Chapter 6

Execution Architecture View

The execution architecture view describes the structure of a system in terms of its runtime platform elements (for example, operating system tasks, processes, threads, address spaces). It captures how the system's functionality is assigned to these platform elements, how the resulting runtime instances communicate, and how physical resources are allocated to them. It also describes the location, migration, and replication of these runtime instances.

Because this mapping will likely change over time (including during development), it is important to design the architecture to be easily adaptable to this kind of change. In addition, the resource decisions you make for a component will likely affect the resources available for other components. Because of this interdependency, it is easier to consider this aspect of the system separately from the others.

The driving forces behind the decisions for designing the execution architecture view are performance, distribution requirements, and the runtime platform, which includes the underlying hardware and software platforms. The execution view is used for performance analysis, monitoring, and tuning as well as for debugging and service maintenance.

The execution view is sometimes trivial; as, for example, when the system is a single-threaded, single process. However, as soon as the system has more than one process, the execution view diverges from the module view. An example of this situation is when an application and server are mapped to separate processes. The common practice is to provide a "client API library" with the server. Functionally, in the module view, the client API is part of the server. But physically, at runtime, the client API is part of the application process, not the server process. The execution view makes this runtime mapping explicit.

The execution view also captures replication. Suppose there are multiple client applications, each in its own process, and one server process to serve them all. In this case, the

client API is instantiated multiple times, once in each client application process, whereas the server is instantiated only once, in the server process.

By making the replication explicit, you expose the concurrency requirements the server must meet. Concurrent requests must be handled not in the client API, but in the server itself. The execution view helps to pinpoint where protocols for things like interprocess communication and concurrency are needed.

Another important reason for having an execution view is so that you can better prepare for change. The execution view will likely change more often than the other views for the following two reasons. First, it has a strong dependency not only on the software platform, but on the hardware platform. Even if the user-level functionality of your system were to remain constant throughout its lifetime, you would most likely have to adapt to changes in the hardware and/or software platform due to advances in technology. Second, the execution view is tightly coupled with performance and distribution requirements. For systems with tight performance requirements, you will likely need to do some tuning of the execution view during development. A separate execution view helps isolate the effects of these kinds of changes.

6.1 Design Activities for the Execution Architecture View

The design tasks for the execution view are global analysis, runtime entities, communication paths, configuration, and resource allocation (Figure 6.1). Global analysis is the first task, but throughout the design you can expect some feedback from the later tasks. Similarly, resource allocation is the last task, but decisions made there may cause you to revisit earlier tasks. The middle tasks, the central design tasks, are much more tightly coupled, and one of these is an ongoing global evaluation. Figure 6.1 shows the design tasks of the execution view and their relations to other design tasks.

As in the other views, the issue cards from the global analysis task are input to the central design tasks. The components, connectors, and configuration from the conceptual view guide the design of the execution view, thus these are also input to the central design tasks. Another input to these tasks are the modules, which are mapped to runtime entities in the execution view.

During these central design tasks you may identify new factors, issues, or strategies that cause you to revisit the global analysis task. It is also possible that runtime constraints force you to modify some of the decisions in the conceptual view, or revise the module partitioning in the module view.

After the runtime entities, communication paths, and execution configuration are complete, you perform the final design task—resource allocation. This task could uncover resource constraints that require changes to some of the decisions you made earlier, but that should happen infrequently.

The hardware architecture is an important input to the design of the execution view. For global analysis you will likely have factors that describe aspects of the hardware for

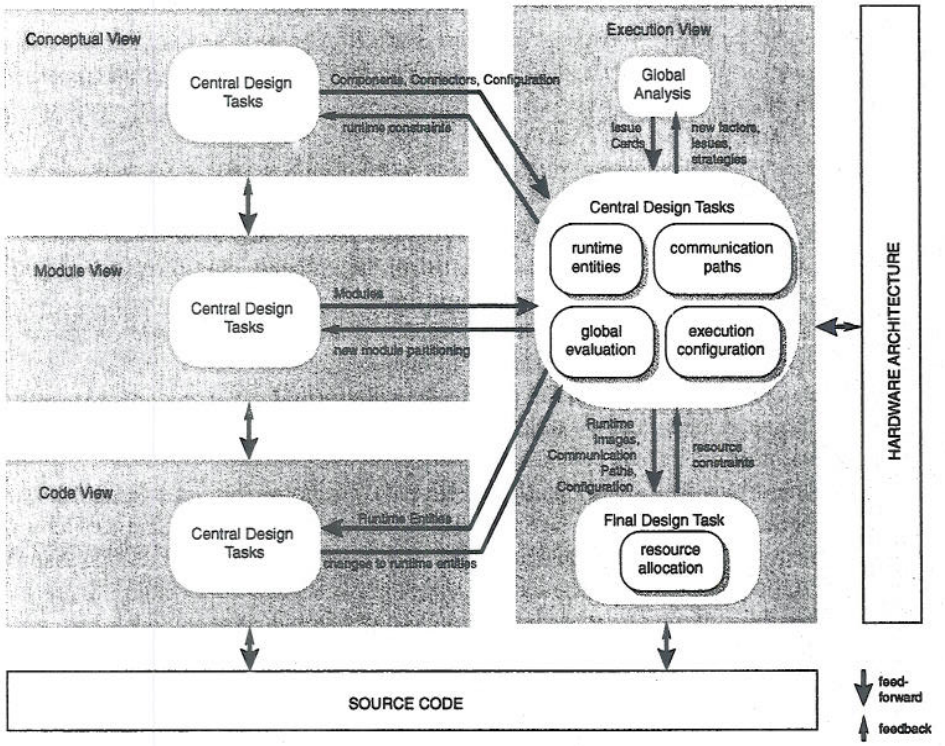

Figure 6.1. Design tasks of the execution architecture view

the system, and strategies related to the hardware architecture. During the central design tasks, the kinds of runtime entities and communication paths you use depend on the hardware. And the execution configuration should, in the end, be mapped to the hardware devices, which happens during resource allocation.

The results of the execution view, in particular the runtime entities, are input to the code architecture view. They also influence the implementation of the system in source code.

6.1.1 Global Analysis

In Chapter 3, we described how to perform the global analysis task. This task precedes all the other steps of the architecture design, and it must be revisited when designing each of the architecture views. You begin the global analysis for the execution view by reviewing the analysis for the conceptual and module views. Identify the factors that affect the execution view; for example, performance requirements and communication mechanisms.

If you haven't already done so, perform an analysis of the hardware platform and the software platform. For the hardware platform you need a list of the hardware components used in the system, and the topology or interconnection of these components. You also need to know which parts of the hardware platform could change, the likelihood of such a change, and when it could occur.

For the software platform you need to know the infrastructure software that is between your product and the hardware platform. Traditionally, this was the operating system. Today, it certainly includes the operating system, but you may also have additional software that is considered to be part of the software platform, such as networking software, other middleware layers on top of the operating system, or something like a DBMS. Often products within a company share a common custom software platform, particularly when they are part of a product line.

Once you've identified the software platform, list the platform elements that you plan to use in the execution architecture view. Figure 6.2 shows the meta-model for a platform element, which can be a process, queue, file, and so forth. You also need to know the basic characteristics of these platform elements. For both UNIX and NT platforms, threads and processes are platform elements. However, because of the difference in overhead for processes, you can expect to use processes more frequently on UNIX than on NT, where you're more likely to use threads.

As with the hardware platform, you determine what could change, and what kind of impact a change would have. Of course, your software platform may have to change as a result of a change in the hardware platform. With this information, you can make informed decisions about where to build in flexibility.

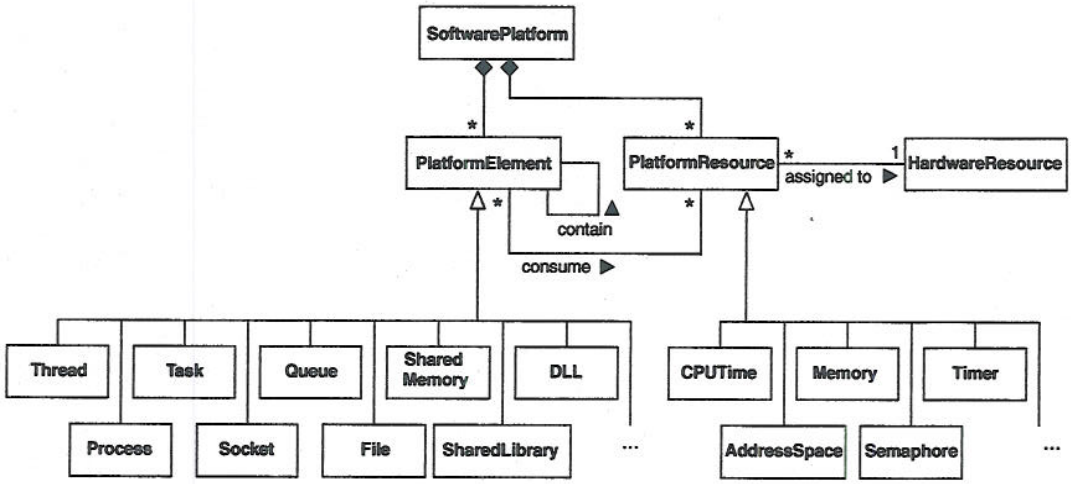

Figure 6.2. Meta-model for platform elements. DLL = dynamic link library.

After analyzing any new factors, record them in your factor tables. Look for new issues, particularly for those related to performance and dependability, and develop strategies for these. You should also eventually record as strategies the resource-sharing and scheduling policies, although you may not know these until the design of the execution view is nearly complete.

6.1.2 Central Design Tasks

Runtime Entities

During the global analysis task you identified the platform elements available on the software platform. Now you must decide how to map conceptual components and modules to these platform elements.

Ultimately the modules will be assigned to runtime entities. Figure 6.3 shows the meta-model for a runtime entity, which can have one or more modules assigned to it, whereas a module can be assigned to more than one runtime entity. A runtime entity is allocated to one of the platform elements defined for the software platform. You start by assigning conceptual components to platform elements for the first approximation of the runtime entities, then refine the partitioning by mapping modules to runtime entities. The end result is a set of runtime entities, their attributes, and the modules assigned to them.

There may also be runtime entities such as daemons or other server processes that have no direct correspondence to modules but are needed to support the other runtime entities. You should also identify and characterize these.

Next you must consider the resource sharing that is allowed or required among the runtime entities. Examples of resource sharing are the sharing of files, buffers, and servers. When definition of the runtime entities is complete, you know which of them will be replicated, and how they will be distributed across hosts. These decisions are recorded as the runtime characteristics of each runtime entity; for example, host type, replication, concurrency control mechanisms used, and other resource-sharing policies used.

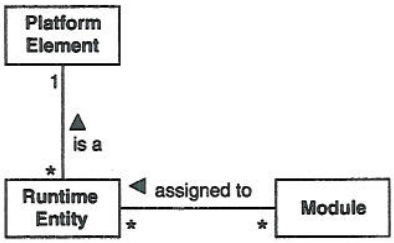

Figure 6.3. Meta-model for runtime entities

Communication Paths

In addition to defining the runtime entities, you need to identify the expected and/or allowable communication paths between them, including the mechanisms and resources used for that communication. Figure 6.4 shows the meta-model for a communication path, which uses a communication mechanism such as interprocess communication (IPC), RPC, the Distributed Component Object Model, and so on. A communication mechanism may use platform elements such as mailboxes, queues, buffers, and files.

The implementation of the protocols for communication paths is often distributed among the runtime entities participating in the communication. You may decide to introduce a new runtime entity if the complexity of the protocol warrants it (for example, links to special hardware).

Execution Configuration

At this point, the building blocks for the execution view are complete. The next step is to describe the system's runtime topology by characterizing the instances of the runtime entities and how they are interconnected.

We make a distinction between a runtime entity and its corresponding runtime instances. If each runtime entity has only one runtime incarnation, then this distinction isn't needed. However, a runtime entity is often replicated (multiple incarnations at runtime), and possibly distributed over multiple hosts. Each of these incarnations is a separate runtime instance. Under UNIX, runtime instances have unique process identifiers, even when they are instances of the same runtime entity.

You should determine each runtime instance and its attributes. These attributes include the corresponding runtime entity and host name. When appropriate, include infor-

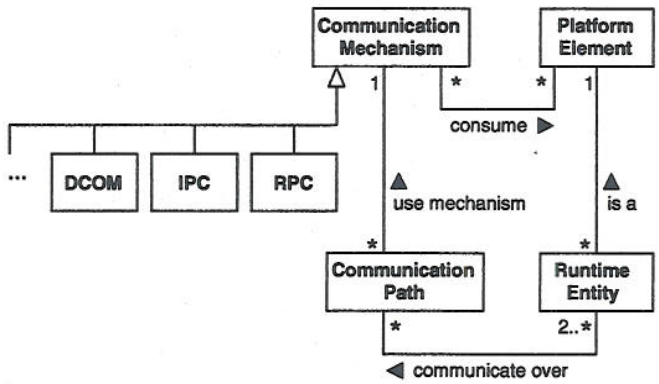

Figure 6.4. Meta-model for communication paths. DCOM = Distributed Component Object Model; IPC = interprocess communication; RPC = remote procedure call.

mation about the resource allocation of each runtime instance, and information about its creation and termination.

Next you need to describe the interconnection of the runtime instances. The interconnection should describe which runtime instances communicate, and it should include temporary as well as permanent communication paths. We do this with an execution configuration diagram. As in the conceptual view, the configuration diagram can contain types (runtime entities) or instances (runtime instances). It is more common for the execution configuration to contain runtime instances, but runtime entities can be useful for describing a set of configurations in a single diagram, as we did for IS2000's image pipeline.

The execution configuration is rarely static; most systems have start-up and shutdown phases in addition to the operating phase. Some systems have a configuration that changes throughout its operation. For example, in IS2000 a new image pipeline is created when an acquisition procedure is requested, and it is destroyed when the acquisition is complete. Thus you need to determine and describe how the configuration changes over time, and how those changes are controlled.

Global Evaluation

During the central design tasks you base your decisions on input from multiple sources. The global analysis gives you strategies for fulfilling performance and dependability requirements. The conceptual view design describes the concurrency among conceptual components, which the execution configuration must support. The modules and their dependencies constrain the runtime entities and how they communicate. The hardware architecture dictates the hardware resources and constrains the software platform, limiting your selection of platform elements and communication mechanisms. In addition, you must consider the implementation cost and try to avoid complex algorithms for implementing concurrency control, communication, and so forth. An ongoing evaluation task is to balance all of these guidelines and restrictions.

During global evaluation you may also need to do performance experiments and/or simulations for the evaluation; analytic techniques may not be sufficient. Based on the results of your ongoing evaluation you must then decide whether to adjust or to refine the boundaries of your runtime entities, and modify their characteristics accordingly.

6.1.3 Final Design Task: Resource Allocation

The remaining task is resource allocation. Here you take the runtime instances and budgets defined in the configuration task, allocate them to particular hardware devices, and assign specific values to the budgeted attributes (for example, by setting process priorities).

The resources to be allocated were identified during the global analysis task. Once the hardware and software platform was defined, you determined the resulting resources. The software platform may have a fixed number of each type of platform element, or this number may be configurable.

The allocation decisions made here are fairly localized, and are often made at build time. The intention is to use standard techniques and specific strategies from the global analysis to determine how resources are allocated. Example decisions are that these processes are assigned 256K of shared memory, or that rate monotonic scheduling (RMS) is used to assign priorities to the processes.

For larger, more complex systems, it is useful to deal with more than one resource or process at a time. For example, when there is a need to guarantee that a low-priority task eventually gets done, the set of processes responsible for this task may be assigned a common CPU time guarantee and budget (assigning them, in a sense, to a virtual CPU). Resources may be allocated to a collection of processes responsible for related functions, and therefore have a related cumulative need for processing resources. If it turns out that there are not enough resources, then you have to revisit the decisions made earlier during the central design tasks.

6.2 Design of Execution Architecture View for IS2000

Now that we have summarized the design tasks, let us return to the example system to show how to design the execution view for IS2000.

6.2.1 Global Analysis

The global analysis we've done so far hasn't produced many strategies that are directly applicable to the execution view. The few relevant strategies are the following:

Issue: Skill Deficiencies

Strategy: *Avoid use of multiple threads.*

Strategy: *Encapsulate multiprocess support facilities.*

Performance is a very important concern for the execution view. Because of the very high data rate of the probe hardware, let's add a new issue: High Throughput.

To keep the product costs down, let's use a single CPU and limit memory size to 64MB. Because of our real-time requirements, the processor is a high-end Pentium processor. However, if we can't meet our performance requirements with a single CPU, we have no choice but to add another. This second CPU can run UNIX, and the first CPU can be reserved for the real-time processes. We then need to add the strategy *Use an additional CPU* to the issue card.

A common technique to achieve higher performance is to increase concurrency by using multiple threads and/or multiple processes. In our case, the development team is deficient in the necessary skills, particularly in multithreaded processing, which caused us to create the strategy *Avoid use of multiple threads*. Another factor is that with UNIX, our selected operating system, it is relatively inexpensive to create and to destroy processes. So let's add another new strategy, *Map independent threads of control to processes*.

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.

Continued

High Throughput (continued)

Related Strategies

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

Next let's revisit the Issue Real-Time Acquisition Performance, which has only one strategy so far, added during the design of the conceptual view. IS2000's real-time performance requirements are given as the maximum signal data rate, which is the rate at which the probe control can acquire data, and acquisition performance. Acquisition performance is measured by the size and number of images, and the acquisition response time measured in terms of end-to-end deadlines.

A common strategy is to run a simulation or other model of the system to estimate its performance. Although the results are only an estimate, they can provide valuable feedback at this early stage—during architecture design—rather than later, after many of the components have been implemented. Let's add the strategy *Use rate monotonic analysis (RMA) to predict performance.*

The strategies developed to support the issue High Throughput say that processes are the basic unit for the execution view, and that we may need to add a second CPU. Even with an analysis technique for predicting performance, we may still need to adjust process boundaries as the system is implemented. Therefore, let's add two more strategies: *Use flexible allocation of modules to processes* and *Develop guidelines for module behavior.* These should reduce the cost of adjusting process boundaries during development.

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

Continued

Real-Time Application Performance (continued)

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).

Lastly, let's look at a related issue, Resource Limitations. These are driven mainly by budget and technological factors, and they have a large impact on the design of the execution view.

To provide support for meeting the real-time processing requirements, let's use QNX, a UNIX-like operating system that supports real-time processes. With the exception of

QNX proxies, we'll use only those features that are POSIX compliant. This means that the operating system could be replaced with another POSIX-compliant operating system.

The size of the memory is limited. Due to budget limitations, it is not likely to be increased. Operating system processes consume software resources such as memory, so too many active processes can degrade system performance. However, it is relatively inexpensive to create and to destroy processes on the selected operating system. Thus let's create a new issue card, for Resource Limitations, and add to it the strategy *Limit the number of active processes.*

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.

6.2.2 Central Design Tasks: Runtime Entities, Communication Paths, and Configuration

With the global analysis under way, let's turn to the central design tasks of the execution architecture view: defining runtime entities, communication paths, and configuration. As with the central design tasks of the conceptual and module views, these can't be done in a strict sequential order. Although there is potential feedback among many other analysis and design tasks, these central design tasks are even more tightly coupled.

In this section the central design tasks for portions of the example system are presented. You'll see how the results of one task feed back into the other immediate tasks, and sometimes into tasks in the other architecture views.

Begin Defining Runtime Entities

A good starting point for the central design tasks is to begin by associating each high-level conceptual component with a set of execution elements. In the case of the example system, let's avoid multithreaded processes if possible and, instead, put each thread in its own process. Thus we begin by drawing the picture shown in Figure 6.5. This shows the main conceptual components as sets of processes, and shows the main communication paths between them.

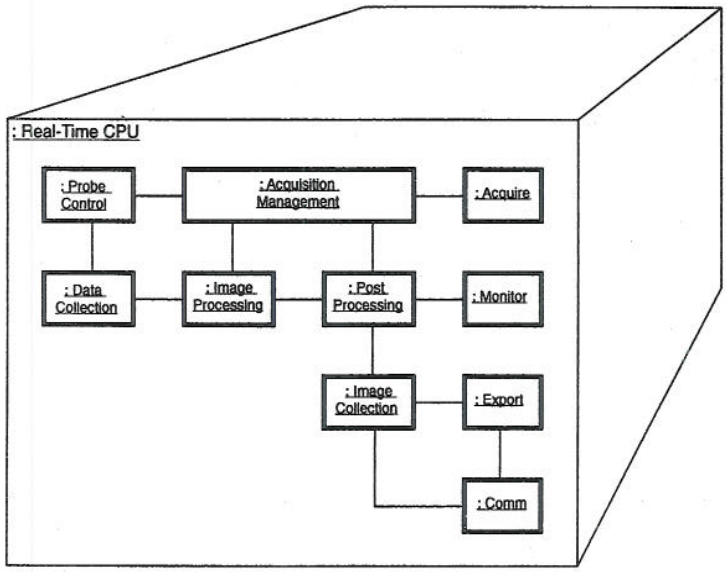

Figure 6.5. Overview of execution architecture view

In Figure 6.5 we used a UML Deployment Diagram to show on which hardware resource these sets of processes will run. The hardware resource, in this case the real-time CPU, is shown as a node instance.

Next let's take each of these sets of processes in turn and go through the central design tasks for each. Let's start with the higher risk parts of the system first. For the example system, the ImageProcessing component is the computationally expensive part of the real-time portion of the system, so it should be examined early in the execution view design. In the rest of this chapter, we use the ImageProcessing component to explain the execution view design.

The heart of the ImageProcessing component, as we defined it in the conceptual view (Figure 4.6), is the ImagePipeline. In the module view we mapped the conceptual elements to modules and determined the decomposition relationships and use-dependencies. Figure 6.6 summarizes these relationships.

Although the imaging subsystem may contain many pipelines, they all follow the same pattern, so let's use just one to illustrate the design tasks. Recall that in the module view we mapped the first pipeline stage to MFramer and each of the later pipeline stages to a separate module, for example MImager. In this chapter we use MImager as a placeholder for the multiple, later stages that may be part of the pipeline.

We know that all modules in Figure 6.6 have to be mapped to a runtime element. There's no fixed rule for how to do this, but the approach we use here is a common one: We start with the assignments that are most straightforward, then let those decisions constrain the later assignments.

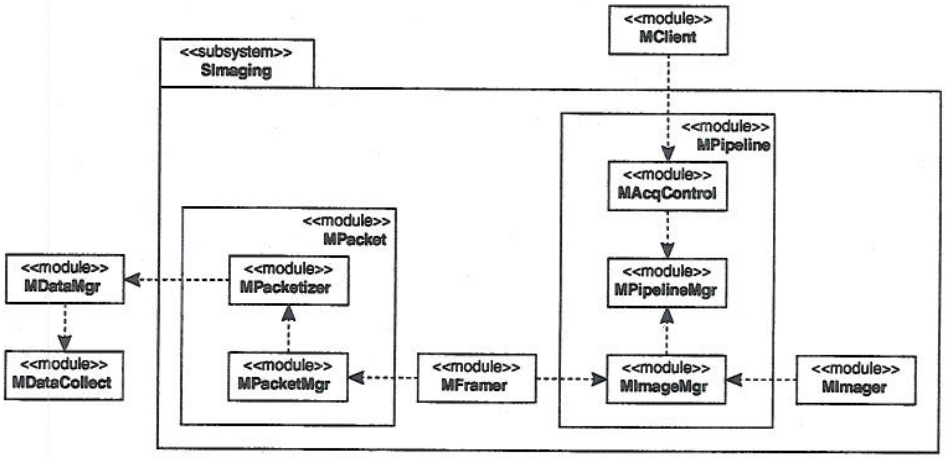

Figure 6.6. Modules in the imaging subsystem (from the module view)

When there is a simple one-to-one correspondence between conceptual components and modules in Table 5.2, we assign a module to a process or a thread. This is a straightforward implementation of the concurrency expressed by the conceptual view. Because of the strategy *Map independent threads of control to processes*, let's create separate processes for each of the pipeline stages, and for the image pipeline client (MClient) and the data collector (MDataCollect).

Next let's examine the dependencies among these modules to determine the resulting communication paths and mechanisms between the processes. First let's look at the communication between the image pipeline client and the imaging subsystem. This takes place through MAcqControl. The MClient module accesses the MAcqControl module to initiate and to control the image pipelines. If MClient and MAcqControl were in different processes, they would have to communicate across process boundaries. Instead, let's link all of the MAcqControls into the process with MClient so communication is via a local procedure call.

These four processes are shown in Figure 6.7. We use a UML stereotyped class for a runtime entity, so these have the stereotype <<process>>. Because a process has a thread of control, it is modeled as an active class, which has a thick border. The modules assigned to each process are nested inside it. The EClient process contains module MClient and multiple MAcqControl modules, one for each image pipeline. Let's use UML multiplicities to show this, and use the convention that the multiplicity is 1 when none is shown. Thus the asterisk in the upper right corner of MAcqControl indicates that there are zero or more of this type of module linked to EClient.

Pipeline Manager

This brings us to the communication between individual pipeline stages, where performance is a major concern. We don't yet have any strategies that address this in the issue Real-Time Acquisition Performance. Because shared memory is available on our operating system, let's use it for sharing images between pipeline stages. This is a new strategy, so we need to add it to the issue card.

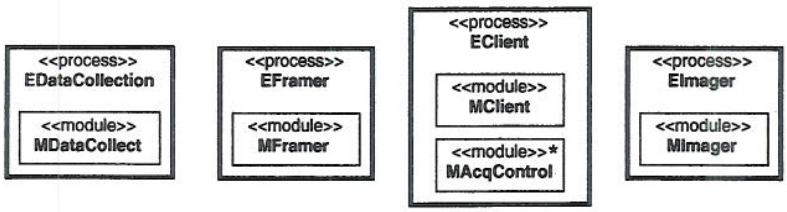

Figure 6.7. Initial processes for the imaging subsystem

Real-Time Acquisition Performance (continued)

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

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

In the module view, the responsibility for managing the image buffers was assigned to module MPipelineMgr. Shared memory is a runtime element—something that is visible to and managed by the operating system—so let's split the MPipelineMgr module into two parts: one for the management of the shared memory pipeline and another for the shared memory itself.

Next we have to go back and revise the module view, splitting the MPipelineMgr module into two modules: one called MPipelineControl for the image pipeline control and one called MImageBuffer for the image pipeline buffer. MPipelineControl coordinates the pipeline by controlling access to the shared memory MImageBuffer. These revisions are shown in Figure 6.8.

Next let's map these new modules to runtime entities. We could replicate the control module and link it to each process containing a pipeline stage. Because MPipelineControl coordinates the pipeline stages, this would force it to use a distributed control algorithm.

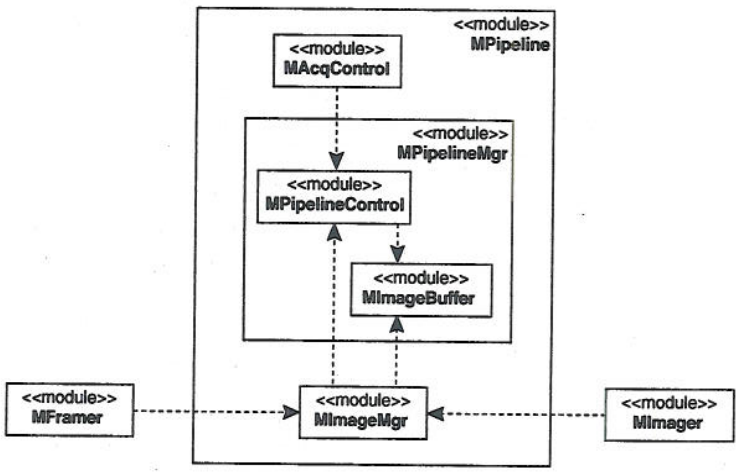

Figure 6.8. Revisions to the module view: MPipelineMgr

Distributed control requires a more complex handshaking protocol than centralized control, and because here the communication is between processes, distributed control is also more costly.

A simpler solution is to centralize pipeline control in a single process, separate from the pipeline stage processes. Because each image pipeline has its own manager, and active image pipelines run concurrently, each MPipelineMgr must be in a separate process. Thus let's create a new process, called EPipelineMgr, that contains MPipelineControl. Although it controls access to the shared memory area MImageBuffer, it does not read or write to MImageBuffer, so there is no communication path between them in the execution view.

Communication Paths for the Pipeline Manager

With the pipeline manager mapped to a separate process, let's use IPC for its communication with MAcqControl because the modules are in different processes. The same is true for communication between the MFramer and MImager modules and MPipelineControl.

This introduces a new technological factor, so we need to return to the global analysis to add this factor and analyze its impact. The use of IPC mechanisms requires resources such as sockets or mailboxes. Such resources are limited on a real-time operating system, so we may need to adjust our usage during development. Table 6.1 shows the new factor.

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 mailboxes. 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

Because it uses other resources, we need to add this factor to the Resource Limitations issue card. Now we also need a strategy that addresses the resource limitations for IPC connections, so let's add a new strategy:

Resource Limitations (continued)

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.

Next let's apply the strategy *Encapsulate multiprocess support facilities* to reduce the burden of using IPC. As a result, we need to create a new module, called MCommLib (communication library), to handle the details of the IPC protocol. This module is linked to the processes EFramer, EImager, EClient, and EPipelineMgr, but to enhance readability it is not shown in the diagrams. Of course, this module must also be added to the module view.

The decisions we've made so far appear in Figure 6.9. We've used the UML association notation (a solid line) for communication paths. These are labeled to show the type of communication. Note that EImageBuffer, the shared data area, doesn't have a thread of control, so it doesn't have the thick border that the active classes have.

We should also consider resource-sharing policies and protocols. The MImageBuffer module, the shared memory for an image pipeline, is shared among the stages for that pipeline. Let's split the shared memory into multiple logical buffers, one for each stage in the pipeline. A pipeline stage has exclusive access (read and write access) to one of these logical buffers, and MPipelineControl controls this exclusive access.

These logical buffers must also be "passed" down the pipeline, and the pipeline stages initiate this transfer by requesting a new buffer, which also releases the old one. The MPipelineControl module accepts requests first come first served. The MPipelineControl module, because it knows the configuration of the pipeline, determines which buffer is to be allocated to the requesting stage. If the buffer is available, access is granted immediately. Otherwise, it continues to service new requests and releases buffers to pending requests as the buffers become available.

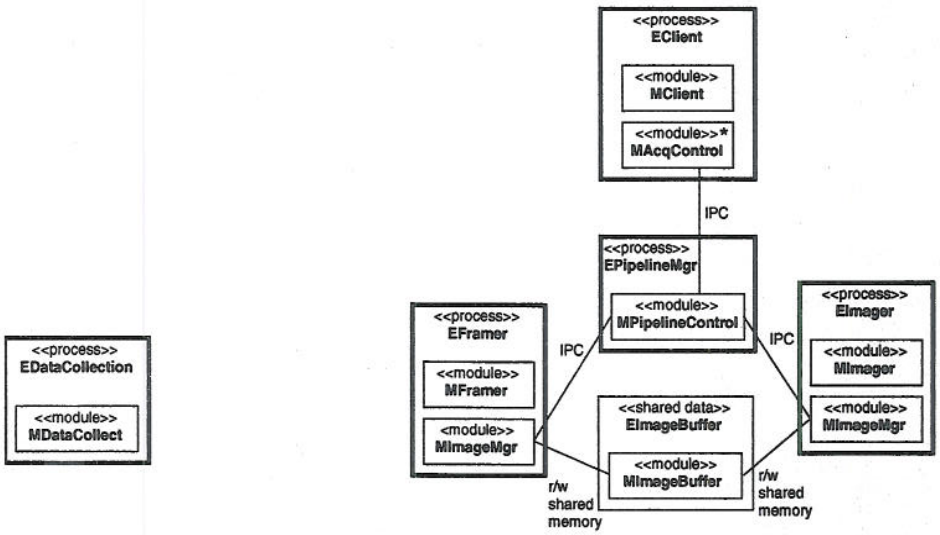

Figure 6.9. Processes and communication paths for the pipeline manager. IPC = interprocess communication; r/w = read/write.

Assigning Multiplicities to the Execution Configuration

Next we come to the question of exactly how many processes we expect to have at runtime. This is part of the configuration design task.

Because each image pipeline contains approximately 4 pipeline stages, and there are at least 10 image pipelines, the pipelines alone could use more than 40 processes. The strategy *Limit the number of active processes* indicates that, because of memory limitations and performance requirements, we can't keep all of these alive for all types of acquisitions and monitoring. The processes for an image pipeline must be created dynamically when the acquisition procedure is requested, and then they must be destroyed when the acquisition is complete.

For systems in which the configuration is fixed, it can be described in a single diagram. But you can also describe a set of configurations in a single diagram, as we did in the conceptual view. For IS2000's execution configuration, let's describe in one diagram all possible configurations of the imaging subsystem during normal operating mode.

Some of the processes have only one runtime instance and exist throughout the lifetime of the system. So far this includes only EClient and EDataCollection. In the configuration diagram (Figure 6.10), these processes have a multiplicity of 1, which is shown in the upper right-hand corner of the EClient and EDataCollection processes.

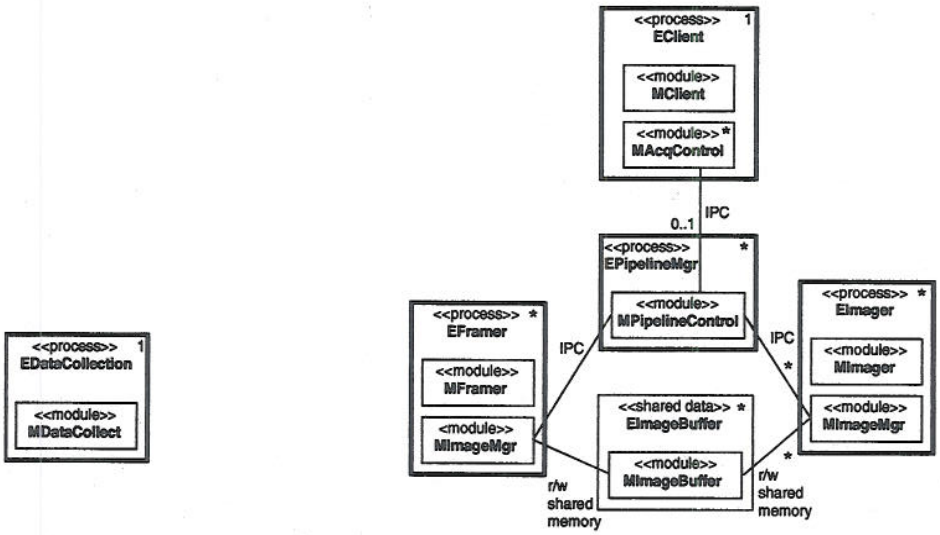

Figure 6.10. Adding multiplicities to the execution configuration for IS2000

Linked into the EClient process are multiple MAcqControl modules, one for each image pipeline that could be created during system execution. Each MAcqControl module creates an EPipelineMgr process, passing to it configuration information for the image pipeline. The MAcqControl module also configures the shared memory for its associated image pipeline.

The EPipelineMgr process in turn creates the pipeline stages. It launches a process for the stage, and binds the stage to the process group by putting it into the correct stage of the pipeline and giving it access to shared memory. We've used a UML Sequence Diagram to show this pipeline start-up (Figure 6.11). The objects in the diagram are runtime instances, and their vertical positions show the order in which they come into being. By looking at the line below each object, you can see its lifetime and the messages it sends. Active objects have a solid line, and passive objects, like the EImageBuffer shared memory, have a dotted line.

The imaging subsystem contains multiple EPipelineMgr processes, one for each image pipeline. Each image pipeline contains exactly one EImageBuffer, one EFramer process, and one or more additional EImager processes. These multiplicity relationships are summarized in Figure 6.10.

In a configuration diagram, the communication paths also have multiplicities. These are marked at each end of the communication path. If no multiplicity is shown, it is understood that it is 1.

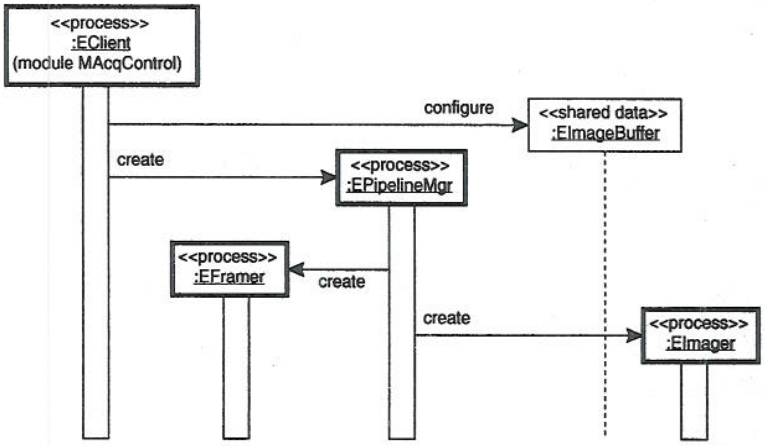

Figure 6.11. Creation of an image pipeline

Marking multiplicities on the communication paths constrains the configuration, so we must figure out how many of each process can exist relative to another. For example, there is exactly one EClient process, and it contains multiple MAcqControl modules. There are multiple EPipelineMgr processes, each containing exactly one MPipelineControl module. Next let's put multiplicities on the path between MAcqControl and MPipelineControl to show that each MPipelineControl communicates with exactly one MAcqControl, and each MAcqControl communicates with either zero or one MPipelineControl modules. The result of this constraint is that at any point in time there is at most one EPipelineMgr process for each MAcqControl module linked to the EClient.

EFramer and EImager both have a multiplicity of "*," which says only that there is zero or more of each of them. But by looking at the multiplicities on their communication paths, you can see that there is one EFramer and multiple EImager processes per pipeline.

Packetizer

Next let's look at the process for the Packetizer component, which passes data from the probe data collector (MDataCollect) to all existing image pipelines. In the module view we implemented its functionality with the MPacketizer module.

Again we must consider real-time performance constraints, so let's apply the strategy *Use shared memory to communicate between pipeline stages*, as we did for the image data within the image pipeline. Although the data is different, it is a very similar situation, in that the probe data is shared among multiple processes.

So we follow similar reasoning, and split the MPacketizer module into two modules: one for data control (MPacketControl) and one for the data buffer (MPacketBuffer). To keep the control algorithm simple, let's create a separate process—EPacketizer—for MPacketControl. Next we must add this to the list of processes that have only one runtime instance and that exist throughout the lifetime of the system.

The MPacketControl module controls access to the shared memory MPacketBuffer. This shared memory is written to by the EDataCollection process and it is read by the first pipeline stage of each image pipeline. MPacketControl must accept interrupts from EDataCollection as the probe data becomes available, and it must handle read requests from the image pipelines. These new modules must be added to the module view (Figure 6.12).

Communication between these processes is via IPC and shared memory. As before, the communication paths are derived from the dependencies shown in the module view. The resulting communication paths are shown in Figure 6.13.

However, access requirements for the shared memory MPacketBuffer are different than for MImageBuffer. Here we need to organize the shared memory into a queue of logical buffers. Buffers are queued by the probe data collector, but a buffer is dequeued only after all image pipelines have read it. The pipelines can read buffers concurrently, but MDataCollect must have exclusive access to a buffer in order to write. The MPacketControl module must enforce this protocol and ensure that each image pipeline receives the buffers in the correct sequence.

This protocol information is part of the definition of the communication path between EPacketizer and its users. A UML Sequence Diagram can be useful for showing this kind

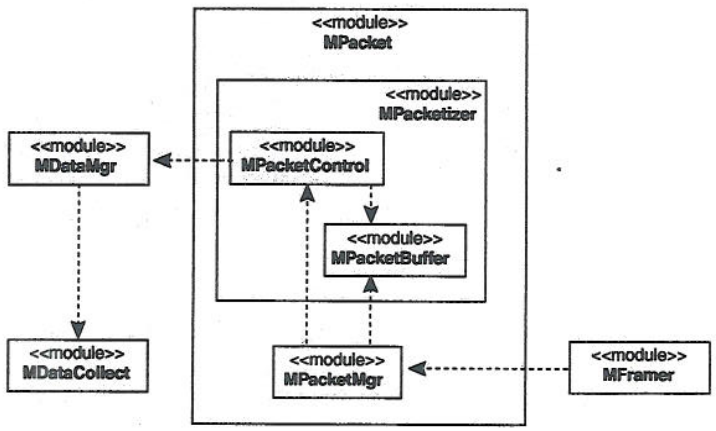

Figure 6.12. Revisions to the module view: MPacketizer

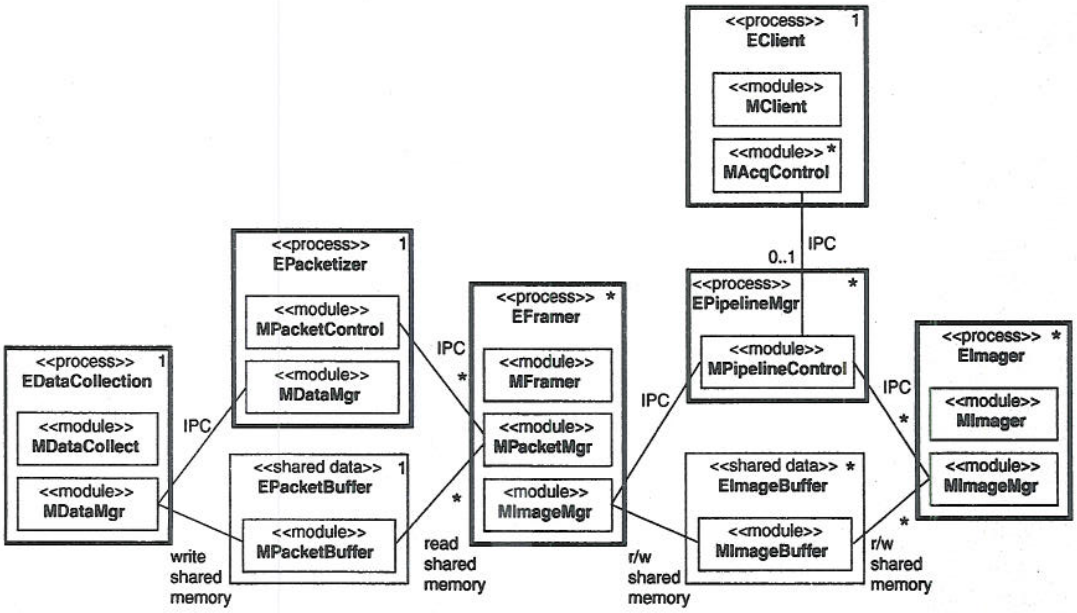

Figure 6.13. Final execution configuration for the imaging subsystem of IS2000. IPC = interprocess communication; r/w = read/write.

of protocol information. These diagrams are limited in that they can only describe a particular sequence rather than a more general pattern of repeated interactions; but often, an example of an interaction is very useful in communicating the protocol, as in this case for the interaction between EPacketizer and the other processes (Figure 6.14).

Now we have finished the central design tasks for the imaging subsystem. Only resource allocation remains.

6.2.3 Final Design Task: Resource Allocation

For the resource allocation task we need to use the global analysis results to allocate resources to the execution configuration. First we must allocate a slice of the CPU to each process by giving it a time budget and priority. Then we need to make decisions about how to allocate other limited resources (for example, address space, memory pool, timers, proxies, ports) to each process.

There are two global analysis strategies that are relevant to allocating processing time to each process. Initially we planned to use one CPU, with all processes assigned to this processor. After the configuration was complete, we applied the strategy *Use RMA to*

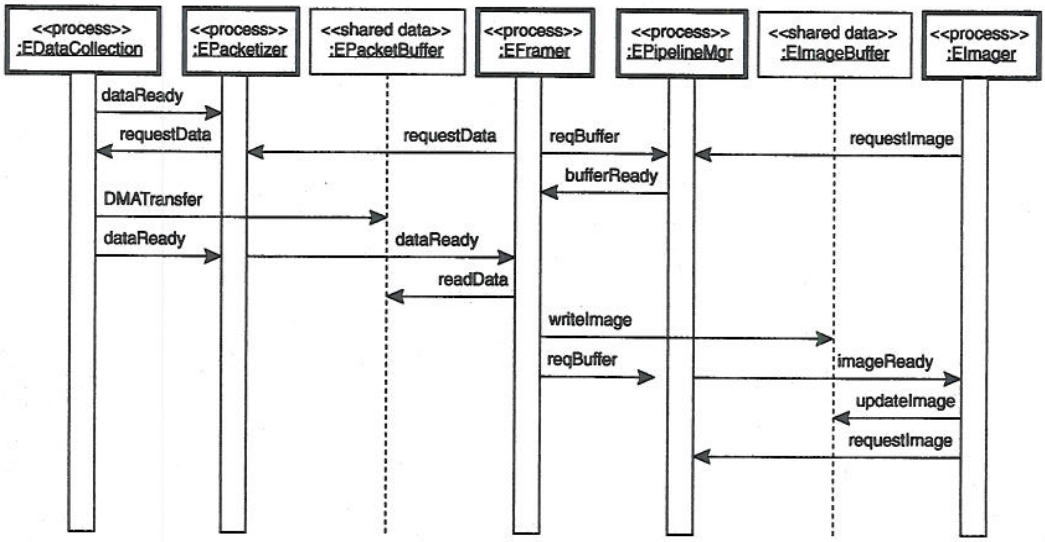

Figure 6.14. An example of EPacketizer's and EPipelineMgr's interactions. DMA = direct memory access.

predict performance, and discovered that the existing architecture design did not meet the real-time performance requirements. We had anticipated this possibility with the strategy *Use an additional CPU*, which we now apply.

Feedback to the Central Design Tasks

With the introduction of an additional processor, we have to redefine the hardware topology and decide how to assign processes to processors. We do this by returning to the diagram in Figure 6.5, which shows the initial sets of processes in the execution view.

The original CPU will handle the real-time processing, and the second CPU, which does not have a real-time operating system, will handle the applications and GUI. We can refine this partitioning by reexamining the processes and explicitly mapping them to processors. Figure 6.15 shows the first step: partitioning the initial sets of processes across the two CPUs.

We also need to introduce a hardware link, define the communication paths between the CPUs, and select the appropriate communication mechanisms. IPC can be encapsulated in the high-level MCommLib module we introduced earlier to handle the details of the IPC protocol.

Data that was passed from the image pipeline to the applications now has to be transferred between processors. One way to accomplish this is to introduce a data transfer ser-

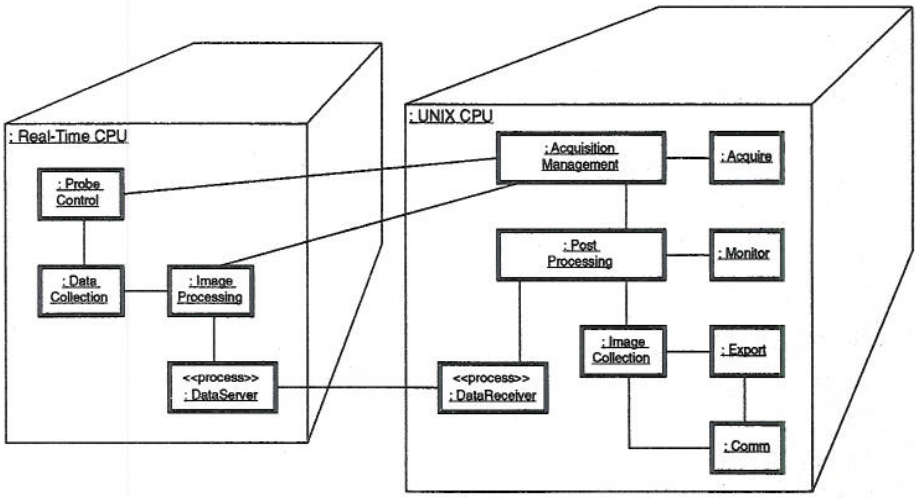

Figure 6.15. Overview of execution architecture view with two CPUs

vice. Such a service reads image buffers coming from the pipeline and sends them to the client running on a different processor.

Adding a data transfer service means introducing new modules to send and receive data between CPUs. We can put each of these in their own process or perhaps combine the data receiver with the client.

Now let's return to the execution configuration diagram and update it to show how it is mapped to the two processors (Figure 6.16). The EClient process is part of acquisition management, so it is mapped to the UNIX CPU. All the other processes in the imaging subsystem stay on the real-time CPU. The communication path between EClient and EPipelineMgr must be reexamined: Let's now use RPC because the path is between processors.

An important difference between these last two figures is that Figure 6.15 shows runtime instances on a particular machine and Figure 6.16 shows runtime entities on a type of machine. We used runtime entities for the second figure so we could describe in a single diagram how all image pipelines are structured. This is similar to the conceptual view, when we didn't want to draw a separate instance diagram for each possible image pipeline. The runtime instance information is better represented in a table than in diagrams.

You may have noticed that the multiplicities on this communication path are not the same as in Figure 6.13. This has nothing to do with the split across two CPUs. The multiplicity is different because in Figure 6.13 we show the communication path between the modules mapped to these processes, and in Figure 6.16 we show it between the processes themselves. There are multiple MAcqControl modules in EClient, and each of these com-

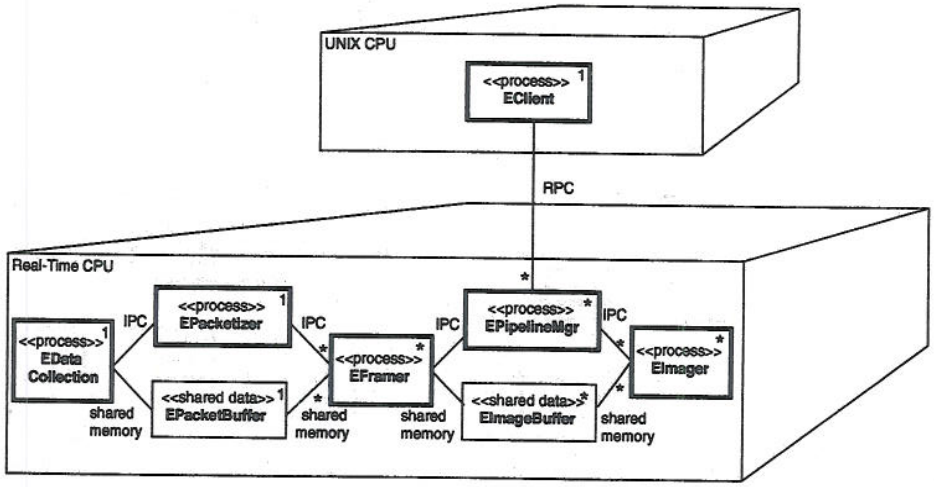

Figure 6.16. Mapping the execution configuration to processors. RPC = remote procedure call; IPC = interprocess communication.

municates with at most one EPipelineMgr. So EClient communicates with multiple (zero or more) EPipelineMgr processes.

Memory and Other Processing Resources

The total amount of memory we have to work with is 64MB. The packetizer and pipeline manager are the biggest memory users. Both use memory to implement buffers as circular queues. To simplify the algorithms for reading the buffers, they acquire memory as contiguous blocks, preallocated based on worst-case conditions.

For the pipeline manager, the size of the image buffer is dependent on the type of application. The number of buffers is typically one more than the number of pipeline stages. Buffers are preallocated to improve performance. Shared memory is shared only among the stages within a processing pipeline.

For the packetizer, the size of the packet buffer is important because it affects throughput and response time. If it is too large and the data arrival rate is low, then the transfer of buffers will be slow because a buffer has to fill before it is transferred. In this case there may be perceptible delays in the screen update. If the buffers are too small, CPU time is wasted servicing interrupts too frequently. Let's assign an initial size based on experience, and fine-tune the estimate as we analyze performance.

Some of the resources are not adequate for our needs, but we can design a system support service that extends the resource to meet the requirements. For example, the UNIX platform supports one real-time interval timer per executing process. The system time-out

support requires capabilities beyond those services offered by the software platform. We can extend the capabilities by providing a timer service that allows users to create one or more concurrent timers and is easy to use: Users need not have platform-specific knowledge about signal handling and blocking.

6.2.4 Design Summary for IS2000 Execution View

When the design of the execution view is finished, we want to have made the implementation decisions for the dynamic aspects of the system, building in flexibility when we think it is needed. To accomplish this we first identified the runtime elements of the software platform, such as processes and threads. Then we related the conceptual components and modules to these runtime elements, first using a one-to-one mapping to get an approximation of the process boundaries, and then refining the process boundaries as the design progressed. Table 6.2 summarizes the design decisions discussed in this chapter.

Design Decision	Rationale
Global analysis	
Add a new issue: High Throughput.	The probe hardware has a very high data rate. Strategy: *Avoid use of multiple threads.*
Add strategies to the issue Real-Time Acquisition Performance: *Use flexible allocation of modules to processes. Develop guidelines for module behavior. Use rate monotonic analysis to predict performance.*	Strategy: *Map independent threads of control to processes.* Strategy: *Use an additional CPU.*
Add a new issue: Resource Limitations.	Strategy: *Map independent threads of control to processes.*
Begin defining runtime entities	
Each pipeline stage (e.g., EFramer, EImager) is a process. EClient is a process. EDataCollection is a process.	If there's a one-to-one correspondence between a conceptual component and a module, put the module in its own thread of control. Strategy: *Map independent threads of control to processes.*

Table 6.2. Sequence of Design Decisions for IS2000 Execution View

Design Decision	Rationale
Link MAcqControl to the client process.	Simplify the client's responsibility.
Pipeline manager	
Add new strategy to the issue Real-Time Acquisition Performance: *Use shared memory to communicate between pipeline stages.*	There is a high volume of data and real-time performance requirements.
Split MPipelineMgr into two modules—one for processing (MPipeline-Control) and one for data (MImageBuffer). Revise module view.	Shared memory is supported directly by the operating system.
Centralize pipeline control in a single process: EPipelineMgr. There is one EPipelineMgr per pipeline.	Keep the control algorithm simple.
Communication paths for the pipeline manager	
EPipelineMgr communicates with the pipeline stages via interprocess communication (IPC).	Use available technology.
Add a new technological factor, T3.4: IPC mechanism. Add a new strategy to the issue Resource Limitations: *Use dynamic IPC connections.*	Use IPC.
Introduce a new module (MCommLib) to provide higher level support for IPC.	Strategy: *Encapsulate multiprocess support facilities.*
Assigning multiplicities to the execution configuration	
Runtime entities for an image pipeline (EFramer, EPipelineMgr, EImage-Buffer, EImager) are created dynamically when the acquisition procedure is requested.	Strategy: *Limit the number of active processes.*

Table 6.2. Sequence of Design Decisions for IS2000 Execution View *(continued)*

Design Decision	Rationale
Packetizer	
The packetizer (EPacketizer) communicates with the pipeline (EFramer) via IPC, and transfers data via shared memory (EPacketBuffer).	Strategy: *Use shared memory to communicate between pipeline stages.*
One process for the packetizer (EPacketizer). One packetizer in the system.	Keep the control algorithm simple.
Resource allocation	
Introduce an additional CPU because performance is inadequate.	Strategy: *Use rate monotonic analysis to predict performance.* Strategy: *Use an additional CPU.*
Determine optimal image buffer sizes.	This depends on the number of pipeline stages in the acquisition procedure.
Determine optimal packet buffer size.	This depends on the data rate of probe hardware.
Preallocate a buffer from contiguous memory.	Keep the algorithm for accessing buffers simple.

Table 6.2. Sequence of Design Decisions for IS2000 Execution View *(continued)*

6.3 Summary of Execution Architecture View

The execution view describes the mapping of functionality to physical resources, and the runtime characteristics of the system. Table 6.3 summarizes the elements, relations, and artifacts to be used in this view. As in the other views, the elements and relations are the building blocks for the architecture view, and the artifacts are used to document or to describe the architecture.

The execution configuration describes how the runtime entities are instantiated as runtime instances, and describes the communication between runtime instances over the life of the system. A system usually has a different configuration at start-up and shutdown than it does during its normal operation. Sometimes a system has a configuration that changes throughout its lifetime.

Element	UML Element	New Stereotype	Notation	Attributes	Associated Behavior
Runtime entity	Process	—	<<...>>	Host type, replication, resource allocation	—
	Thread	—			
	Class or active class	<<shared data>>, <<task>>, etc.			
Communication path	Association	—	———	—	Communication protocol

Relation	UML Element	Notation	Description
use mechanism	Association name	Name of communication mechanism; for example, IPC, RPC	A communication path uses a communication mechanism.
communicate over	—	▭— (connection of class and association)	A runtime entity (or the module assigned to it) communicates over a communication path.
assigned to	Composition	Nesting (or ◆)	A module is assigned to zero or more runtime entities.

Artifact	Representation
Execution configuration	UML Class Diagram
Execution configuration mapped to hardware devices	UML Deployment Diagram
Dynamic behavior of configuration, or transition between configurations	UML Sequence Diagram
Description of runtime entities (including host type, replication, and assigned modules)	Table or UML Class Diagram

Table 6.3. Summary of Execution Architecture View

Description of runtime instances (including resource allocation)	Table
Communication protocol	Natural language description, or UML Sequence Diagram or Statechart Diagram
IPC = interprocess communication; RPC = remote procedure call; UML = Unified Modeling Language.	

Table 6.3. Summary of Execution Architecture View *(continued)*

The meta-model in Figure 6.17 shows the basic elements used in the execution view, and how they are related to the physical resources and to the modules from the module view. The basic elements are runtime entities and their communication paths. A module is assigned to zero or more runtime entities, and a runtime entity can have multiple modules assigned to it. Each runtime entity is mapped to exactly one platform element.

The runtime entities communicate with each other over communication paths. A communication path uses a particular communication mechanism, which in turn may consume additional platform elements.

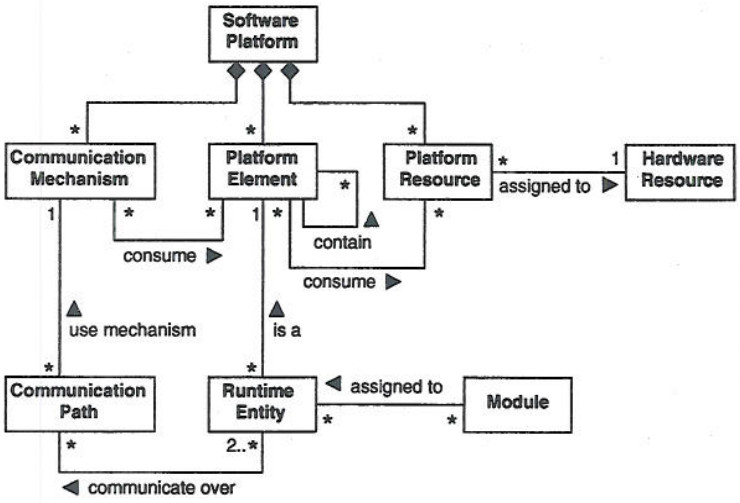

Figure 6.17. Meta-model of the execution architecture view

6.3.1 Traceability

Describing the relationships of the execution view to requirements, external factors, and the other architecture views provides traceability. The following three items should be traceable in the execution view:

1. *Critical requirements and organizational and technological factors.* Traceability between design decisions for the execution view and organizational, technological, and product factors helps you determine whether your design meets the requirements. It also helps you determine the impact of changes on requirements or factors.
2. *Elements in the module view.* To implement modules correctly, developers must know how they are mapped to elements of the execution view.
3. *Elements in the conceptual view.* Traceability between conceptual components and execution view elements can be done directly, or through the modules. In the case study of Chapter 8, traceability is used to determine the correctness of the automatically generated code. More commonly, it helps you determine whether the implementation is correct.

Looking ahead to the code architecture view, there is a correspondence between elements (for example, runtime components) in the code architecture view and the runtime elements in the execution view for the purpose of system building.

6.3.2 Uses for the Execution Architecture View

The execution view is used by

- Architects, to design the runtime aspects of the system so that it meets the requirements and can adapt to expected changes
- Developers, to provide a correct implementation
- Testers, who need to know the runtime aspects of the system to plan the testing (particularly unit testing)
- Maintainers, to determine how a change in the runtime platform affects the system or how changes in requirements affect the system's runtime aspects

Additional Reading

When systems become distributed, developers need to consider dynamic structure and communication, coordination, and synchronization. A number of interconnection languages have been introduced that address the issue of allocating components in a distributed environment. These include those cited by Barbacci, Weinstock, and Wing (1988); Magee, Dulay, and Kramer (1994); Purtilo (1994); and Royce (1990). These issues are addressed in the execution architecture view.

Hatley and Pirbhai (1988) and Schmidt and Suda (1993) emphasize the need for separate module and execution views and providing flexible assignment of modules to runtime elements.

Shaw et al. (1995) introduce a connector for real-time scheduling in their language for universal connector support (UniCon). Given the priority and period information, it is sufficient to schedule the processes. If the scheduling policy is set to rate monotonic, then UniCon can package the trace, period, execution time, and priority information and transmit it to the analysis tool for scheduling analysis.

Chatterjee et al. (1997) present a systems engineering tool kit for the design and analysis of real-time systems. This toolset was applied to the Healthy Vision case study, in which information from the execution architecture was used as input to the toolset to evaluate performance properties of the system.

Nord and Cheng (1994) use RMA (Klein et al., 1993) to evaluate the performance properties of a real-time system based on the execution architecture, and they demonstrate how this provides feedback to the architecture design decisions.

Kazman et al. (1999) have extended the work on SAAM to evaluate other quality attributes of a system such as performance and availability based on structures from the execution architecture view.