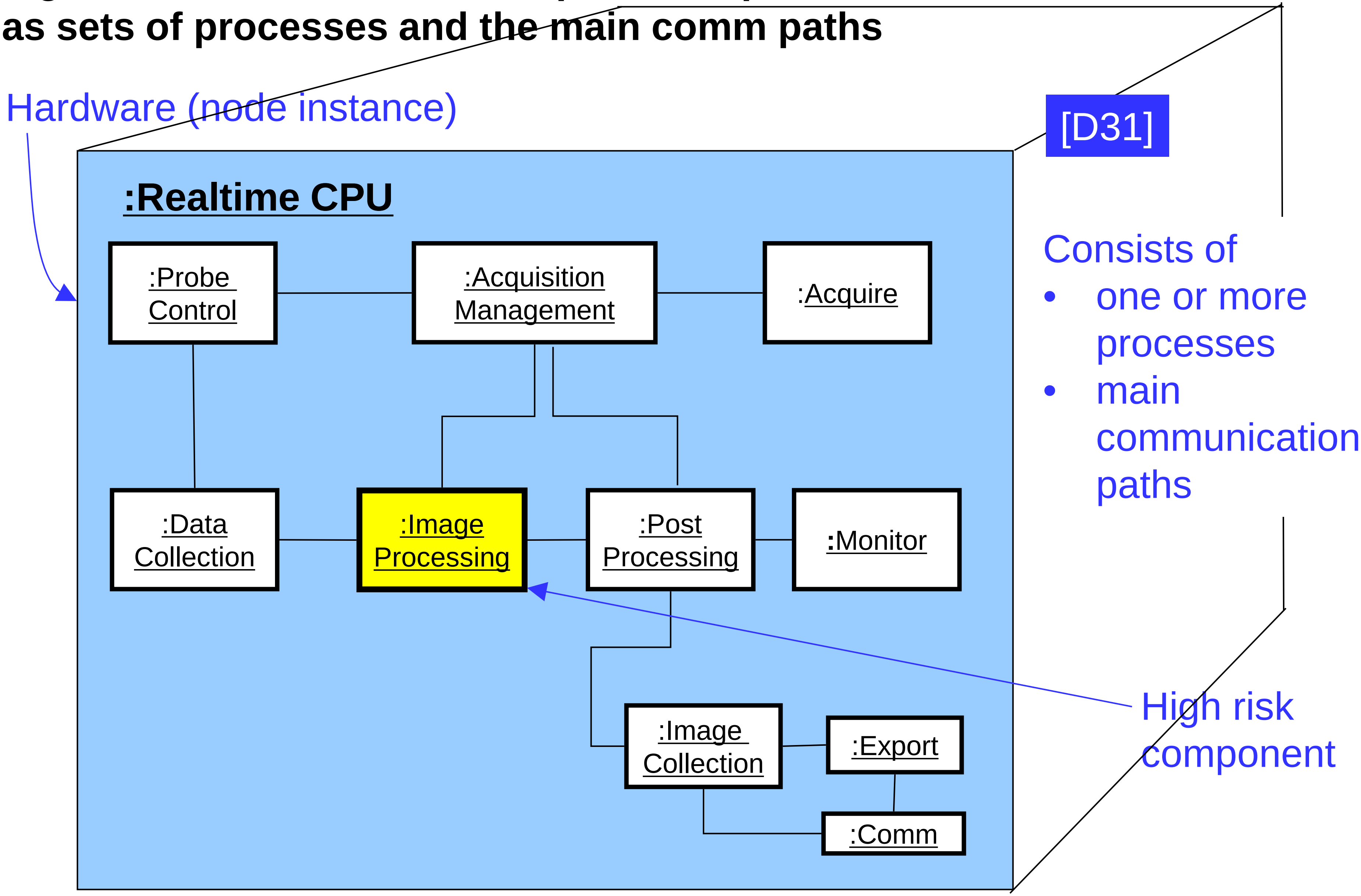


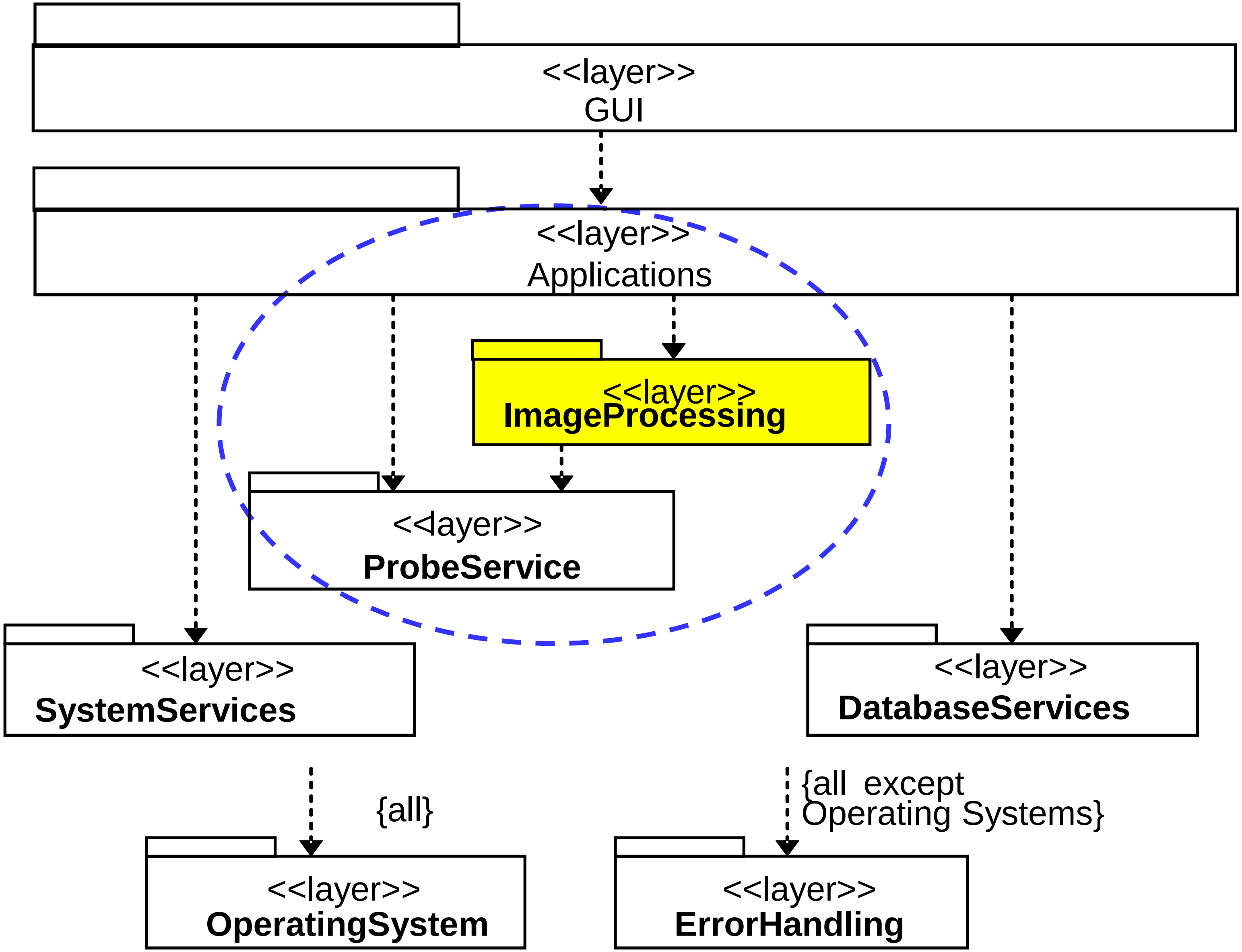

Fig 4.6

**Conceptual View**

# Fig 6.5 shows the main conceptual components as sets of processes and the main comm paths



# Recall the module view



# A. Begin Defining Runtime Entities

[D32] Look each process and perform central design tasks for each.

- Start with the highest risk parts first
  - ImageProcessing component is the computationally expensive part of the realtime part of the system

# Step A1.1)

- When there is a simple 1-1 correspondence between conceptual components and modules in Tables 5.1 and 5.2, assign a module to a process or a thread.

  (Straightforward implementation of the concurrency expressed by the conceptual view.)

# 1-1 Correspondence between Conceptual Components and Modules

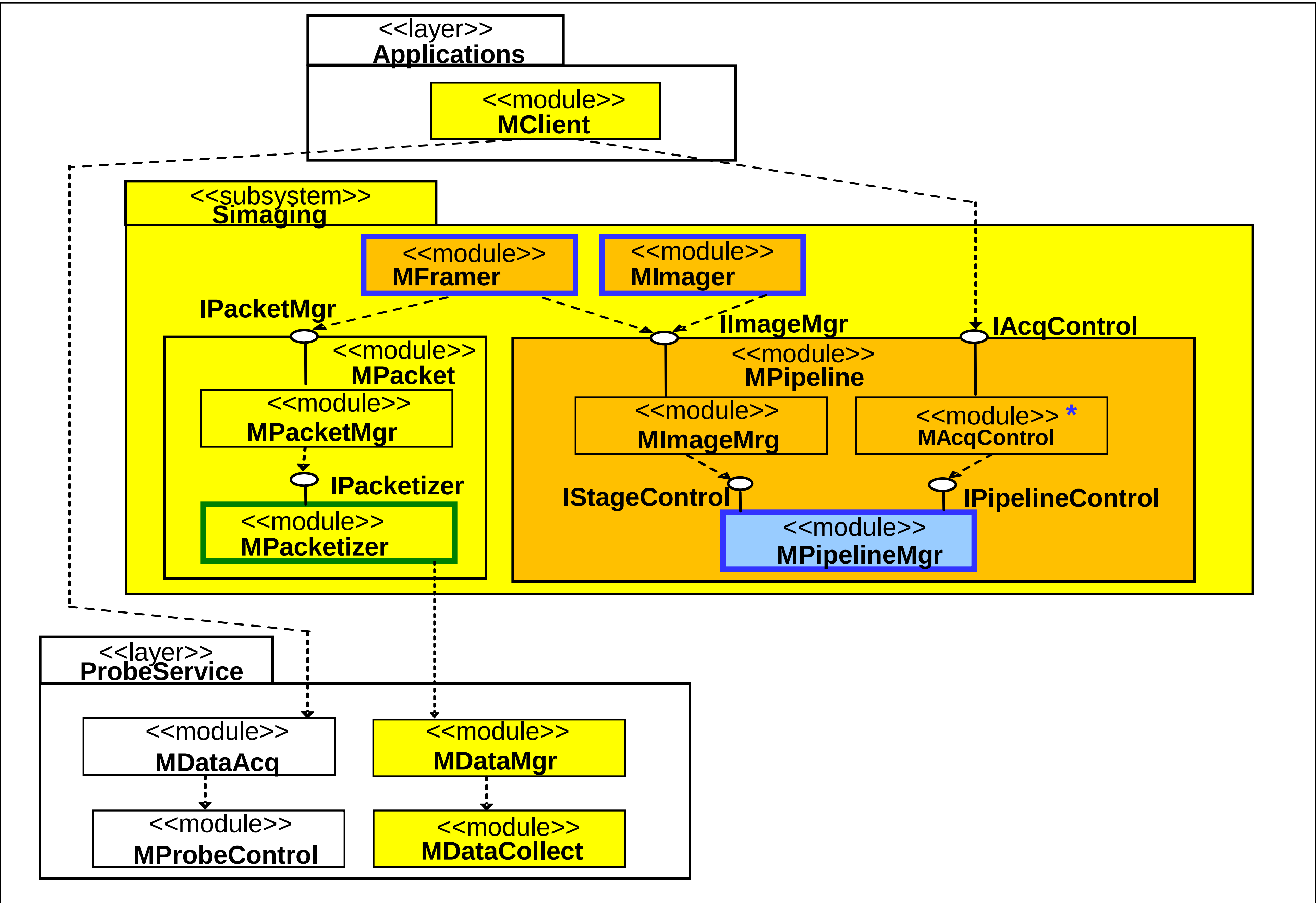
| Conceptual Element | | Module Element | |
|---|---|---|---|
| Name | Kind | Name | Kind |
| **ImageProcessing** | **Component** | **SImaging** | **Subsystem** |
| **ImagePipeline** | **Component** | **SPipeline** | **Subsystem** |
| **Packetizer**<br>packetOut<br>**PacketPipe**<br>source, dest | **Component**<br>Port<br>**Connector**<br>roles | **MPacketizer**✓ | **Module** |
| packetIn | Port | MPacketMgr | Module |
| acqControl | Port | MAcqControl | Module |

**Table 5.1 Mapping Conceptual Elements to Module Elements: ImageProcessing**

| Conceptual Element | | Module Element | |
|---|---|---|---|
| Name | Kind | Name | Kind |
| **PipelineMgr**<br>pipeline, stages<br>**ImagePipe**<br>source, dest<br>**Client/Server**<br>sender, receiver<br>**Event**<br>sender, receiver | **Component**<br>Ports<br>**Connector**<br>roles<br>**Connector**<br>roles<br>**Connector**<br>roles | **MPipelineMgr**✓ | **Module** |
| pipelineControl, stageControl, imageIn, imageOut | Ports | MImageMgr | Module |
| **Framer** | **Component** | **MFramer**✓ | **Module** |
| **Imager** | **Component** | **MImager**✓ | **Module** |

**Table 5.2 Mapping Conceptual Elements to Module Elements: ImagePipeline**

# Step A1) “One module to One process”



# Step A1.2)

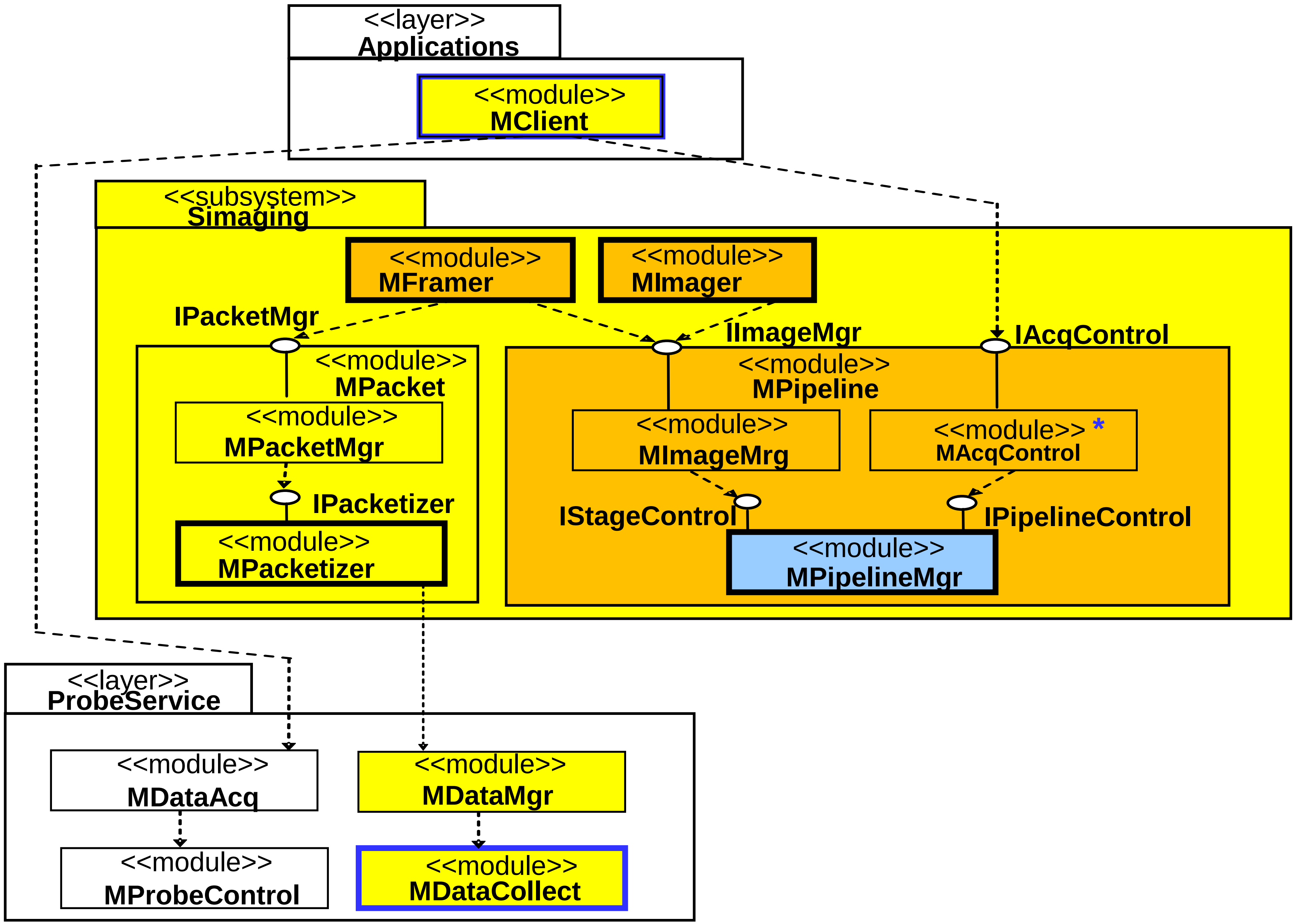
- **Strategy**: *Map independent threads of control to processes*

  => Create separate processes for each of the pipeline stages,

  for the image pipeline client (MClient) and

  the data collector (MDataCollect)

# Step A1.2)

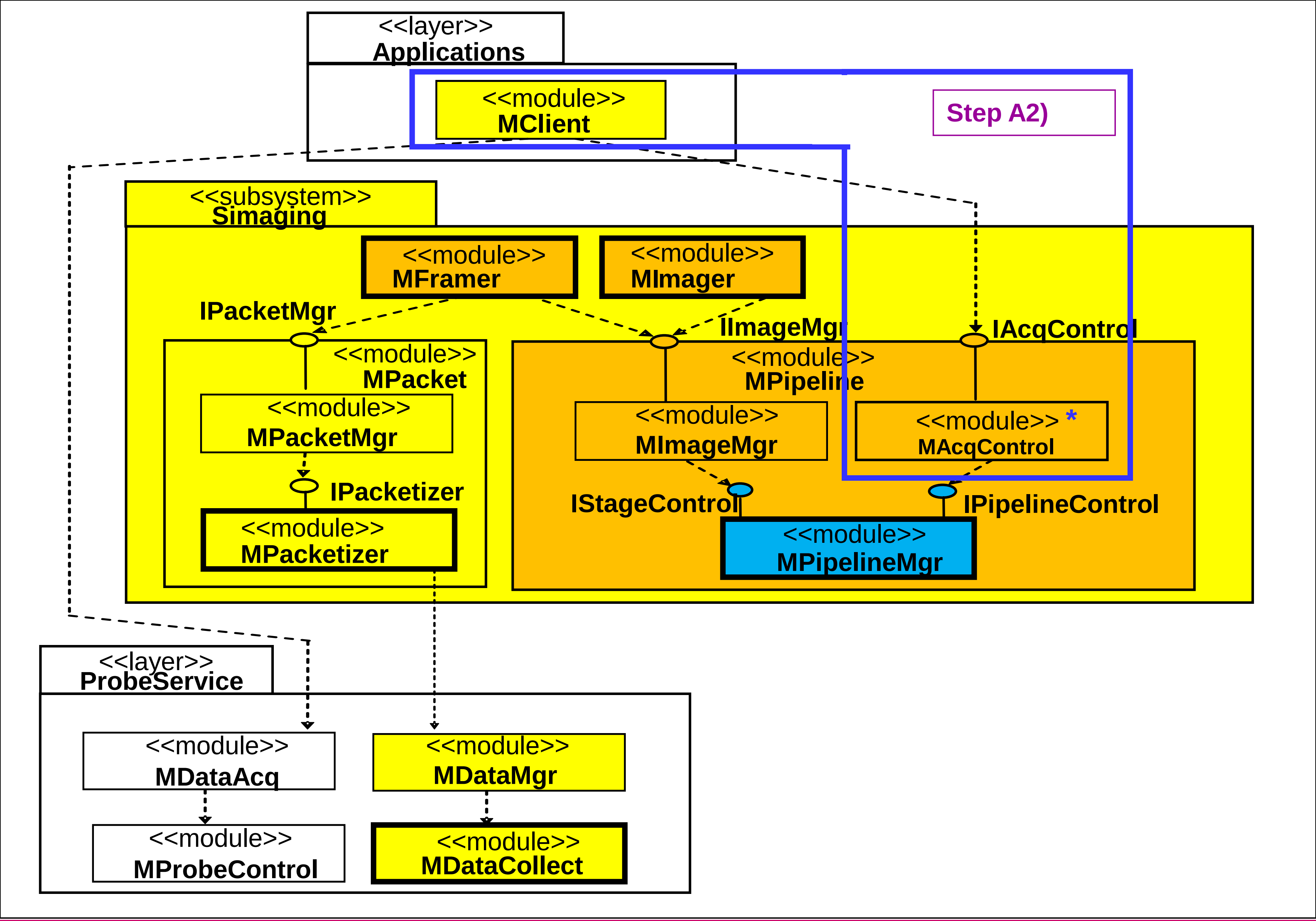


<<layer>>
Applications
<<module>>
MClient
<<subsystem>>
Simaging
<<module>>
MFramer
<<module>>
MImager
IPacketMgr
IImageMgr
IAcqControl
<<module>>
MPacket
<<module>>
MPacketMgr
IPacketizer
<<module>>
MPacketizer
<<module>>
MPipeline
<<module>>
MImageMrg
<<module>> *
MAcqControl
IStageControl
IPipelineControl
<<module>>
MPipelineMgr
<<layer>>
ProbeService
<<module>>
MDataAcq
<<module>>
MDataMgr
<<module>>
MProbeControl
<<module>>
MDataCollect

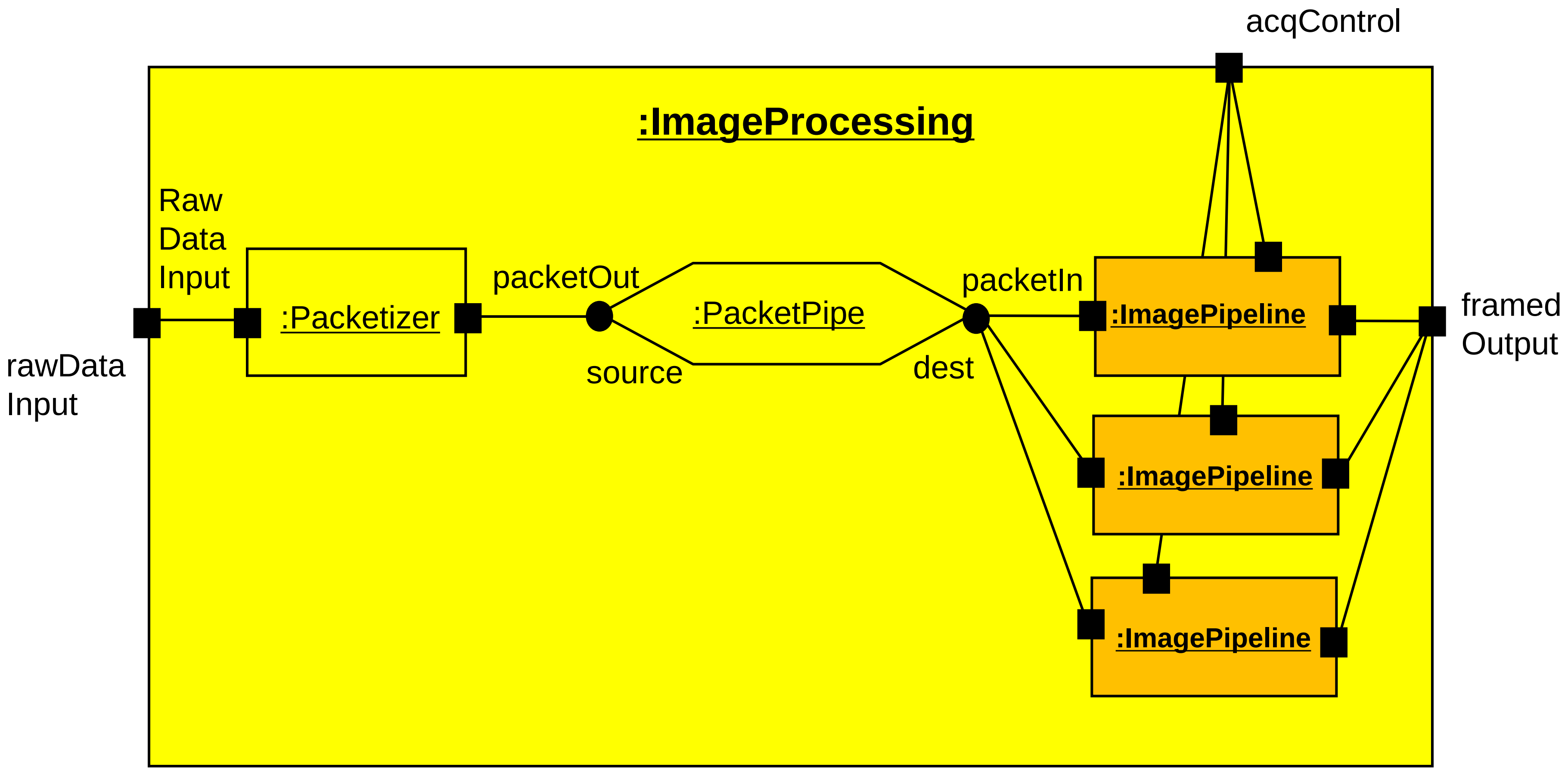
# Step A2)

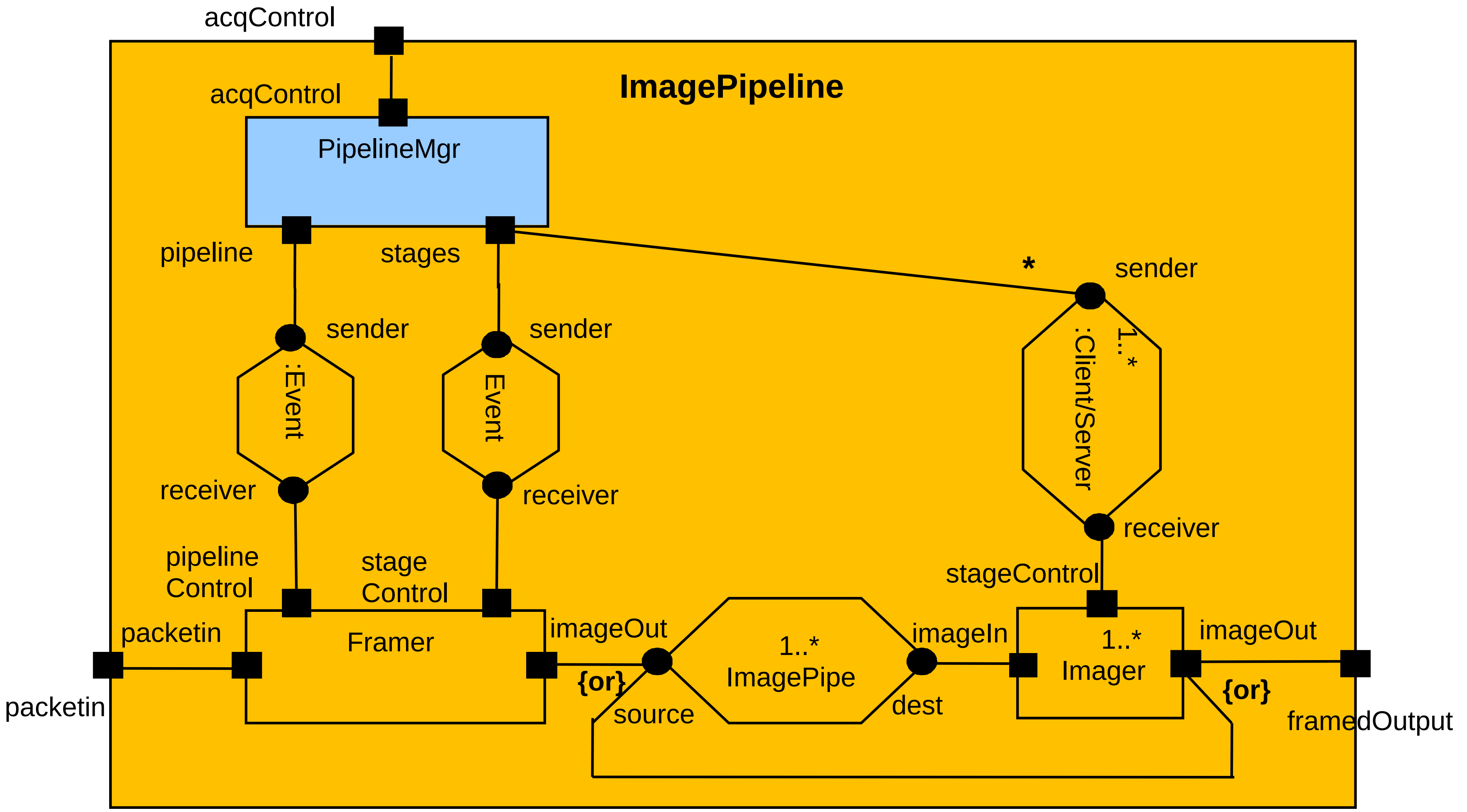
- Examine the dependencies to determine the resulting communication paths and mechanisms between the processes (See next slide)
  - If MClient and MAcqControl were in different processes, they have to communicate across process boundaries.

  => Putting them in one process lets them use a local procedure call.


<<layer>>
Applications
<<module>>
MClient
Step A2)
<<subsystem>>
Simaging
<<module>>
MFramer
<<module>>
MImager
IPacketMgr
IImageMgr
IAcqControl
<<module>>
MPacket
<<module>>
MPipeline
<<module>>
MPacketMgr
<<module>>
MImageMgr
<<module>> *
MAcqControl
IPacketizer
IStageControl
IPipelineControl
<<module>>
MPacketizer
<<module>>
MPipelineMgr
<<layer>>
ProbeService
<<module>>
MDataAcq
<<module>>
MDataMgr
<<module>>
MProbeControl
<<module>>
MDataCollect

# (Recall the Conceptual View)




acqControl
ImagePipeline
acqControl
PipelineMgr
pipeline
stages
*
sender
sender
sender
:Event
Event
:Client/Server
1..*
receiver
receiver
receiver
pipeline
Control
stage
Control
stageControl
packetin
Framer
imageOut
1..*
ImagePipe
imageIn
1..*
Imager
imageOut
packetin
{or}
source
dest
{or}
framedOutput

# B. Pipeline Manager

**Step B1)** Communication between pipeline stages

- Performance is important
- Not addressed yet by a strategy for **Issue 9**
- Shared memory is available in the selected OS.

## 9. Real-time Acquisition Performance

Meeting real-time performance requirements is critical to the success of the product. There is no separate source code for meeting the real-time performance requirements directly. The source code that implements functional processing must also meet the performance constraints.

**Influencing Factors**

T1: General-purpose hardware

T3: Operating system, operating system processes, and database management system

P3.1: Maximum signal data rate

P3.2: Acquisition performance

**Solution**

Partition the system into separate components for algorithms, communication, and control to provide the flexibility to implement several different strategies. Use analysis techniques to predict performance to help in the early identification of performance bottlenecks.

**Strategy:** ***Separate time-critical components from nontime-critical components.***

To isolate the effects of change in the performance requirements, partition the system into components (and modules) that participate in time-critical processing and those that do not. This requires careful consideration at the interface between the real-time and nonreal-time sides of the system.

**Strategy:** ***Develop guidelines for module behavior.***

Impose a set of guidelines on module behavior to help eliminate performance bottlenecks and to support correct behavior. For example, ensure that modules have a single thread of execution, are reentrant, and are nonblocking.

**Strategy:** ***Use flexible allocation of modules to processes.***

Make it easy to change the module-to-process allocation so that the system can be tuned to achieve the required performance. This flexibility can also be used to group modules or threads with similar deadlines, periods, or frequencies, then assign the group to the same process to reduce scheduling and switching overhead.

**Strategy:** ***Use rate monotonic analysis (RMA) to predict performance.***

Use RMA to make sure the project is on track for fulfilling performance requirements.

---

**Related Strategies**

See also *Separate components and modules along dimensions of concern* (issue, Skills Deficiencies) and *Encapsulate multiprocess support facilities* (issue, Easy Addition and Removal of Features).

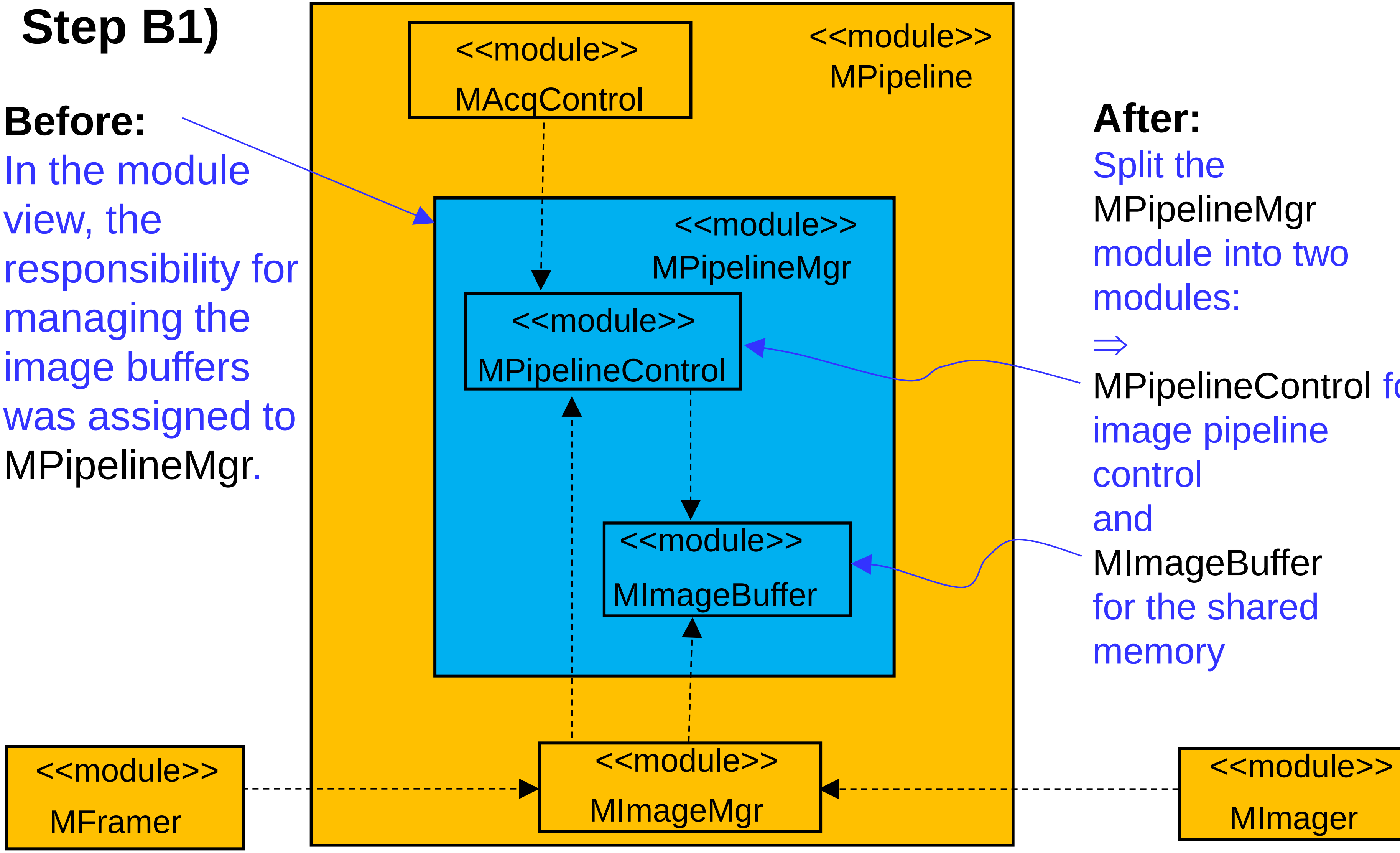
**Strategy:** ***Use shared memory to communicate between pipeline stages.*** New

Use shared memory between pipeline stages to eliminate any unnecessary data copying in the acquisition and processing pipelines.

**Figure 6.8** Revisions to the module view - MPipelineMgr

## Step B1)

**Before:**
In the module view, the responsibility for managing the image buffers was assigned to MPipelineMgr.




**After:**
Split the MPipelineMgr module into two modules:
⇒ MPipelineControl for image pipeline control
and MImageBuffer for the shared memory

# Step B2) Mapping Modules to Runtime Entities

- One possibility
  - *Replicate the control module* and link it to each process containing a pipeline stage (=> 1 control process per 1 pipeline stage)
    - Since MPipelineControl coordinates the pipeline stages, we should use a distributed control algorithms between EPipelineControl processes, which requires a complicated handshaking protocol than centralized control

      => **Distributed control:** Costly

☛ Simpler solution **- Centralized control**

- *Centralize pipeline control in a single process*, separate from the pipeline stage processes (=> 1 control process per 1 pipeline)

  ⇒ Since each image pipeline has its own manager, and active image pipelines run concurrently, each MPipelineMgr must be in a separate process

Create a new process called EPipelineMgr that contains MPipelineControl, which controls access but does not itself read or write to MImageBuffer

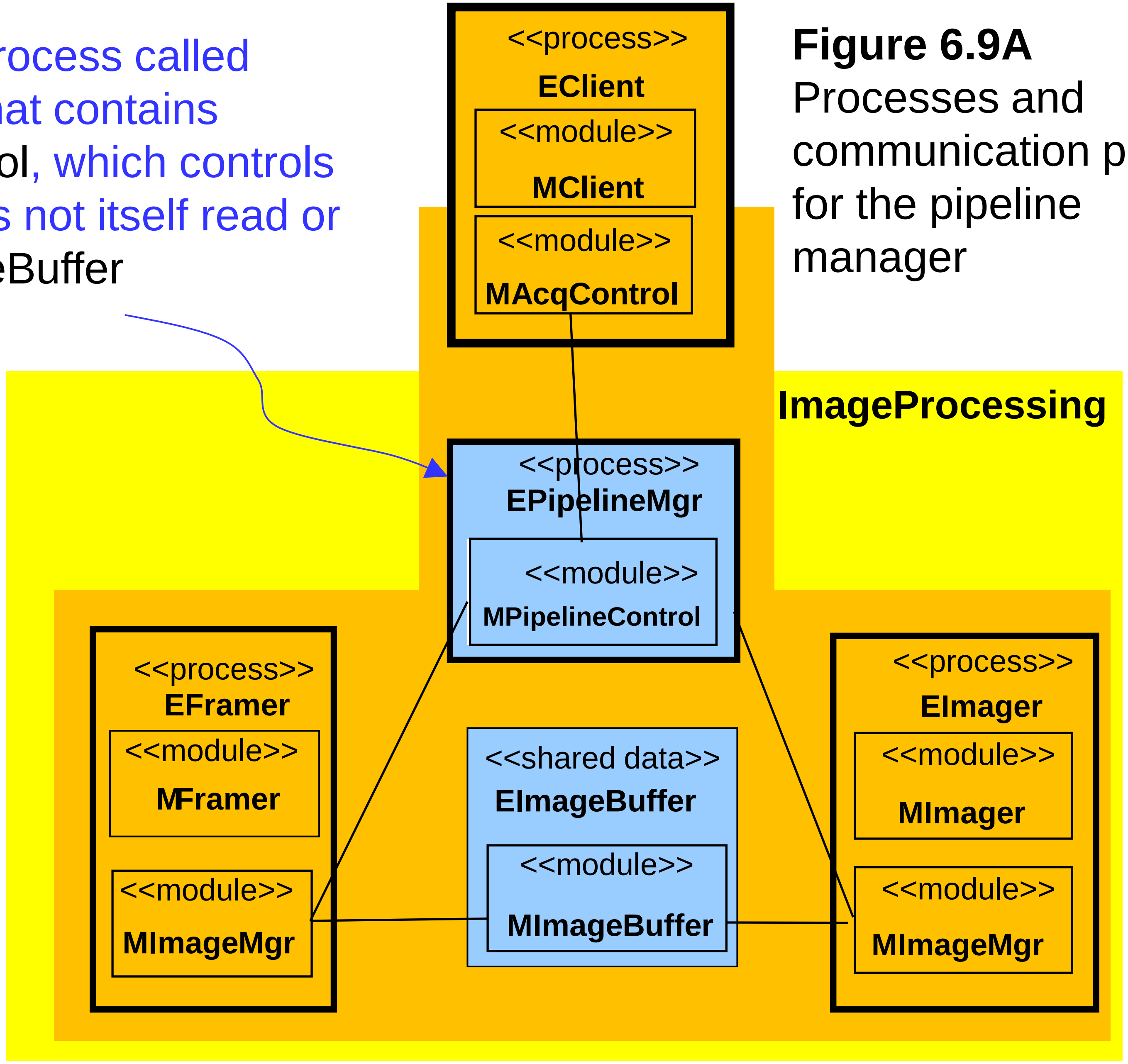



**Figure 6.9A** Processes and communication path for the pipeline manager

# C. Communication Paths (1/2)

ksw: Should we view this as a strategy or a requirement ?

- For communication between processes, use IPC.
- This is a new technology factor

  => Return to Global Analysis

| Technological Factor | Flexibility and Changeability | Impact |
|---|---|---|
| **T3: Software technology** | | |
| T3.4: Interprocess communication (IPC) mechanism | | |
| Use of IPC mechanisms requires resources such as sockets or mail-boxes. Such resources may be limited on a real-time operating system. | These resource limitations are often based on memory size. Because memory size is not expected to change during development, the limitation is not likely to change. The IPC mechanism is likely to change every five years. | The impact on components is moderate at the process boundary. We may need to develop an approach to deal with the limitation. A change in IPC mechanism can have a large impact on design. |

**Table 6.1.** Factor Added During Execution View Design

# C. Communication Paths (2/2)

- IPC requires limited resources such as sockets or mailboxes

  ⇒ Limited in RTOS

  - Strategy: *S5B: Use dynamic interprocess communication (IPC) connections*

    Warning: This strategy may degrade performance if the cost of creating and destroying IPC connections is too high

  - Use the strategy *Encapsulate multipocess support facilities* to reduce the burden of using IPC.

    (See **Figure 6.9B** )

# 5. Resource Limitations

To provide support for meeting the real-time processing requirements, a UNIX-like operating system that supports real-time processes is selected. The platform elements are processes, timers, shared memory buffers, and queues. It is relatively inexpensive to create and to destroy processes. Also, there are a fixed number of resources, such as sockets and timers.

The architecture design must cope with the limitations of these hardware and software resources. The strategies should provide guidance for making design choices that cope with resource limitations and make it easy to adapt the system when these limitations change.

**Influencing Factors**

T1.3: The size of the memory is limited. It is not likely to change drastically due to budget limitations.

T3.2: Operating system processes also consume software resources such as memory. Too many active processes may degrade system performance. However, it is relatively inexpensive to create and to destroy processes on the selected operating system.

**Solution**

Use a flexible approach for the usage of limited resources.

**Strategy:** ***Limit the number of active processes.***

If memory requirements of active processes cause performance degradation, consider limiting the number of active processes that can run at the same time. We need to terminate and restart processes in this case. This is acceptable due to the low cost of process creation and destruction.

**Strategy:** ***Use dynamic interprocess communication (IPC) connections.***

Make use of dynamic IPC connections between processes when possible. In this way, limited IPC resources such as sockets are used only when the processes are communicating. This strategy may degrade overall performance if the cost of creating and destroying IPC connections is too high.

- Create a new module MCommLib (communication library) to handle details of the IPC protocol
- Link it to the processes in green

UML association for communication path

ksw: With UML 2.0, may be view as a “UML 2.0 connector”

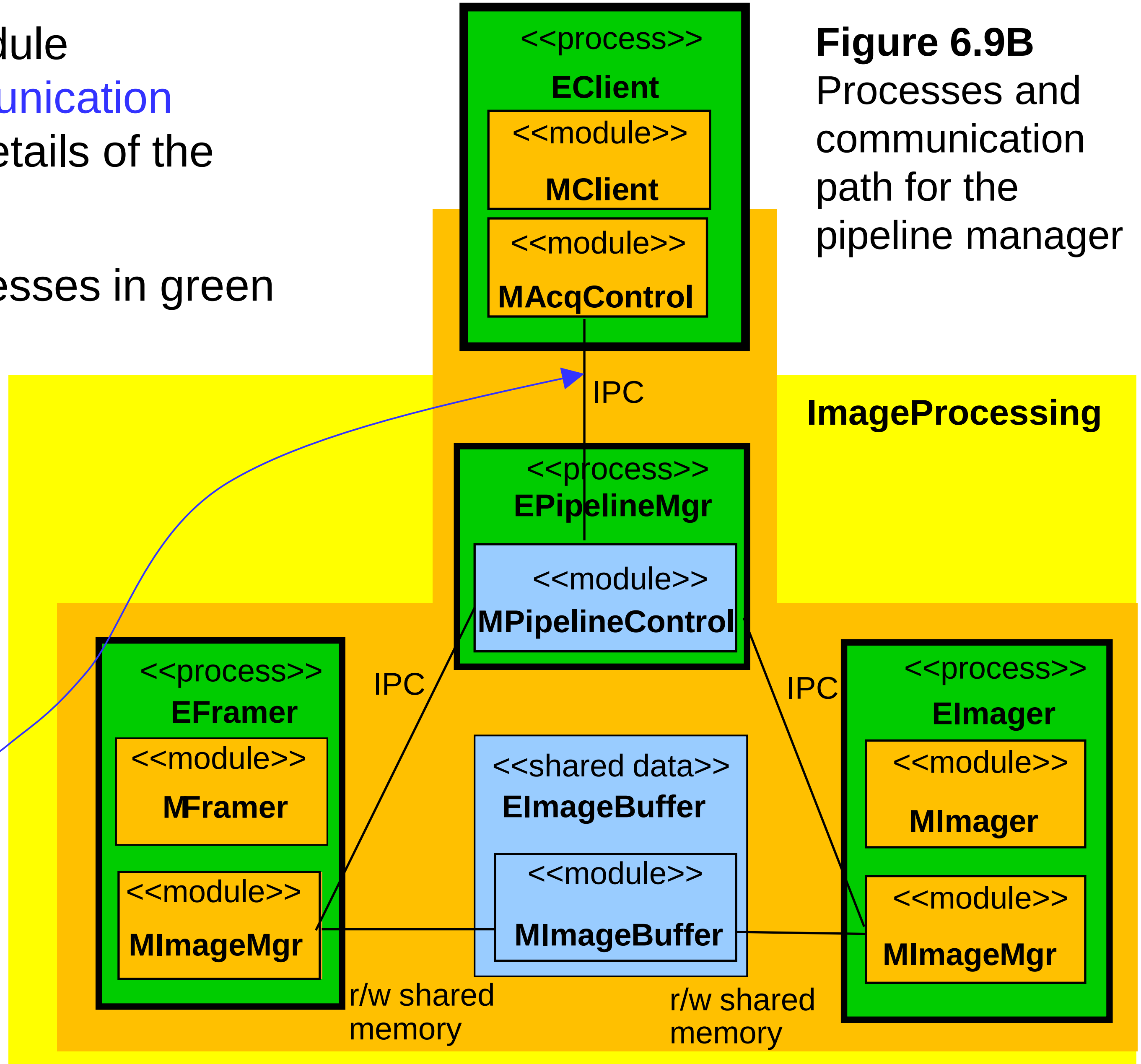
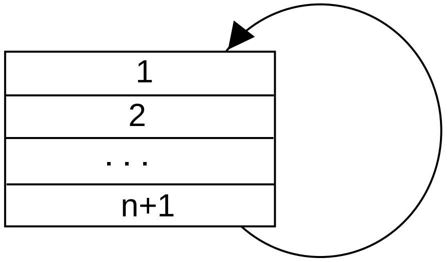



**Figure 6.9B** Processes and communication path for the pipeline manager

# Resource sharing policies and protocols

- In Figure 6.9B, MImageBuffer module is the data shared among the stages for that pipeline.

- Let's split it into multiple logical buffers, one for each stage
  - Each stage (Framer or Imagers) process has exclusive access (read/write) to one of these logical buffers
  - Each stage requests a buffer and releases it.
  - MPipelineControl accepts requests FCFS and determines which buffer is to be allocated



# D. Assigning Multiplicities

- Exactly how many processes will be running?
  – Part of *configuration design*

- Each image pipeline contain about 4 stages (Framer or Imagers) and there are at least 10 image pipelines, the pipelines could use more than 4 x10 processes.

  <= Memory limitation and performance requirements

  ⇒ *S5A: Limit the number of active processes*

  ⇒ The processes for an image pipeline must be created dynamically when the acquisition procedure is requested

- When configuration is not fixed as here, how can we describe dynamically changing configuration in a single diagram?

One for each image pipeline

Each MPipelineControl communicates with exactly one MAcqControl and each MAcqControl communicates with zero or one MPipelineControl.

**How many EPipelineMgr processes exist for each MAcqControl module of EClient process?**

**A Problem: IPC is a runtime entity but is connecting modules.**

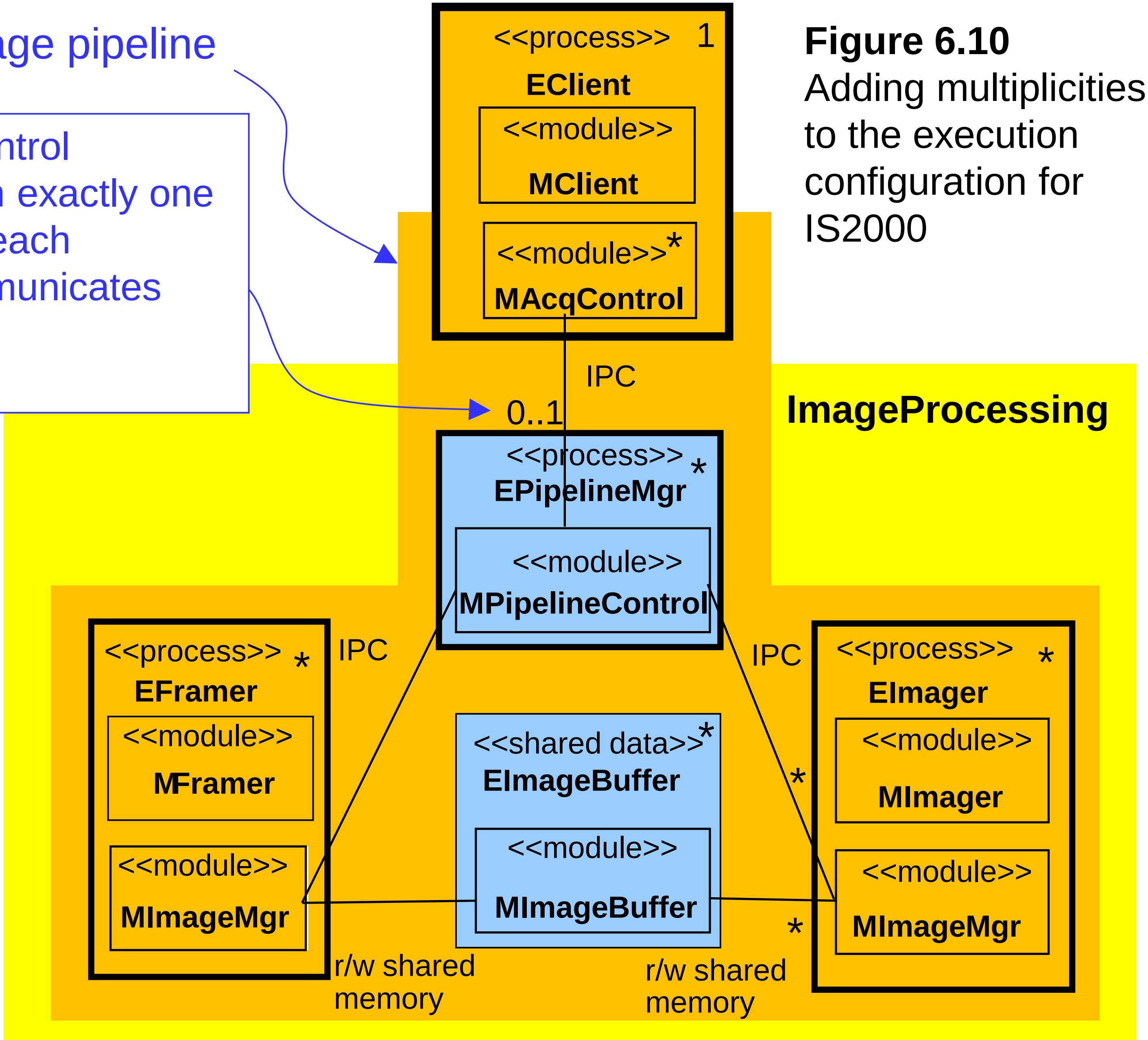



**Figure 6.10**
Adding multiplicities to the execution configuration for IS2000

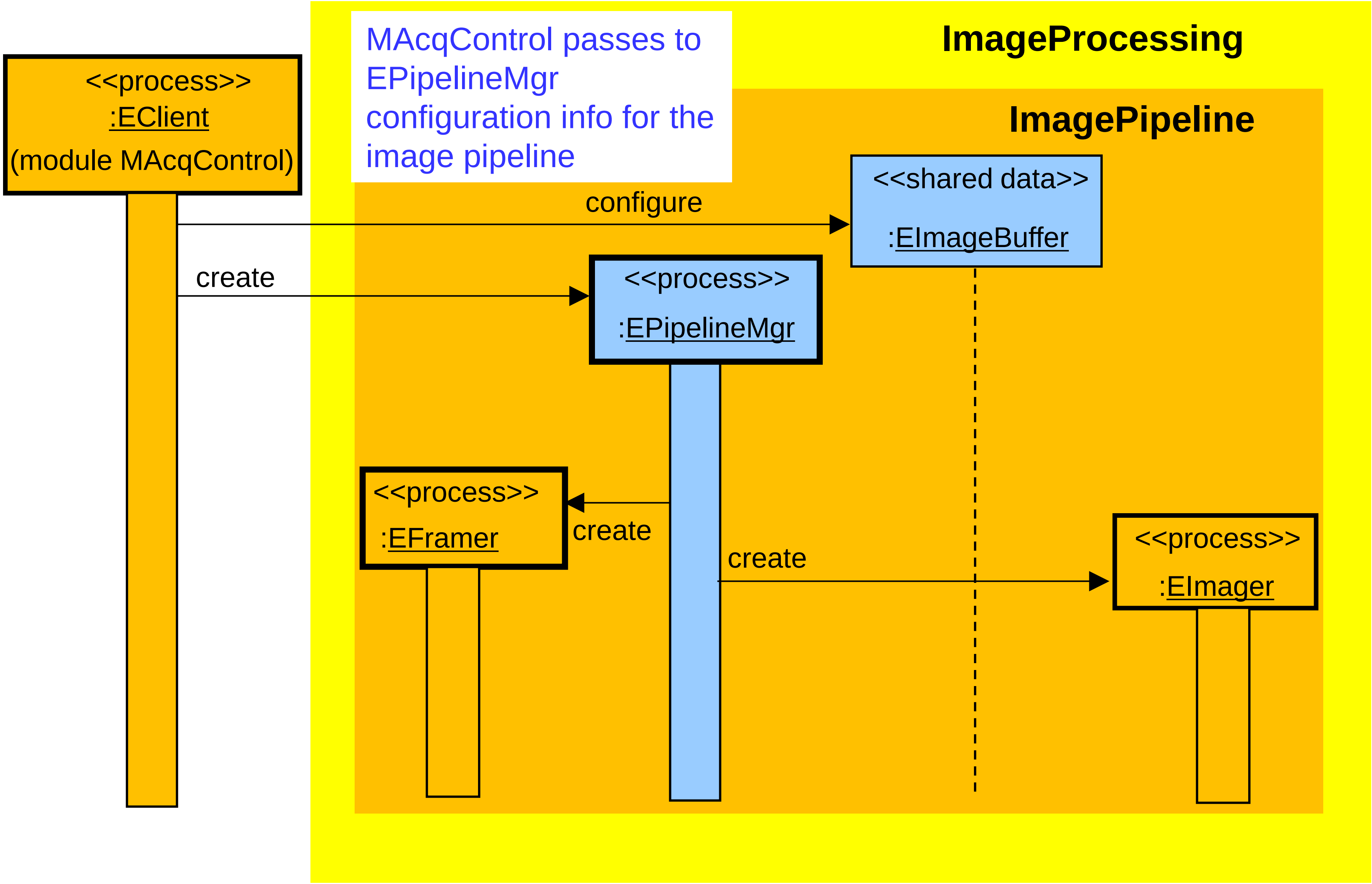



**Figure 6.11 Creation of an image pipeline**

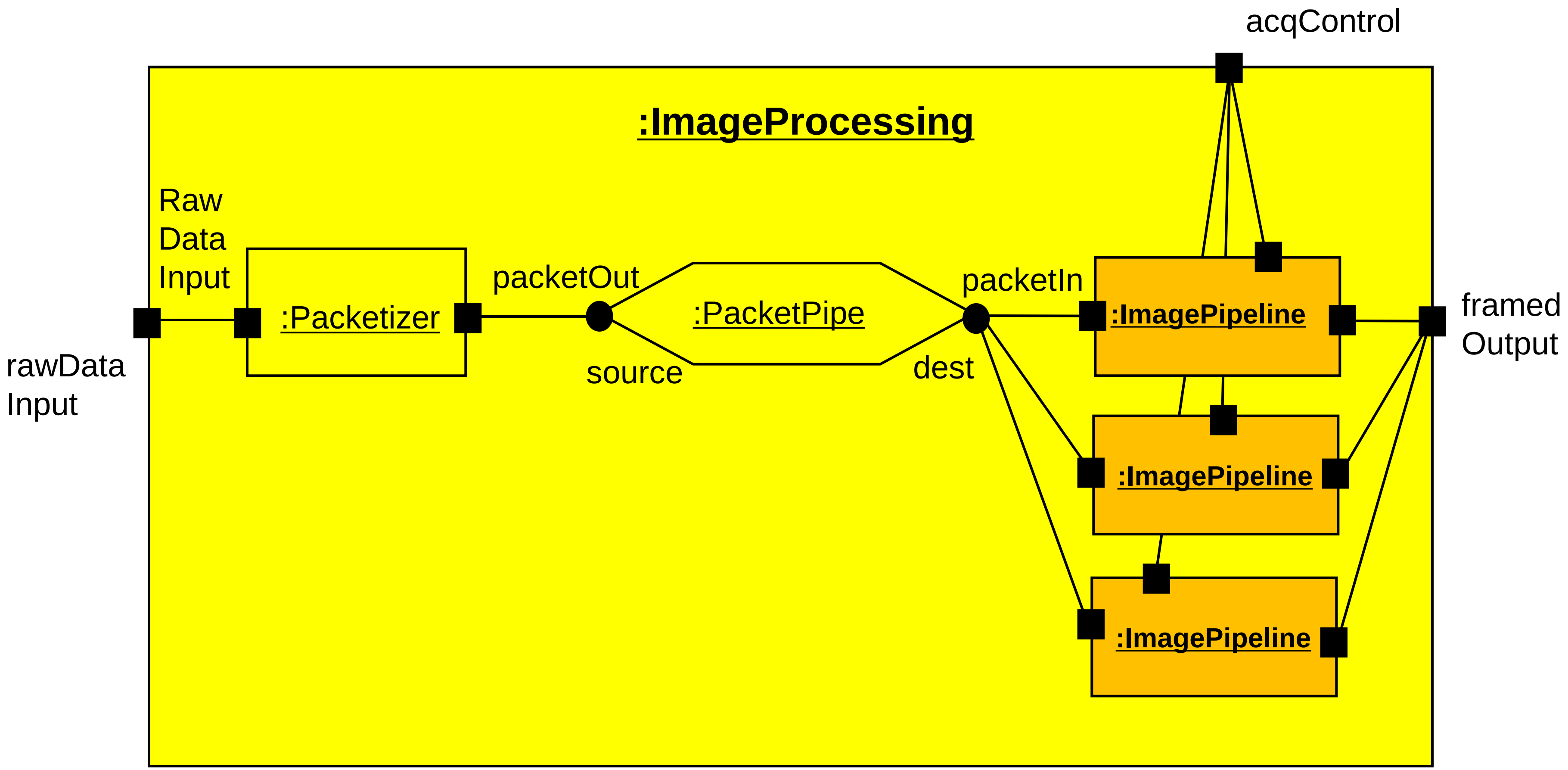
# E. Packetizer

- Realtime performance constraints

  ⇒ Use the strategy: *S9E: Use shared memory to communicate between pipeline stages*

- As before. Although the data is different, the situation is similar.

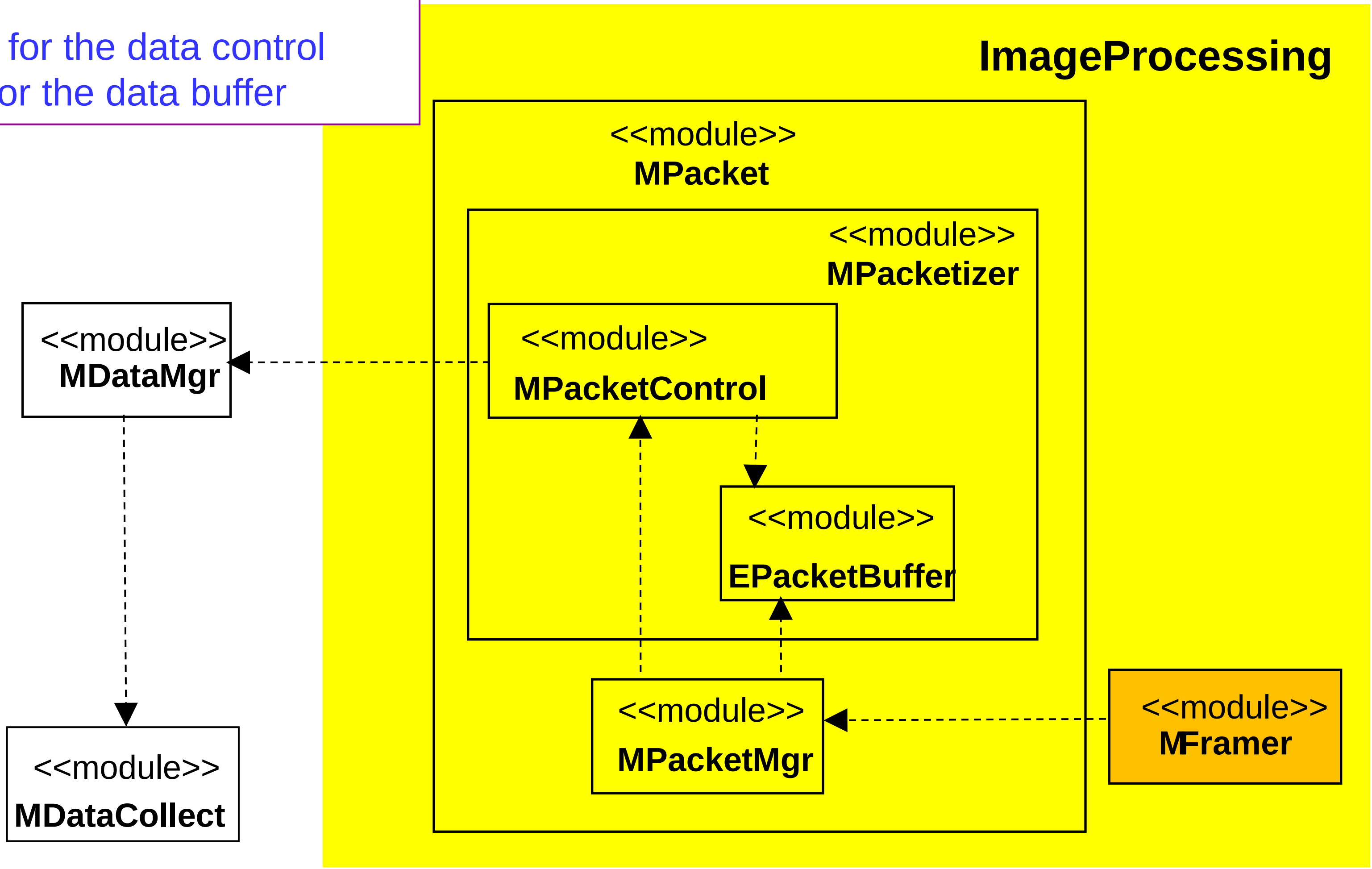
# (From Conceptual View)



# Fig. 6.12. Revisions to the module view: MPacketizer

Split MPacketizer into two modules:

- MPacketControl for the data control
- MPacketBuffer for the data buffer



# Execution View: Packetizer

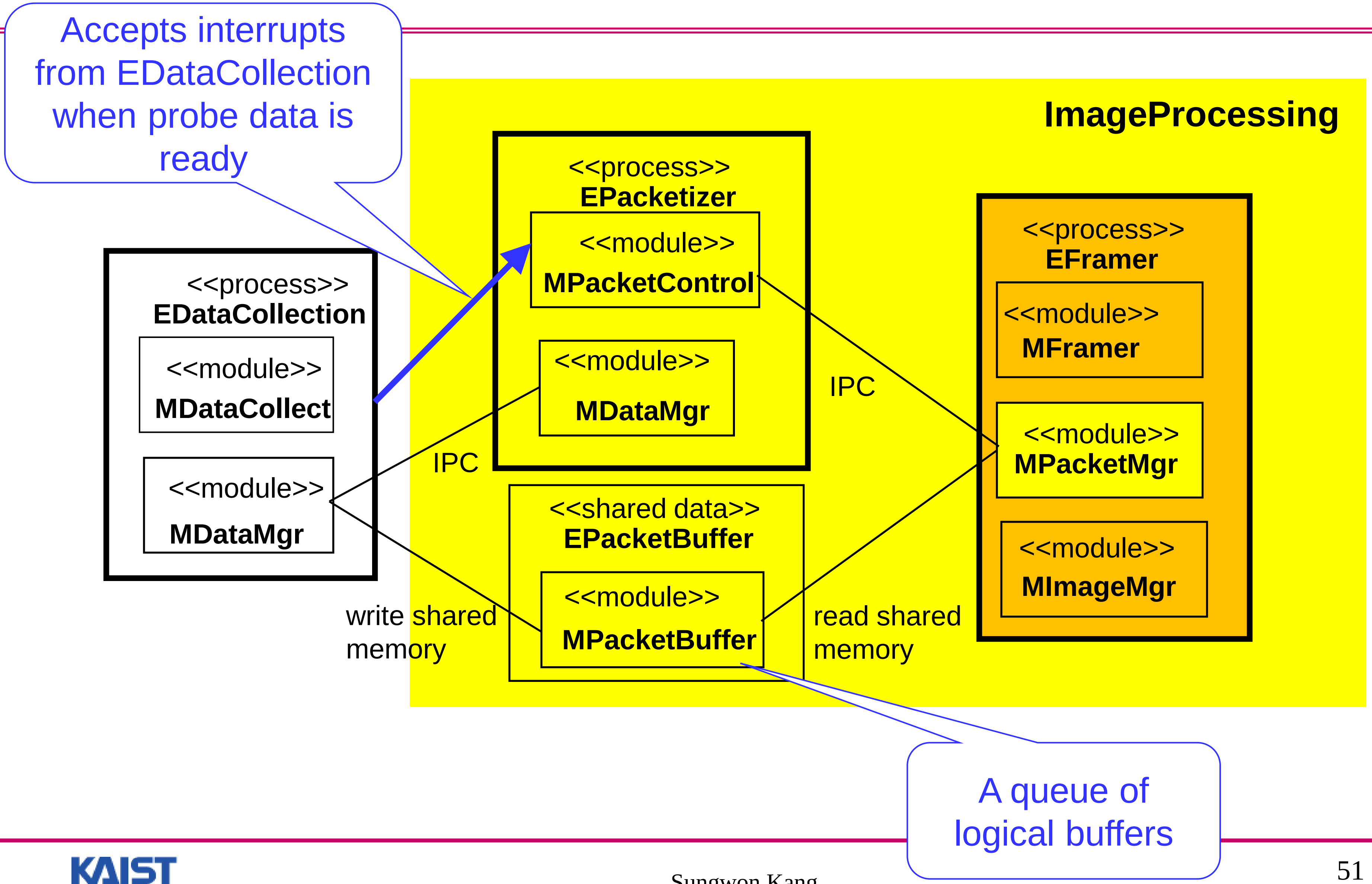
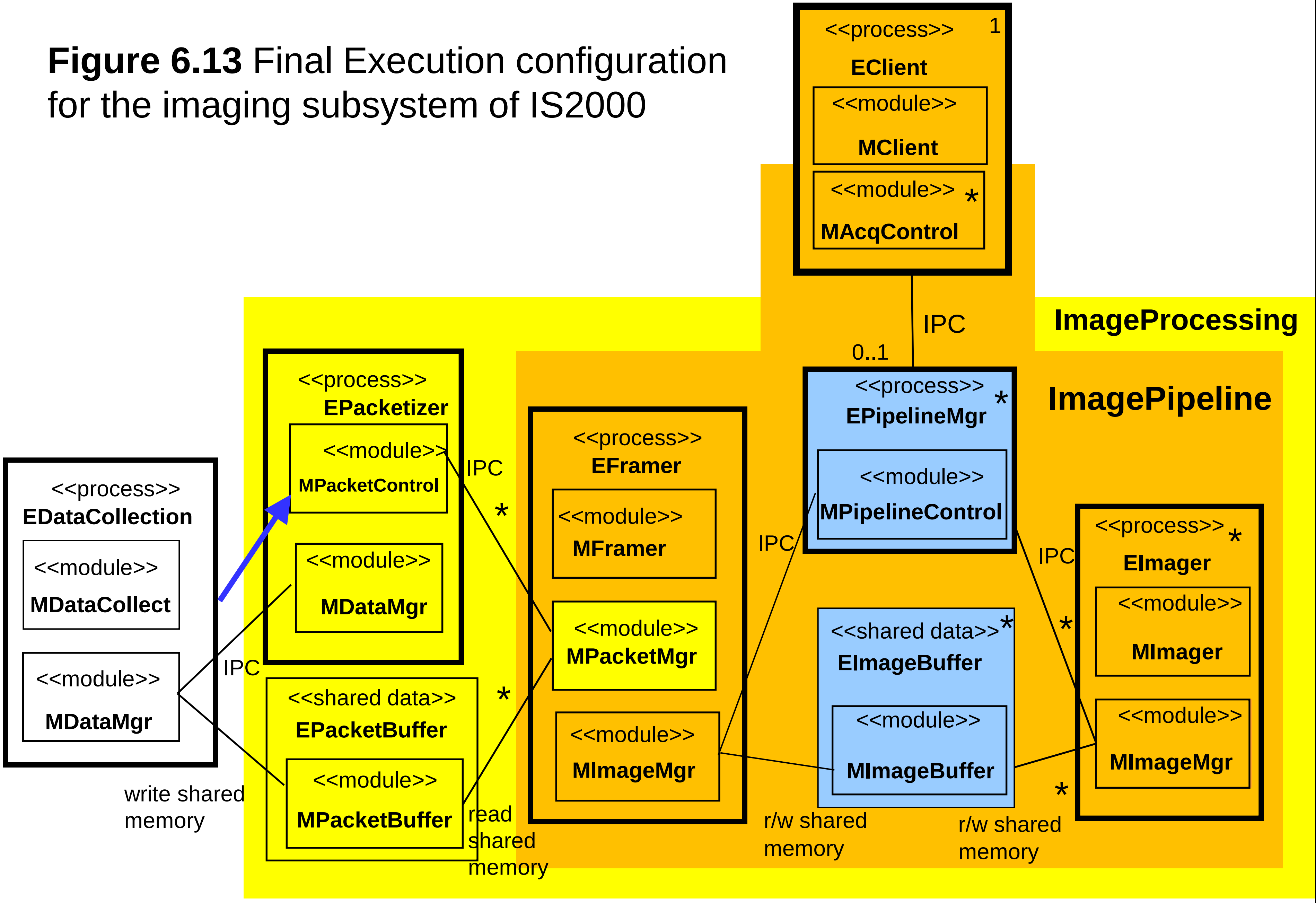


**Figure 6.13** Final Execution configuration for the imaging subsystem of IS2000



# DMA: Direct Memory Access

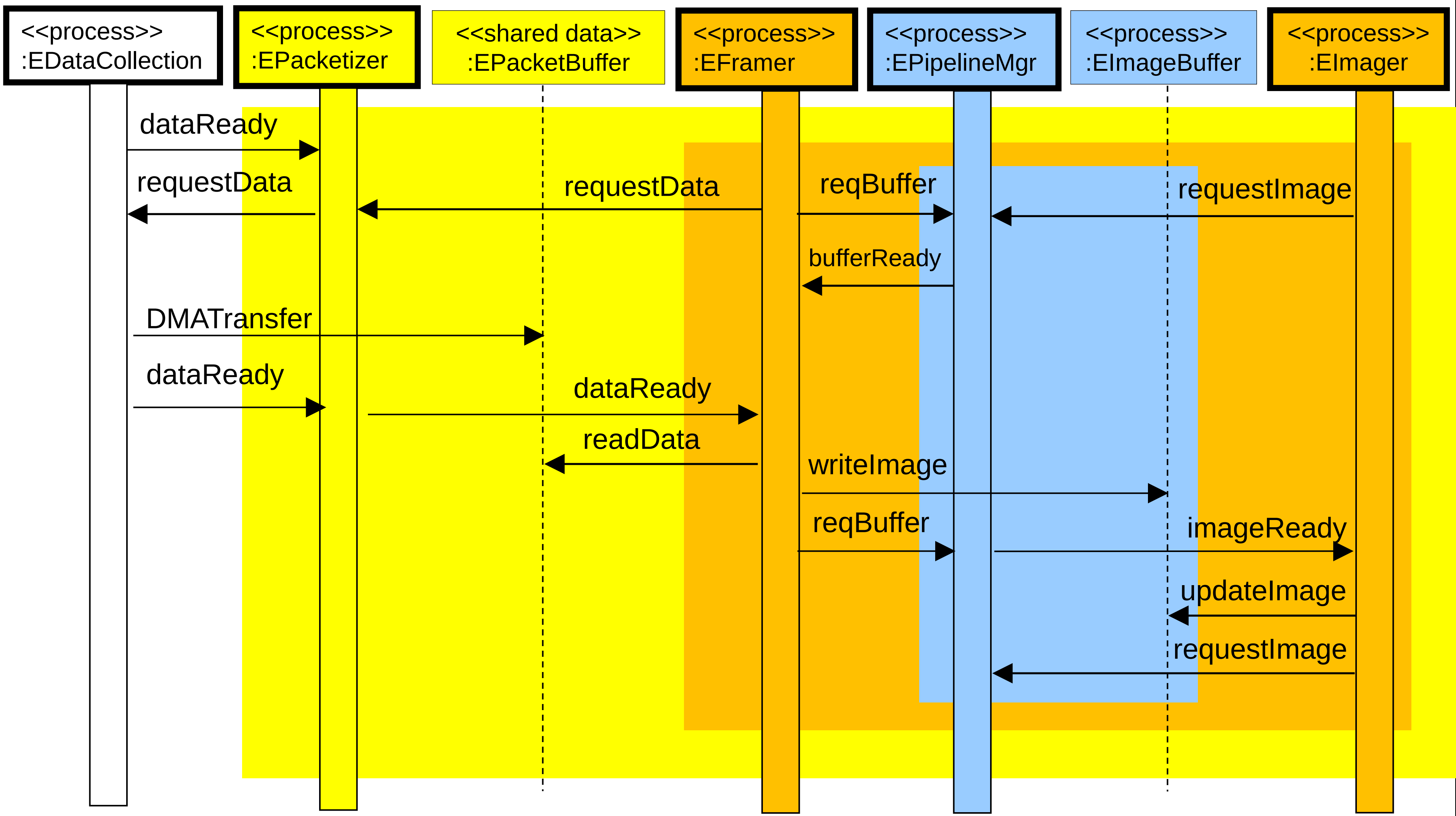



**Fig. 6.14** An example of EPacketizer’s and EPipelineMgr’s interaction

# [3] Final Design Tasks:
## Resource Allocation

# Final Design Tasks: Resource Allocation

- Use the global analysis results to allocate resources to the execution configuration
  1) Allocate a slice of the CPU to each process
  2) Decide how to allocate other resources (e.g. address space, memory pool, timers, proxies, ports) to each process

- Relavant strategies from the Global Analysis:
  1) Allocate a slice of the CPU to each process
     - Initially decided to use one CPU
     - After configuration is complete, we apply
       *S9E: Use RMA to predict performance*
     - The existing architecture does not meet the realtime performance requirements
     - Apply
       *S8B: Use an Additional CPU*
  2) Decide how to allocate other resources

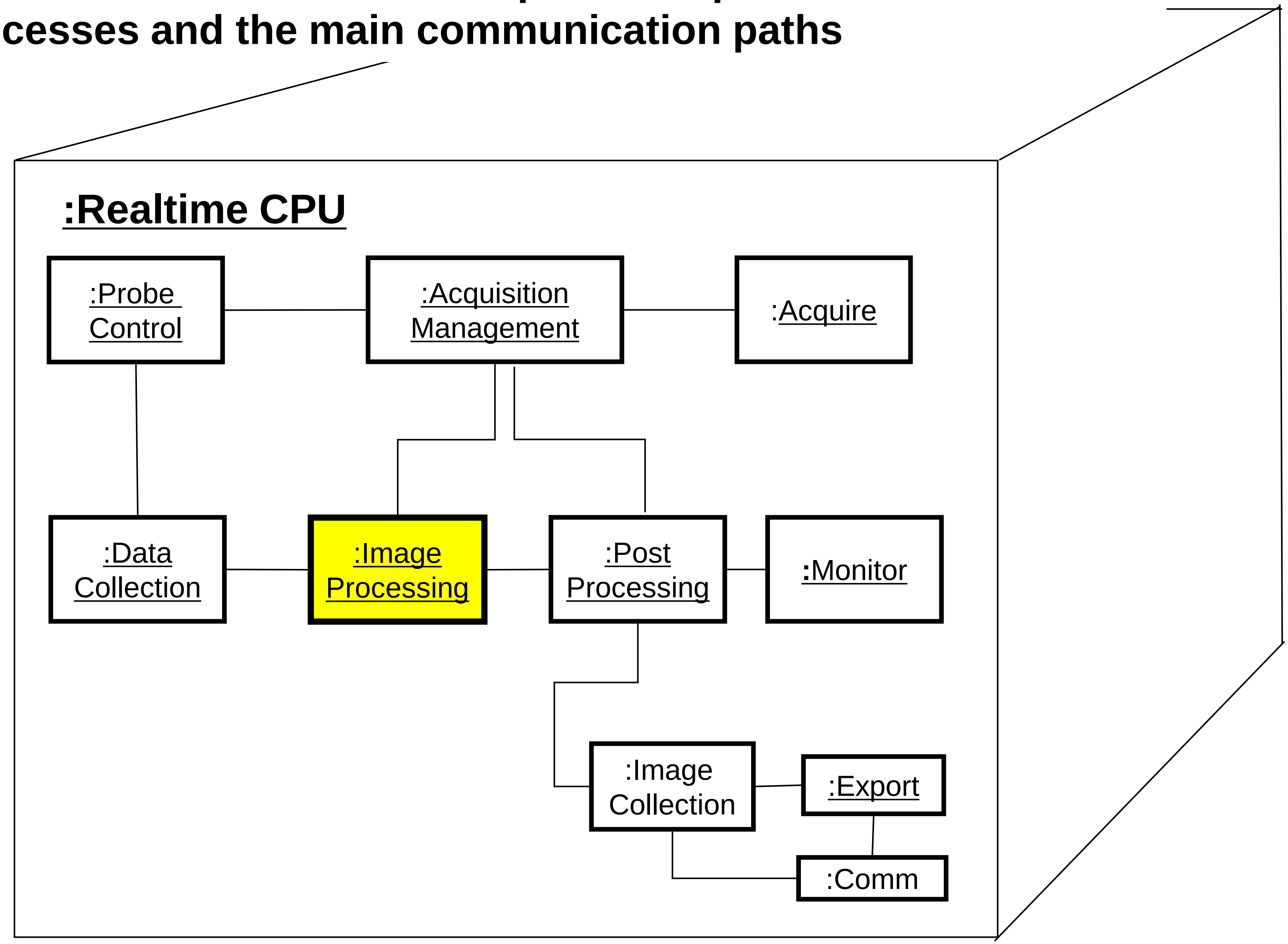
# A. Feedback to the Central Design Tasks

1) 1st CPU will handle the realtime processing and
   2nd CPU will handle the application and GUI

2) Introduce hardware link
   - Define communication paths between CPUs
   - Select communication mechanisms
     => IPC => MCommLib

3) Data from image pipeline to the applications
   => Data between processors
   => Introduce data transfer service
   => New modules
   1. Put in their own processes
   2. Combine the data receiver with the client

**Fig 6.5 shows the main conceptual components as sets of processes and the main communication paths**



# Use an Additional CPU: 1st Step

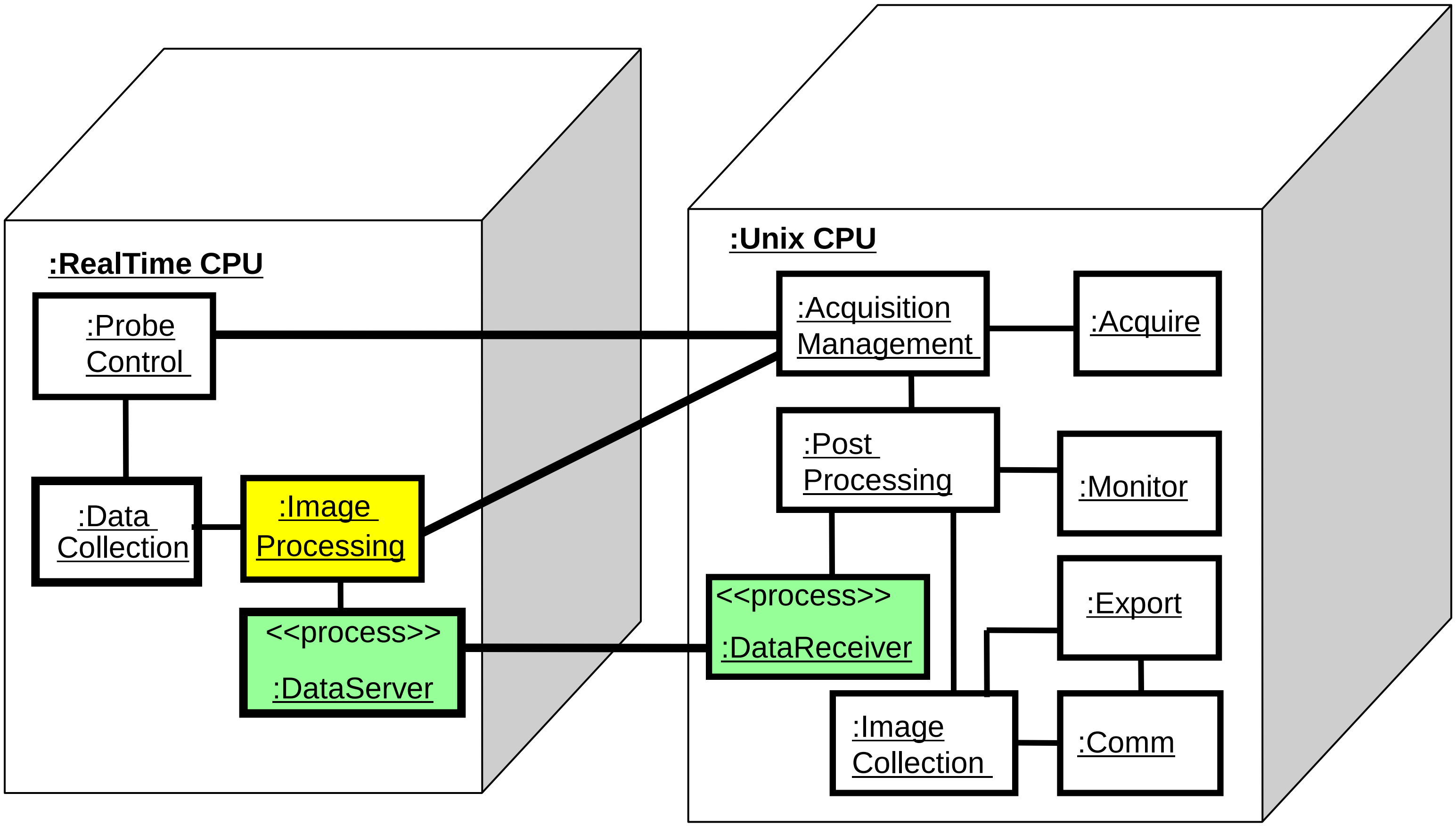
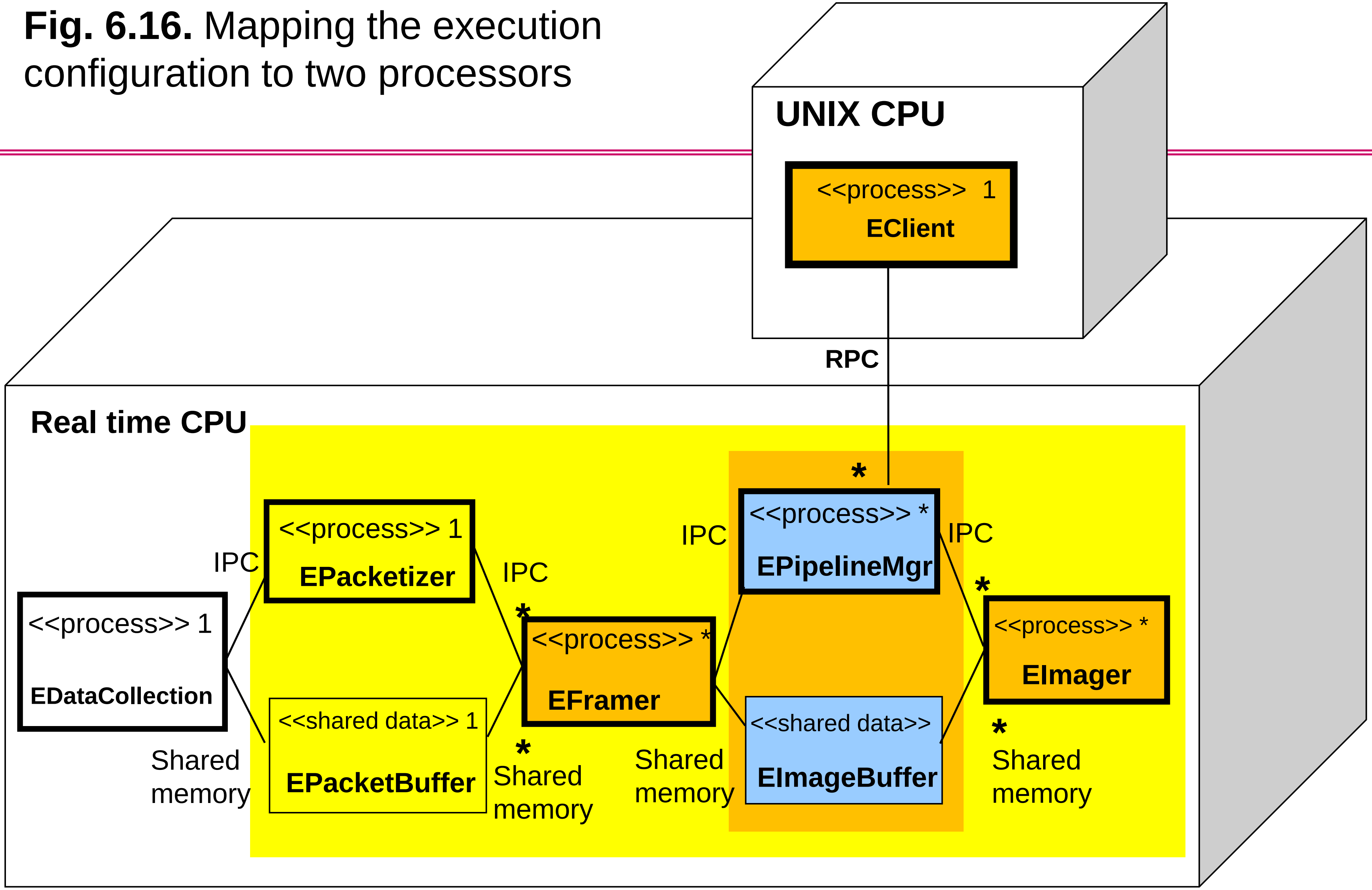
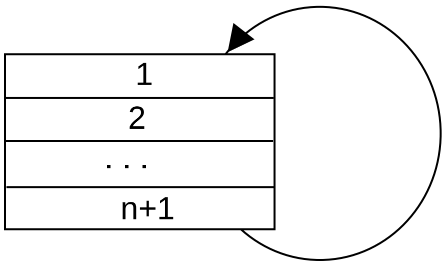



Figure 6.15. Overview of execution architecture view with two CPUs
(Compare with the previous slide)

**Fig. 6.16.** Mapping the execution configuration to two processors



# B. Memory and Other Processing Resources




- Total amount of memory: 64 MB
- Biggest memory users: Packetizer and PipelineManager
  - Implement circular queue buffers
  - Memory: to simplify, preallocate contiguous blocks
    - For PipelineManager
      - Size of image buffer: dependent on the type of application
      - Number of buffers = 1 + number of pipeline stages
    - For Packetizer
      - Size of packet buffer: important (affect throughput and response time)
        - Too large => low data arrival rate => buffer transfer slow
        - Too small => CPU wasted servicing interrupts too frequently
- Extending resources with system support service:
  - UNIX platform supports one real-time interval timer per executing process => UNIX provides a timer service for creating concurrent timers

# Questions?