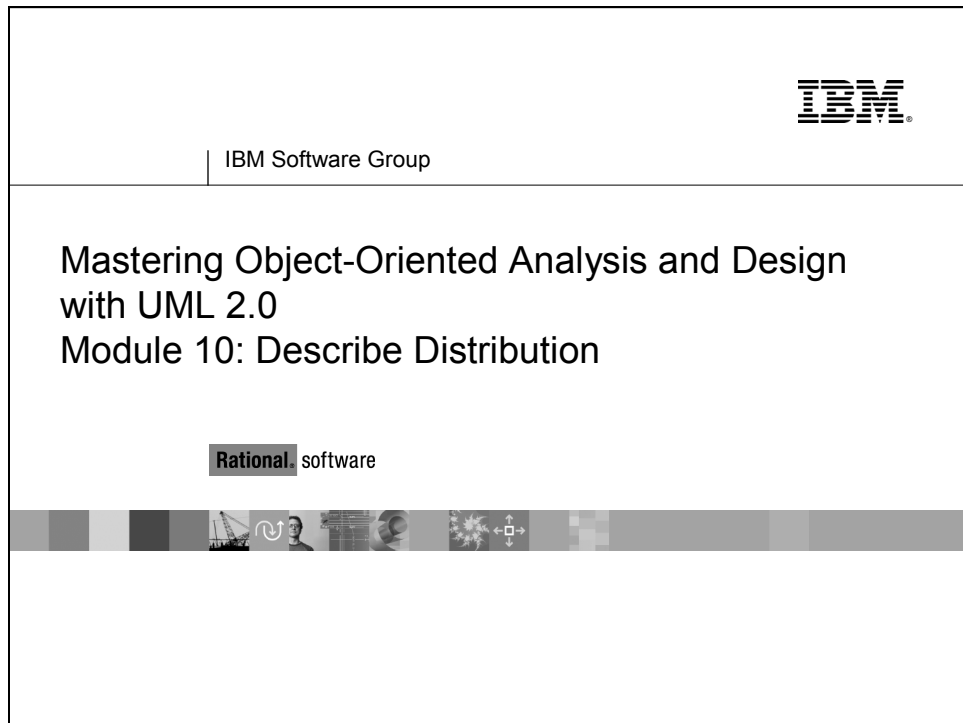


► ► ► Module 10 Describe Distribution



Topics

Describe Distribution Overview	10-4
The Network Configuration	10-15
Process-to-Node Allocation Considerations	10-20
What is Deployment?	10-22
Distribution Mechanism	10-29
Review	10-38

Objectives: Describe Distribution

Objectives: Describe Distribution

- ♦ Explain the purpose of the Describe Distribution activity and when in the lifecycle it is performed
- ♦ Describe how the functionality of the system can be distributed across physical nodes
- ♦ Model the distribution decisions of the system in the Deployment Model
- ♦ Articulate the rationale and considerations that support the architectural decisions

2



The **Describe Distribution** activity is where the processes defined in the Describe the Run-time Architecture activity are allocated to actual physical nodes. Just identifying the individual processes is not enough; the relationship between the processes and the hardware must be described.

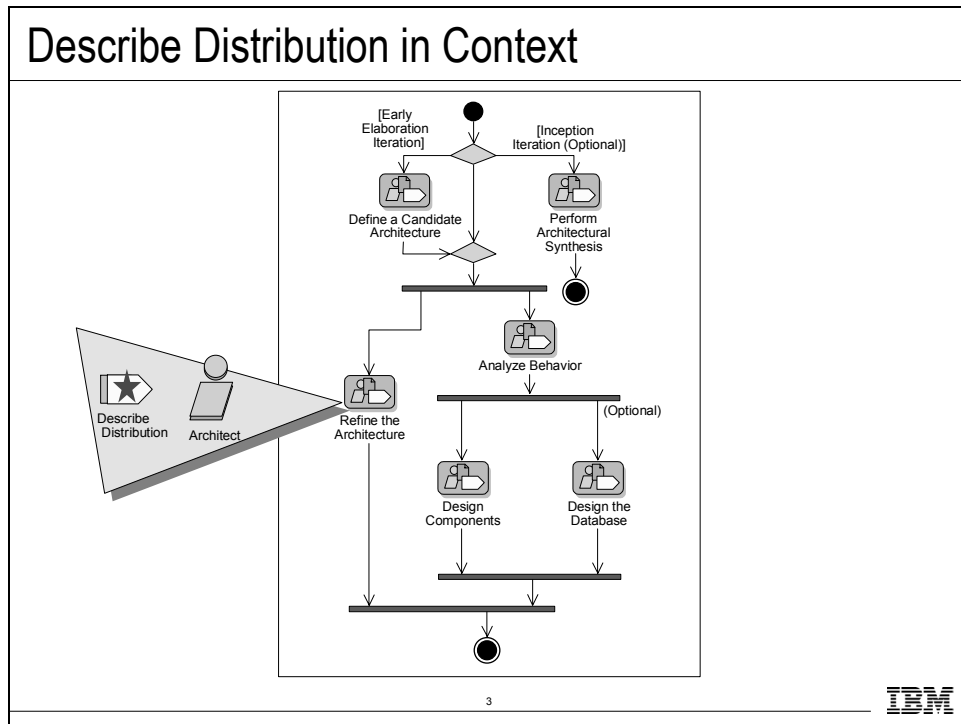
The focus of the **Describe Distribution** activity is on developing the Deployment View of the architecture. This activity is really only required for distributed systems.

In this module, we will describe *what* is performed in **Describe Distribution**, but will not describe *how* to do it. Such a discussion is the purpose of interest in an architecture course, which this course is not.

The goal of this module is to give the student an understanding of how to model the Deployment Model using the UML.

An understanding of the rationale and considerations that support the architectural decisions is needed in order to understand the architecture, which is the framework in which designs must be developed.

Describe Distribution in Context



As you might recall, the above diagram illustrates the workflow that we are using in this course. It is a tailored version of the Analysis and Design core workflow of the Rational Unified Process.

In the Architectural Analysis module, distribution was identified as an analysis mechanism. This analysis mechanism was then used to indicate what classes need to be distributed.

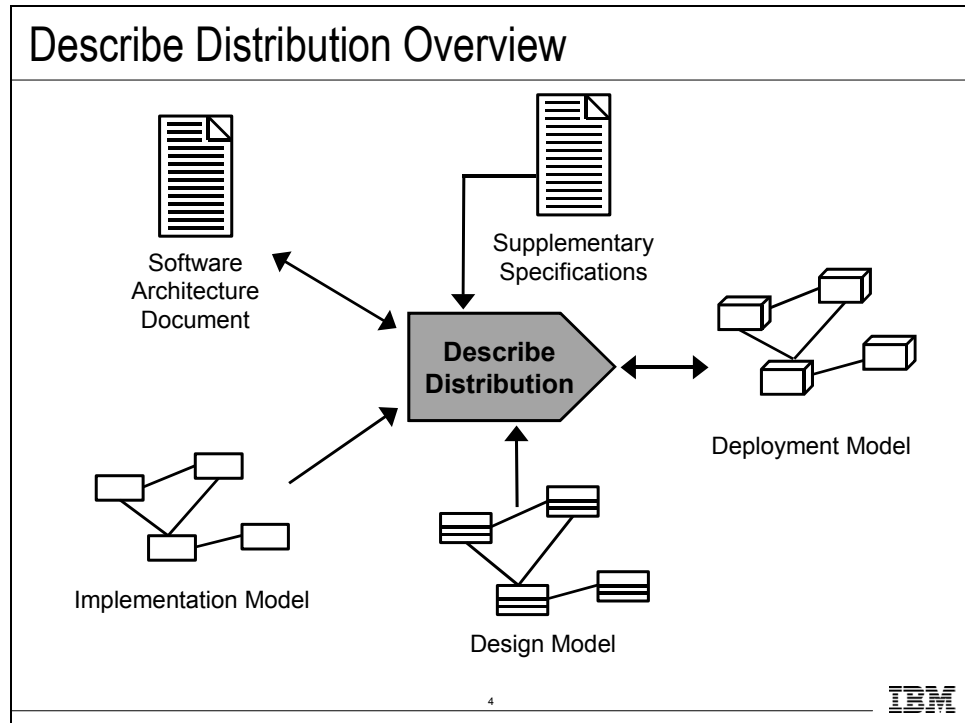
In the Identify Design Elements module, the subsystems, their interfaces, and their dependencies were defined. The initial design classes and the packages in which they belong were also defined. In addition to the definition of the design elements, the technologies and mechanisms to support distribution were selected in the Identify Design Mechanisms module.

In the Describe the Run-time Architecture module, the independent threads of control were identified and the design elements were mapped to these threads of control.

In **Describe Distribution**, the physical architecture will be modeled using nodes and connections. The independent threads of control identified in Describe the Run-time Architecture are mapped to the nodes. **Describe Distribution** is a separate activity, which is at the same level as Identify Design Elements and Identify Design Mechanisms.

If the system under development will only run on one node, then there is no need for a separate Deployment Model. In such a case, the **Describe Distribution** activity can be skipped.

Describe Distribution Overview



The **Describe Distribution** activity is performed by the Architect.

Purpose

The purpose of the **Describe Distribution** activity is to describe how the functionality of the system is distributed across physical nodes. This is required only for distributed systems.

Input Artifacts

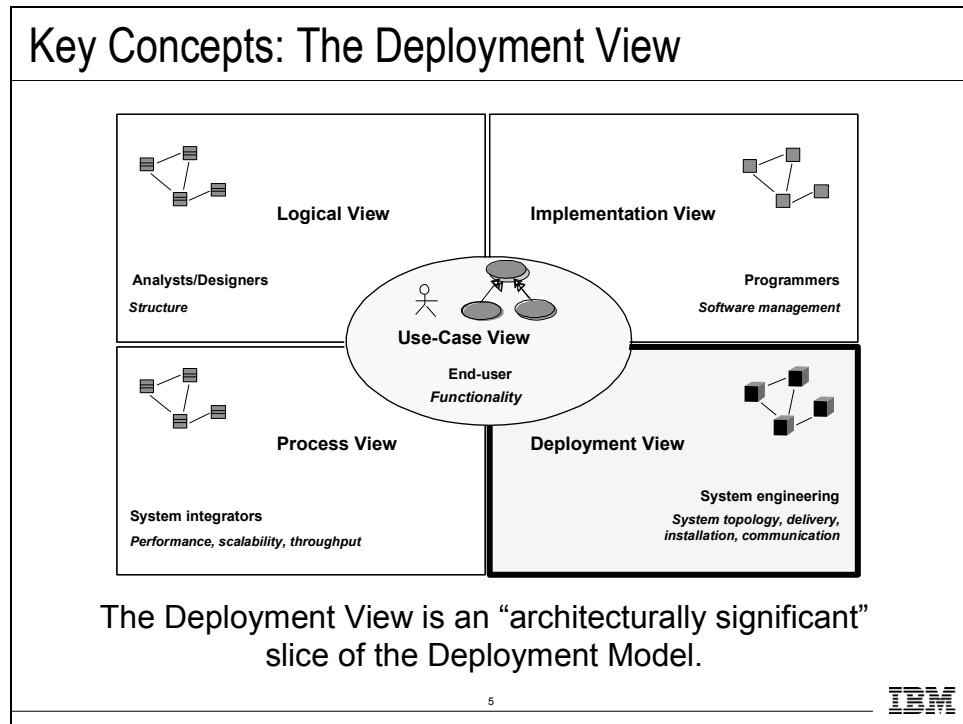
- Software Architecture Document
- Deployment Model
- Implementation Model
- Design Model
- Supplementary Specifications

Resulting Artifacts

- Software Architecture Document
- Deployment Model

Note: The Implementation Model is a collection of components and the implementation subsystems that contain them. Components include both deliverable components, such as executables, and components from which the deliverables are produced, such as source code files. The Implementation Model is developed in the Implementation discipline which is out of the scope of this Analysis and Design course.

Key Concepts: The Deployment View



In **Describe Distribution**, we will concentrate on the Deployment View. Thus, before we can discuss the details of what occurs in **Describe Distribution**, you need to review what the Deployment View is.

The above diagram describes the model Rational uses to describe the software architecture. For each view, the stakeholder interested in the view, and the concern addressed in the view, are both listed.

The Deployment View contains the description of the various physical nodes and their interconnections for the most typical platform configurations, along with the allocation of the threads of control (from the Process View) to the physical nodes.

The Deployment View is a subset of the Deployment Model, and is necessary only if the system is distributed.

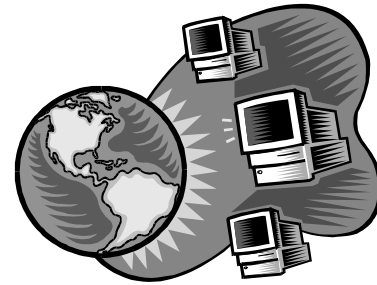
Note: All processors are considered architecturally significant, so the Deployment View of the architecture contains all processors.

The Deployment View is represented graphically on Deployment diagrams. These diagrams will be discussed in detail later in this module.

Why Distribute?

Why Distribute?

- ♦ Reduce processor load
- ♦ Special processing requirements
- ♦ Scaling concerns
- ♦ Economic concerns
- ♦ Distributed access to the system



6

IBM

Before we discuss some common distribution patterns, you need to look at some of the reasons why you would distribute an application in the first place. There are many cases where the system workload cannot be handled by one processor. This can be due to special processing requirements, as in the case of digital signal processing, which may require specialized and dedicated processors. The need for distribution may also result from inherent scaling concerns, where the large number of concurrent users are simply too many to support on any single processor. Alternatively, it may result from economic concerns, where the price performance of smaller, cheaper processors cannot be matched in larger models. For some systems, the ability to access the system from distributed locations might mean that the system itself must be distributed.

The distribution of processes across two or more nodes requires a closer examination of the patterns of inter-process communication in the system. Often, there is a naïve perception that distribution of processing can “off-load” work from one machine onto a second. In practice, the additional inter-process communication workload can easily negate any gains made from workload distribution if the process and node boundaries are not considered carefully. Achieving real benefits from distribution requires work and careful planning.

Distribution Patterns

Distribution Patterns

- ◆ Client/Server
 - 3-tier
 - Fat Client
 - Fat Server
 - Distributed Client/Server
- ◆ Peer-to-peer



7

IBM

There are a number of typical patterns of distribution in systems, depending on the functionality of the system and the type of application. Typically, the distribution pattern is described as “the architecture” of the system, although the full architecture encompasses this, but also many more things.

In client/server architectures, there are specialized network processor nodes called clients and nodes called servers. Clients are consumers of services provided by servers.

The distribution of the system functionality to the clients and servers distinguishes the different types or styles of client/server architectures.

3-tier : Functionality is equally divided into three logical partitions: application, business, and data services.

Fat client: More functionality is placed on the client. Examples: database and file servers.

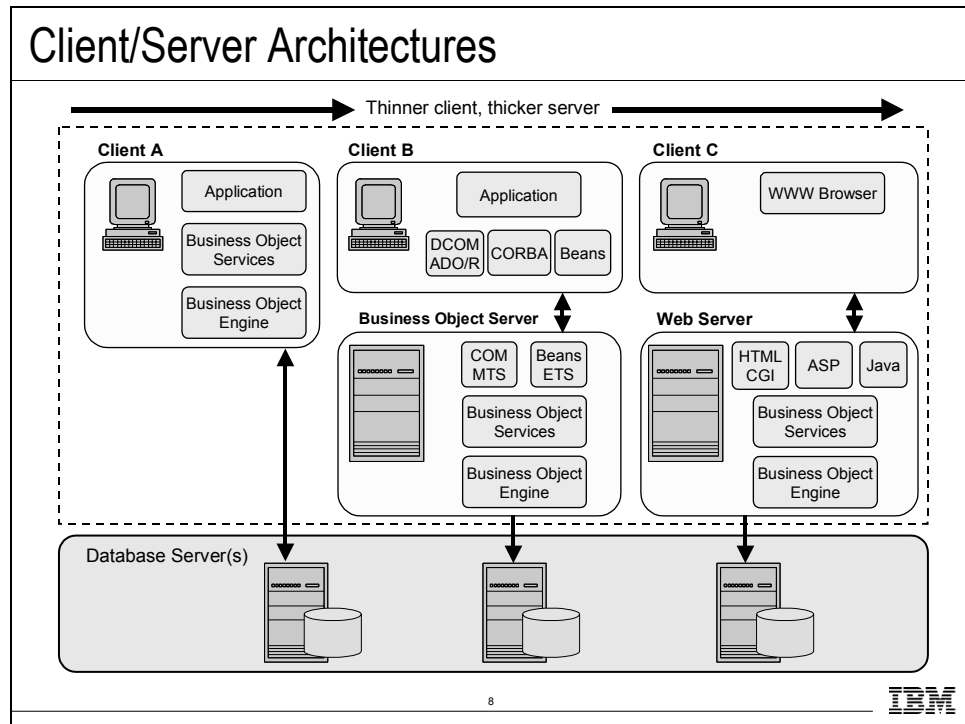
Fat server: More functionality is placed on the server. Examples: Groupware, transaction, and Web servers.

Distributed client/server: The application and business and data services reside on different nodes, potentially with specialization of servers in the business services and data services tiers. A full realization of a 3-tier architecture.

In a **Peer-to-Peer** architecture, any process or node in the system can be both client and server.

Each of these architectural patterns is described briefly in this module.

Client/Server Architectures



Client/server is a conceptual way of breaking up the application into service requestors (clients) and service providers (servers).

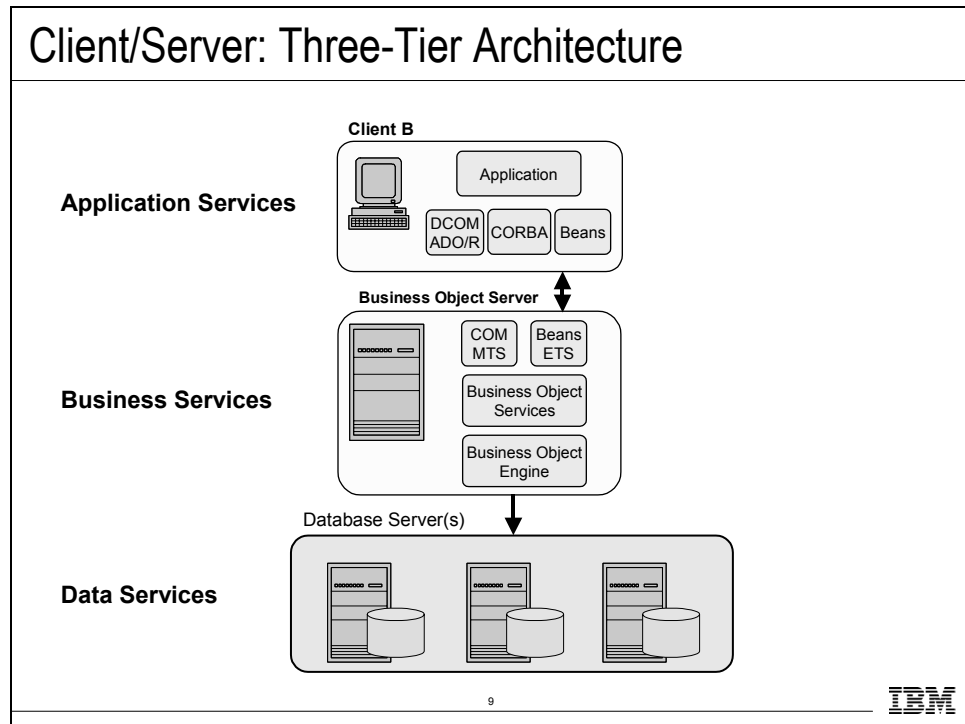
A client often services a single user and often handles end-user presentation services (GUIs). A system can consist of several different types of clients, examples of which include user workstations and network computers.

The server usually provides services to several clients simultaneously. These services are typically database, security, or print services. A system can consist of several different types of servers. For example: *database servers*, handling database machines such as Sybase, Ingres, Oracle, Informix; *print servers*, handling the driver logic, such as queuing for a specific printer; *communication servers* (TCP/IP, ISDN, X.25); *window manager servers* (X); and *file servers* (NFS under UNIX).

The application and business logic is distributed among both the client and the server (application partitioning).

In the above example, Client A is an example of a two-tier architecture, with most application logic located in the server. Client B is a typical three-tier architecture, with business services implemented in a Business Object Server. Client C is a typical Web-based application. You will look at each of these distribution strategies in more detail.

Client/Server: Three-Tier Architecture



Three-tier architecture is a special case of the client/server architecture in which functionality in the system is divided into three logical partitions: application services, business services, and data services. The “logical partitions” may in fact map to three or more physical nodes.

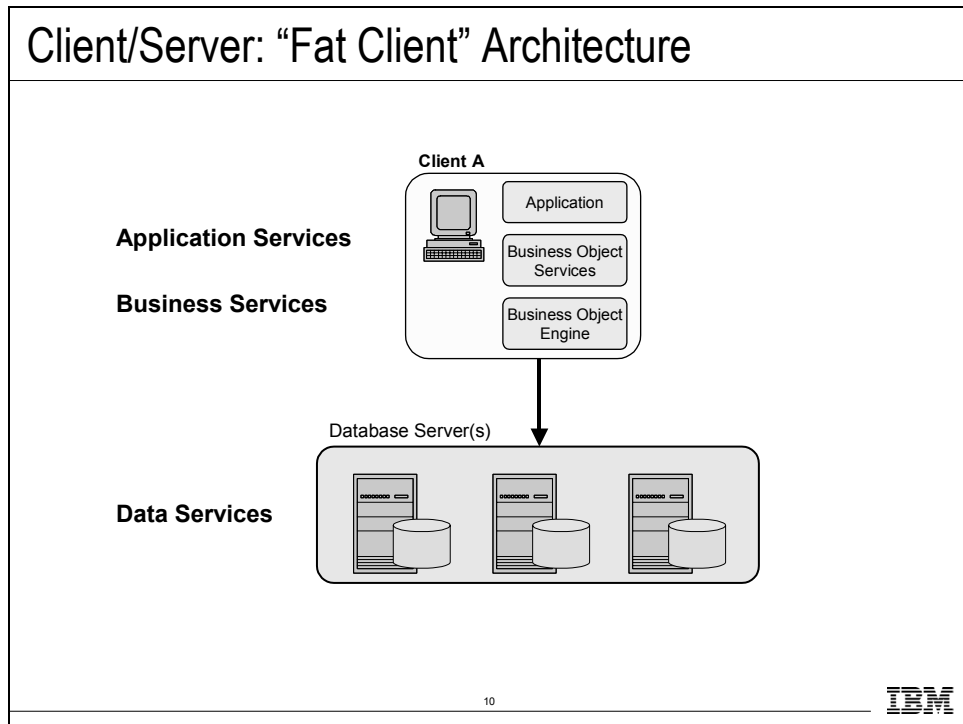
Application services, primarily dealing with GUI presentation issues, tend to execute on a dedicated desktop workstation with a graphical, windowing operating environment.

Data services tend to be implemented using database server technology, which normally executes on one or more high-performance, high-bandwidth nodes that serve hundreds or thousands of users, connected over a network.

Business services are typically used by many users in common, so they tend to be located on specialized servers as well, although they may reside on the same nodes as the data services.

Partitioning functionality along these lines provides a relatively reliable pattern for scalability: by adding servers and rebalancing processing across data and business servers, a greater degree of scalability is achieved.

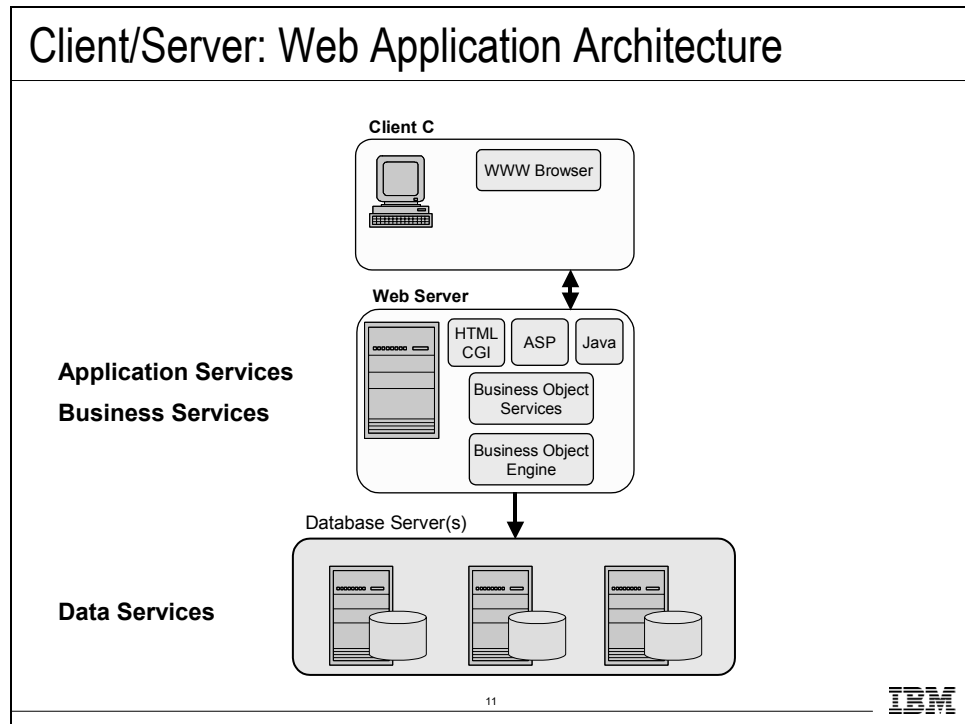
Client/Server: “Fat Client” Architecture



The fat client distribution pattern is a special case of the client/server architecture in which much of the functionality in the system runs on the client.

The client is “fat” since nearly everything runs on it (except in a variation, called the “two-tier architecture,” in which the data services are located on a separate node). Application services, business services and data services all reside on client machine. The database server is usually on another machine.

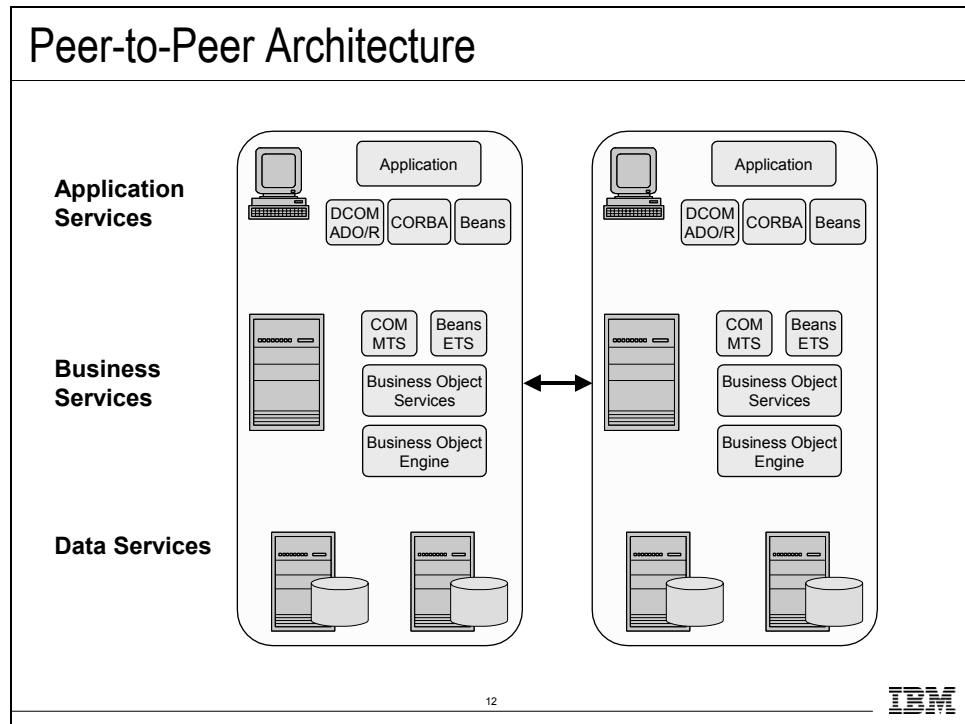
Client/Server: Web Application Architecture



At the other end of the spectrum from the fat client is the typical Web Application (which might be characterized as fat server or “anorexic client”). Since the client is simply a Web browser running a set of HTML pages and Java applets, Java Beans, or ActiveX components, there is very little application there at all. Nearly all work takes place on one or more Web servers and data servers.

Web applications are easy to distribute and easy to change. They are relatively inexpensive to develop and support (since much of the application infrastructure is provided by the browser and the web server). However, they might not provide the desired degree of control over the application, and they tend to saturate the network quickly if not well-designed (and sometimes despite being well-designed).

Peer-to-Peer Architecture



In the peer-to-peer architecture, any process or node in the system can be both client and server. Distribution of functionality is achieved by grouping inter-related services together to minimize network traffic while maximizing throughput and system utilization. Such systems tend to be complex, and there is a greater need to be aware of issues such as deadlock, starvation between processes, and fault handling.

Describe Distribution Steps

Describe Distribution Steps

- ◆ Define the network configuration
- ◆ Allocate processes to nodes
- ◆ Define the distribution mechanism

13

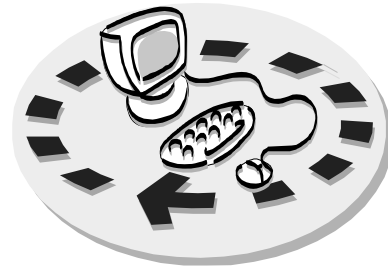


The above are the topics you will discuss within the **Describe Distribution** module. Unlike the Designer activity modules, this module does not discuss each step of the activity, since the objective is to understand the important distribution concepts, not to learn *how* to design the distributed aspects of the architecture.

Describe Distribution Steps

Describe Distribution Steps

- ☆ ♦ Define the network configuration
 - ♦ Allocate processes to nodes
 - ♦ Define the distribution mechanism

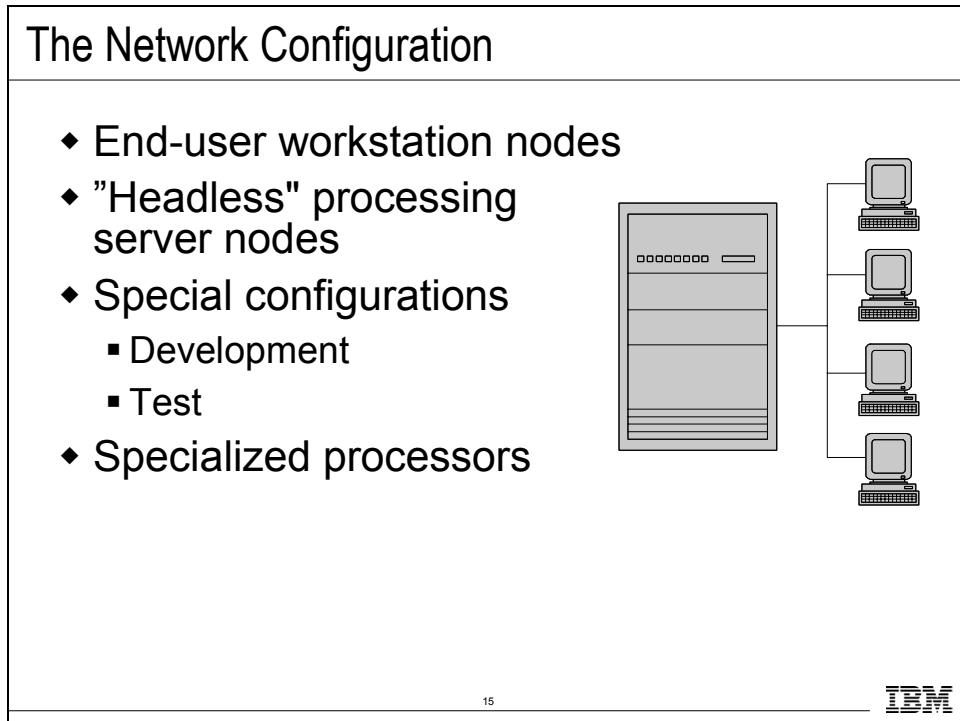


14

IBM

The configuration and topology of the network is modeled in order to make it understandable and check for any inconsistencies.

The Network Configuration



The topology of the network and the capabilities and characteristics of the processors and devices on the network determine the nature and degree of distribution possible in the system.

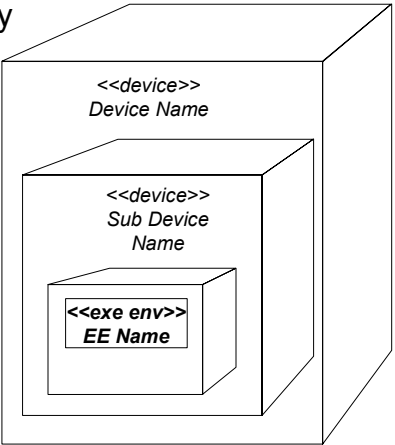
The following information needs to be captured:


- The physical layout of the network, including locations.
- The nodes on the network, their configurations and capabilities. The configuration includes both the hardware and the software installed on the nodes, the number of processors, the amount of disk space, the amount of memory, the amount of swap, and so forth. Hardware installed on the node can be represented using "devices."
- The bandwidth of each segment on the network.
- The existence of any redundant pathways on the network. (This will aid in providing fault tolerance capabilities.)
- The primary purpose of the node. This includes:
 - Workstation nodes used by end users
 - Server nodes on which "headless" processing occurs
 - Special configurations used for development and test.
 - Other specialized processors
- IP design and facilities (for example, DNS and VPN), if IP network exists.
- The role of the Internet in the solution.

Review: What Is a Node?

Review: What Is a Node?

- ♦ Represents a run-time computational resource
 - Generally has at least memory and often processing capability.
- ♦ Types:
 - Device
 - Physical computational resource with processing capability.
 - May be nested
 - Execution Environment
 - Represent particular execution platforms



16


The essential elements of a deployment diagram are nodes and their connectors. A *node* is a run-time physical object that represents a computational resource, generally having at least memory and often processing capability as well.

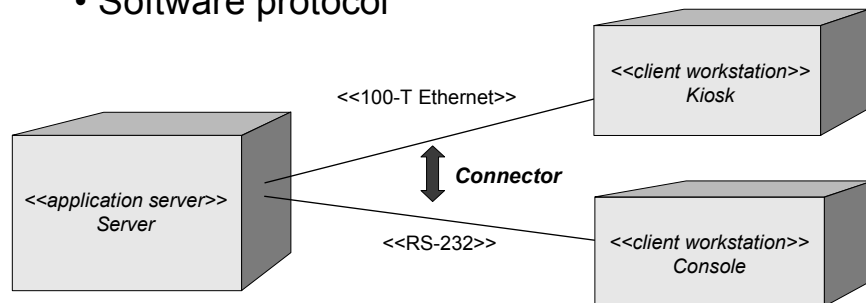
Execution Environment and *device* are types of *node* but the UML 2 distinction between them is rather vague.

- **Devices** may have artifacts deployed for execution software that controls the functionality of the device itself (physical computational resource with processing capability). Typical examples include <<print server>>, <<application server>>, <<client workstation>>, <<mobile device>>, <<embedded device>>, <<processor>>, etc. Devices may be complex and contain other devices.
- **Execution Environments** represent particular execution platforms, such as an operating system (<<Win2K>>, <<VxWorks>>, etc.), a workstation engine (<<workstation engine>>), a database management system (<<DB2>>), etc.

Review: What Is a Connector?

Review: What Is a Connector?

- ♦ A connector represents a:
 - Communication mechanism
 - Physical medium
 - Software protocol

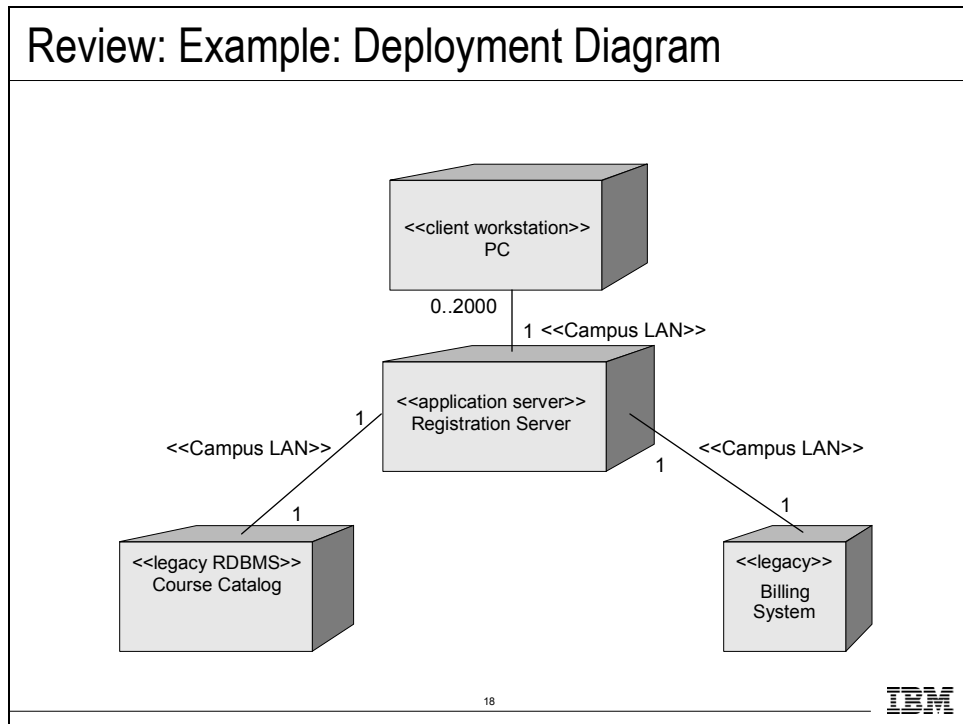


17

IBM

Connectors can be drawn between nodes. These connectors represent communication mechanisms and can be described by physical mediums (for example, Ethernet, fiber optic cable) or software protocol (for example, TCP/IP, RS-232). A stereotype may be used to specify the type of connector.

Review: Example: Deployment Diagram



Deployment diagrams allow you to capture the topology of the system nodes, including the assignment of run-time elements to nodes. This allows you to visually see potential bottlenecks.

A Deployment diagram contains nodes connected by associations. The associations indicate a communication path between the nodes.

The above diagram illustrates the Deployment View for the Course Registration System.

There are PC clients on the campus network.

The main business processing hardware is the Registration Server. It talks to the two machines that host the legacy systems. All nodes are on the campus network.

Describe Distribution Steps

Describe Distribution Steps

- ♦ Define the network configuration
- ☆ ♦ Allocate processes to nodes
- ♦ Define the distribution mechanism

Process-to-Node Allocation Considerations

Process-to-Node Allocation Considerations

- ◆ Distribution patterns
- ◆ Response time and system throughput
- ◆ Minimization of cross-network traffic
- ◆ Node capacity
- ◆ Communication medium bandwidth
- ◆ Availability of hardware and communication links
- ◆ Rerouting requirements



20



Processes must be assigned to a hardware device for execution in order to distribute the workload of the system.

If the chosen architecture implies or requires a specific distribution pattern, this should be realized. Distribution patterns were discussed earlier in this module.

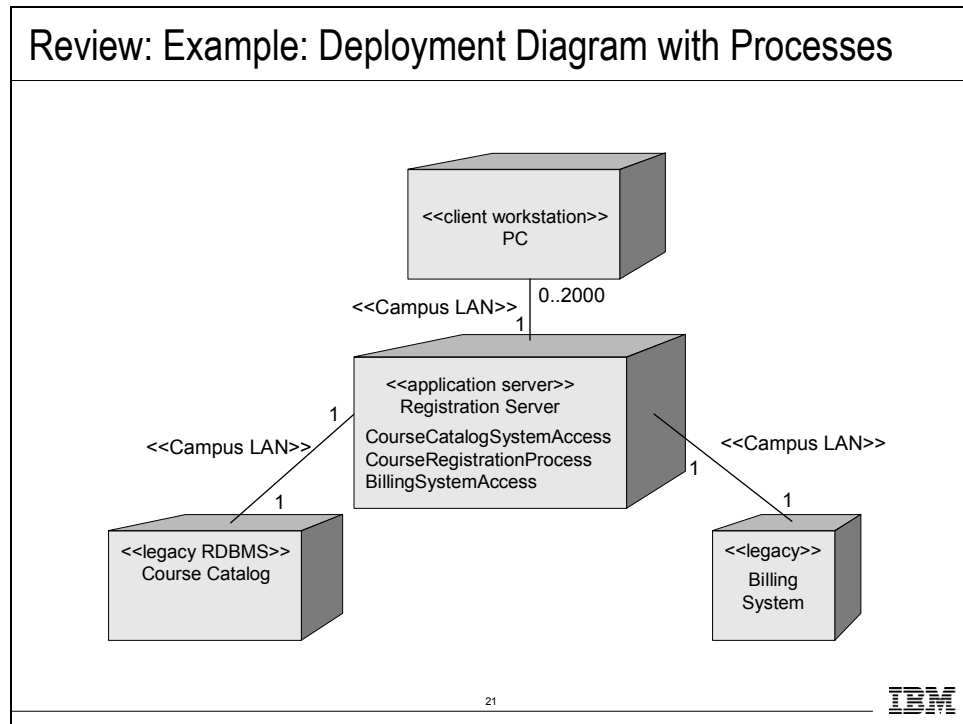
Those processes with fast response time requirements should be assigned to the fastest processors.

Processes should be allocated to nodes so as to minimize the amount of cross-network traffic. Network traffic, in most cases, is quite expensive. It is an order of magnitude or two slower than inter-process communication. Processes that interact to a great degree should be co-located on the same node. Processes that interact less frequently can reside on different nodes. The crucial decision, and one that sometimes requires iteration, is where to draw the line.

Additional considerations:

- Node capacity (in terms of memory and processing power)
- Communication medium bandwidth (bus, LANs, WANs)
- Availability of hardware and communication links
- Rerouting requirements for redundancy and fault-tolerance

Review: Example: Deployment Diagram with Processes



Deployment diagrams allow you to capture the topology of the system nodes, including the assignment of run-time elements to them. This allows you to visually see potential bottlenecks.

As discussed earlier, a Deployment diagram contains nodes connected by associations. The associations indicate a communication path between the nodes.

Nodes may contain artifacts which indicates that the artifact lives on or runs on the node. Those entities that have been “compiled away” are not shown. An example of a run-time object is a process.

The above diagram once again illustrates the Deployment View for the Course Registration System. It has been updated to include the processes which execute on the nodes. These processes are the ones that were defined in the Describe the Run-time Architecture module.

Note: No threads are shown in the above diagram, because threads always run in the context of a process.

What is Deployment?

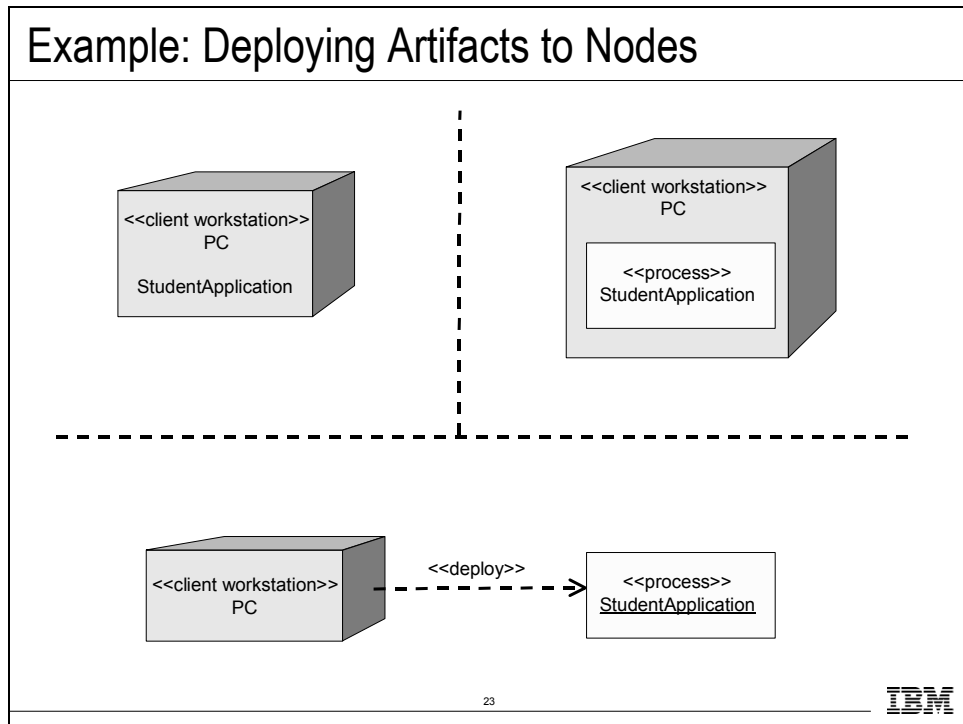
What is Deployment?

- ♦ Deployment is the assignment, or mapping, of software artifacts to physical nodes during execution.
 - Artifacts are the entities that are deployed onto physical nodes
 - Processes are assigned to computers
- ♦ Artifacts model physical entities.
 - Files, executables, database tables, web pages, etc.
- ♦ Nodes model computational resources.
 - Computers, storage units.

22



Example: Deploying Artifacts to Nodes



Here are three examples of valid UML 2 artifacts deployed to nodes.

What is Manifestation?

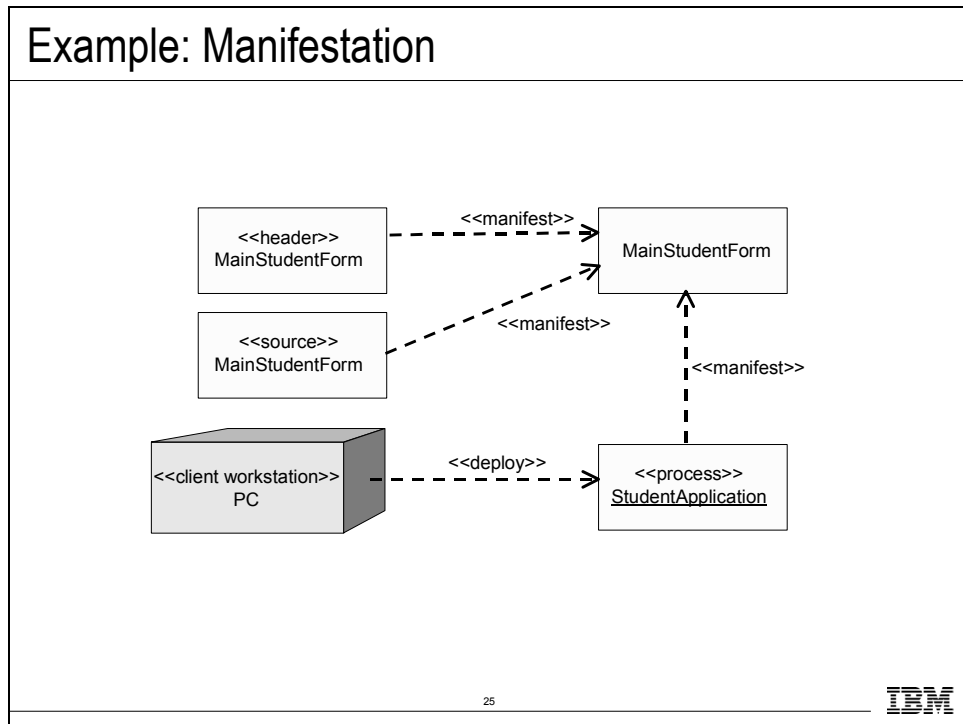
What is Manifestation?

- ♦ The physical implementation of a model element as an artifact.
 - A relationship between the model element and the artifact that implements it
 - Model elements are typically implemented as a set of artifacts.
 - ♦ Source files, executable files, documentation file

24



Example: Manifestation



The `<<header>>` and `<<source>>` artifacts are an implementation manifestation for the `MainStudentForm` model element class.

The `<<client workstation>> PC` deploys the `<<process>> StudentApplication` artifact which is a run-time manifestation of the `MainStudentForm` model element class.

What is a Deployment Specification?

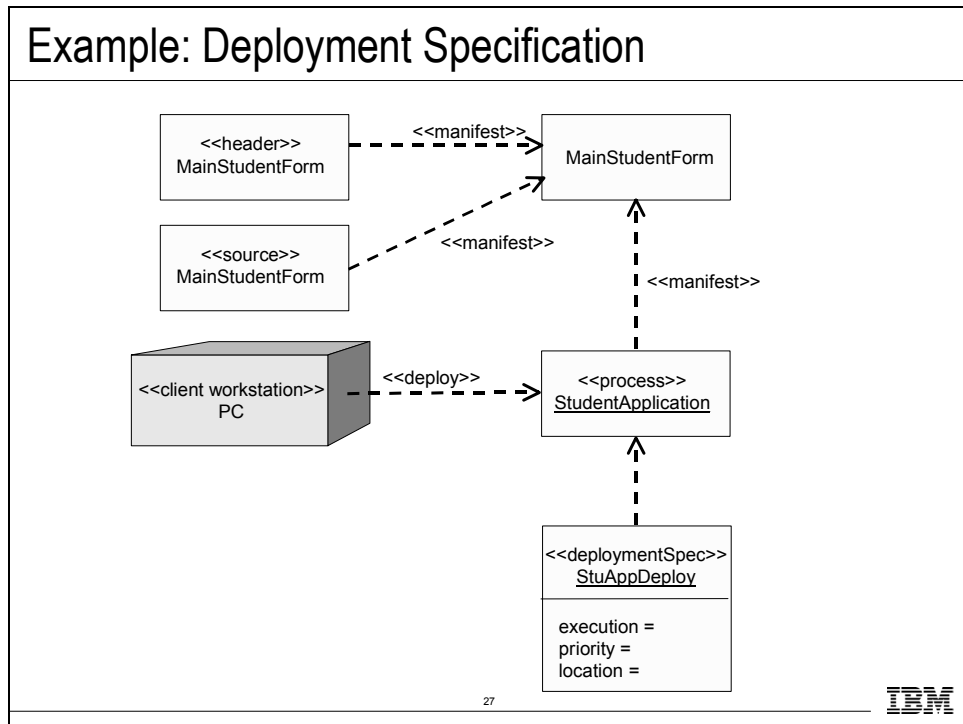
What is a Deployment Specification?

- ♦ A detailed specification of the parameters of the deployment of an artifact to a node.
 - May define values that parameterize the execution

26

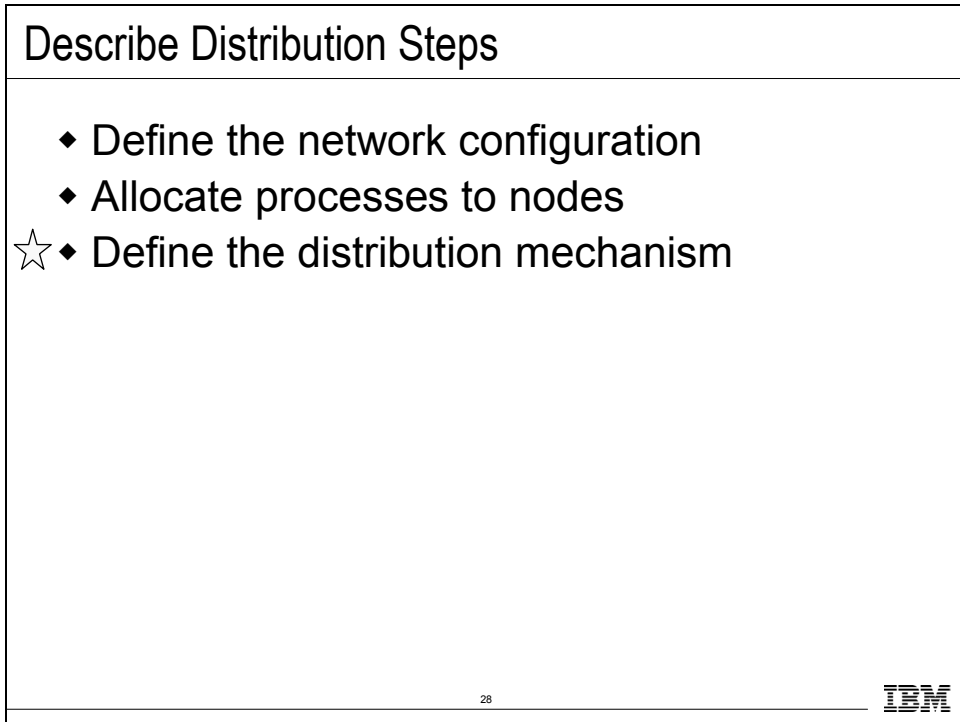


Example: Deployment Specification



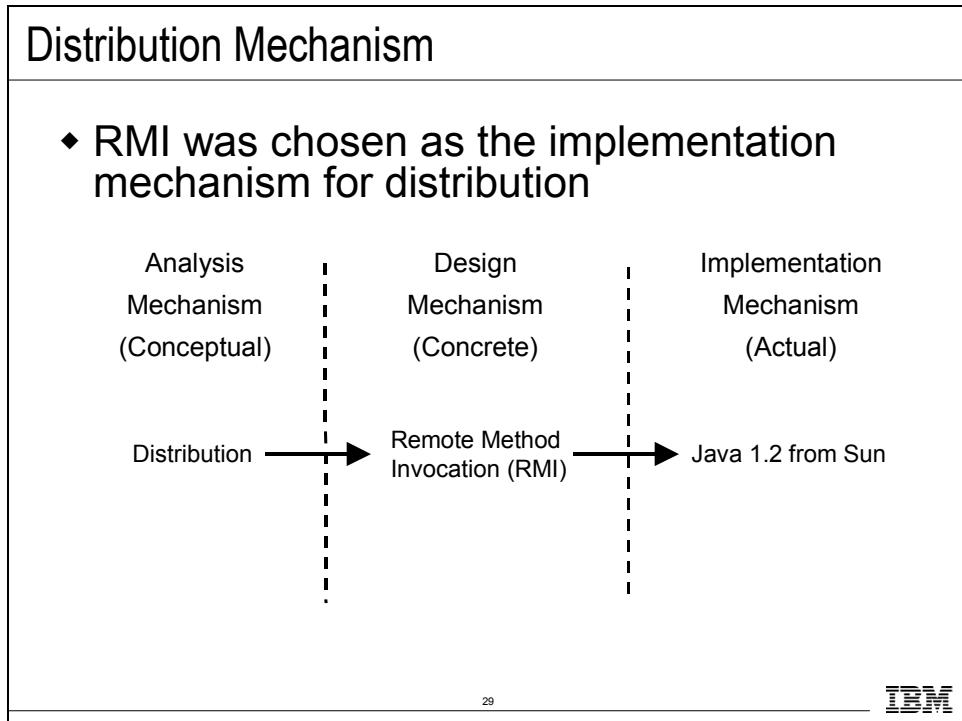
The `<<header>>` and `<<source>>` artifacts are an implementation manifestation for the `MainStudentForm` model element class.

Describe Distribution Steps



If you remember from Architectural Analysis, distribution was identified as an important architectural mechanism. In this section, we will discuss the patterns of use for incorporating the chosen distribution mechanism.

Distribution Mechanism



As mentioned in Identify Design Mechanisms, Remote Method Invocation (RMI) was chosen as the implementation mechanism for distribution.

Lightweight RMI was chosen as a low-cost alternative to a full CORBA distribution. The use of CORBA would be reconsidered if you needed to connect to non-Java systems or in cases where you required features of CORBA that were not found in RMI.

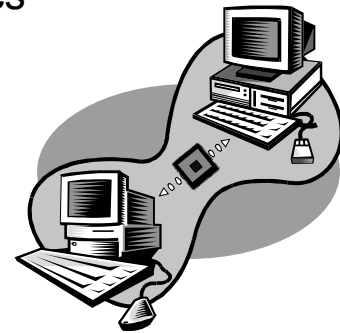
The details of the RMI mechanisms are provided in the Additional Information Appendix, RMI Mechanism section.

Design Mechanisms: Distribution: RMI

Design Mechanisms: Distribution: RMI

- ◆ Distribution characteristics

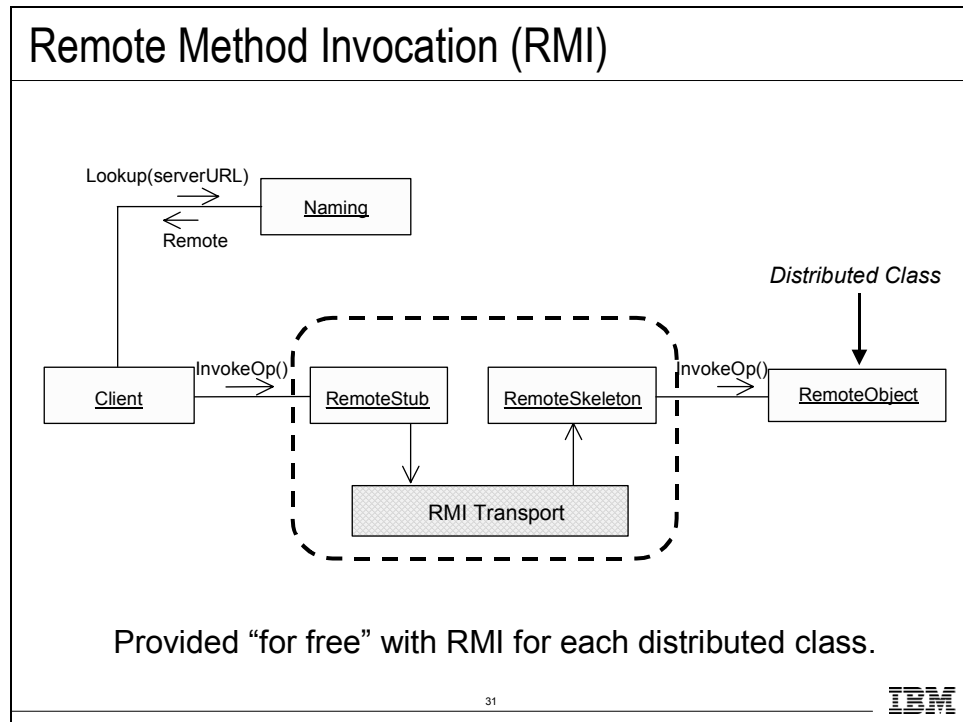
- Latency
- Synchronicity
- Message Size
- Protocol



30

IBM

Remote Method Invocation (RMI)



Remote Method Invocation (RMI) is a Java-specific mechanism that allows client objects to invoke operations on server objects as if they were local. The only catch is that, with basic RMI, you must know where the server object resides.

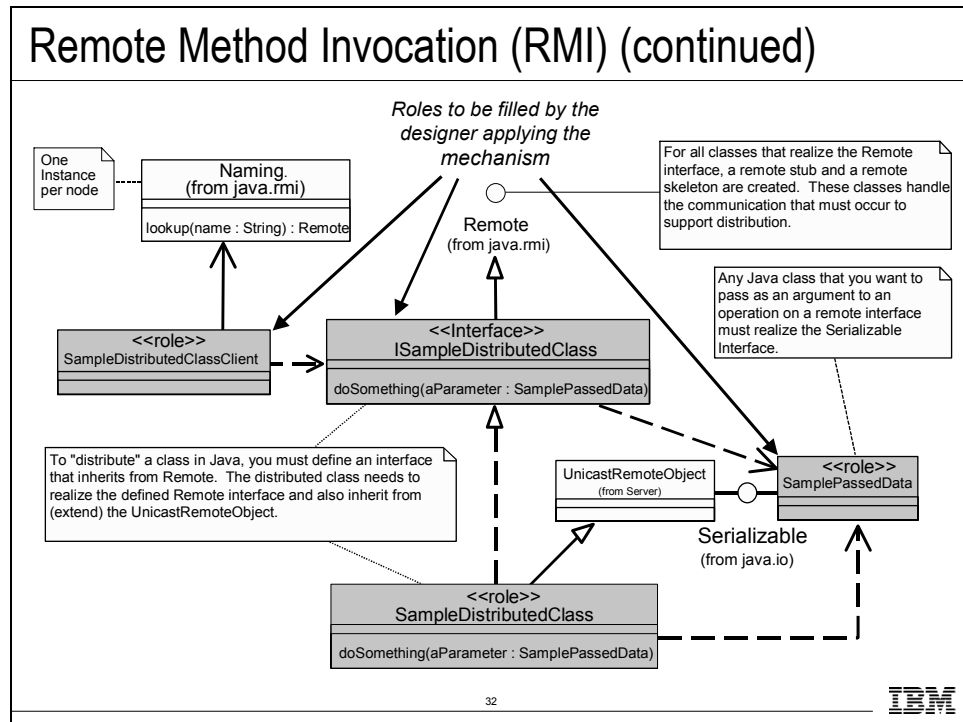
The mechanisms of invoking an operation on a remote object are implemented using “proxies” on the client and server, as well as a service that resides on both that handles the communication.

The client establishes the link with the remote object via the Naming utility that is delivered with RMI. There is a single instance of the Naming class on every node. The Naming instances communicate with one another to locate remote objects. Once the connection is established (via lookup()), it may be reused any time the client needs to access the remote object.

The above diagram describes what happens “under the hood,” but in reality, you do not need to model the RemoteStub and RemoteSkeleton since these are automatically generated by tools from Sun. To get them, you run the compiled distributed class through the rmic compiler to generate the stubs and skeletons. You then must add the code to look up the object on the server. The lookup returns a reference to the auto-generated RemoteStub.

For example, say we had a class, ClassA, that is distributed through RMI. Once ClassA is created, it is run through the rmic compiler, which generates the stub and skeleton. When you do the lookup, the Naming object returns a reference to a ClassA, but it is really a ClassA stub. Thus, no client adjusting needs to happen. Once a class is run through rmic, you can access it as if it were a local class, the client does not know the difference.

Remote Method Invocation (RMI) (continued)



The above diagram describes the pattern that will be used to implement the distribution mechanism. It provides a static view of the classes needed to incorporate the RMI distribution mechanism.

As discussed in the Identify Design Mechanisms module, the `<<role>>` stereotype can be used to denote those elements that need to be supplied by the designer incorporating the mechanism. The roles for the RMI distribution mechanism are shown above.

Note: In the above example the **ISampleDistributedClass** interface has a stereotype of `<<interface>>`, rather than role. This is because it must be an interface, and a class can only have one stereotype.

To "distribute" a class in Java, you must define an interface that inherits from **Remote**. For all classes that realize the **Remote** interface, a remote stub and a remote "skeleton" are created. These classes handle the communication that must occur to support distribution (see the previous slide).

The distributed class needs to realize the defined **Remote** interface and also inherit from (extend) the **UnicastRemoteObject**.

Any Java class that you want to pass as an argument to an operation on a remote interface must realize the **Serializable** interface. Java RMI uses that interface to marshal instances of the class back and forth across a distributed environment. The same holds true for returned values.

Clients of distributed classes will need to look up the location of the remote object using the provided **Naming** service.

Note: Both **Remote** and **Serializable** have no operations. The interfaces themselves only "declare" an object as either a **Remote** or **Serializable**, but there are no operations to implement.

Incorporating RMI: Steps

Incorporating RMI: Steps

1. Provide access to RMI support classes (e.g., Remote and Serializable interfaces, Naming Service)
 - *java.rmi and java.io package in Middleware layer*
 2. For each class to be distributed:
 - *Controllers to be distributed are in the Application layer*
 - *Dependency from the Application to the Middleware layer is needed to access java packages*
- Deferred {
- Define interface for class that realizes Remote
 - Have class inherit from UnicastRemoteObject

33



The next few slides contain a summary of the steps that can be used to implement the RMI distribution mechanism described in this module. The italicized text on the slide describes the architectural decisions made with regards to RMI for our Course Registration example.

- The RMI support classes can be found in the java.rmi (including the nested Server package) and the java.io packages. These packages reside in the Middleware layer.
- For any class that is to be distributed, an interface must be defined that realizes the Java Remote interface. The distributed class will need to realize that interface, as well as inherit from UnicastRemoteObject.
- For the Course Registration System, the control classes are distributed. (The classes to be distributed were tagged with the analysis mechanism, distribution.) The interface classes that are defined for the distributed control classes are placed in the same package as the associated distributed control classes. The control classes were allocated to the Application layer of the architecture in Identify Design Mechanisms. Thus, a dependency must be added from the Application layer to the Middleware layer so the control class interfaces can access the Remote interface, and so the distributed class can access to UnicastRemoteObject class.
- The definition of the distributed class interfaces, and the generalization and realization relationships described above have been deferred until detailed design (for example, Use-Case and Subsystem Design).

The remaining steps are discussed on the subsequent slides.

Incorporating RMI: Steps (continued)

Incorporating RMI: Steps (continued)

3. Have classes for data passed to distributed objects realize the Serializable interface

- *Core data types are in Business Services layer*
- *Dependency from Business Services layer to the Middleware layer is needed to access java.rmi*
- Add the realization relationships } *Deferred*

4. Run pre-processor } *Out of scope*

34



- Any class whose instances are passed between the client and the server needs to realize the Serializable interface.
- For the Course Registration System, most of the data passed will be of one of the core data types. The core data types were allocated to the Business Services layer of the architecture (the University Artifacts package, specifically) in Identify Design Elements. Thus, a dependency is needed from the Business Services layer to the Middleware layer so the core data classes can access the Remote interface. This relationship already exists.
- The definition of the realizes relationships from the classes to be passed and the Serializable interface has been deferred until detailed design (for example, Use-Case and Subsystem Design).
- The developer must run the compiled distributed class through the rmic compiler provided by Sun to generate the stubs and skeletons for all classes that realize the Remote interface. These classes handle the communication that must occur to support distribution (see previous slide). Once a class is run through rmic, you can access it as if it were a local class. The client does not know the difference. This is really implementation, and thus, is out of the scope of this course.

The remaining steps are discussed on the next slide.

Incorporating RMI: Steps (continued)

Incorporating RMI: Steps (continued)

5. Have distributed class clients lookup the remote objects using the Naming service

- *Most Distributed Class Clients are forms*
- *Forms are in the Application layer*
- *Dependency from the Application layer to the Middleware layer is needed to access java.rmi*
- Add relationship from Distributed Class Clients to Naming Service

6. Create and update interaction diagrams with distribution processing

} *Deferred*

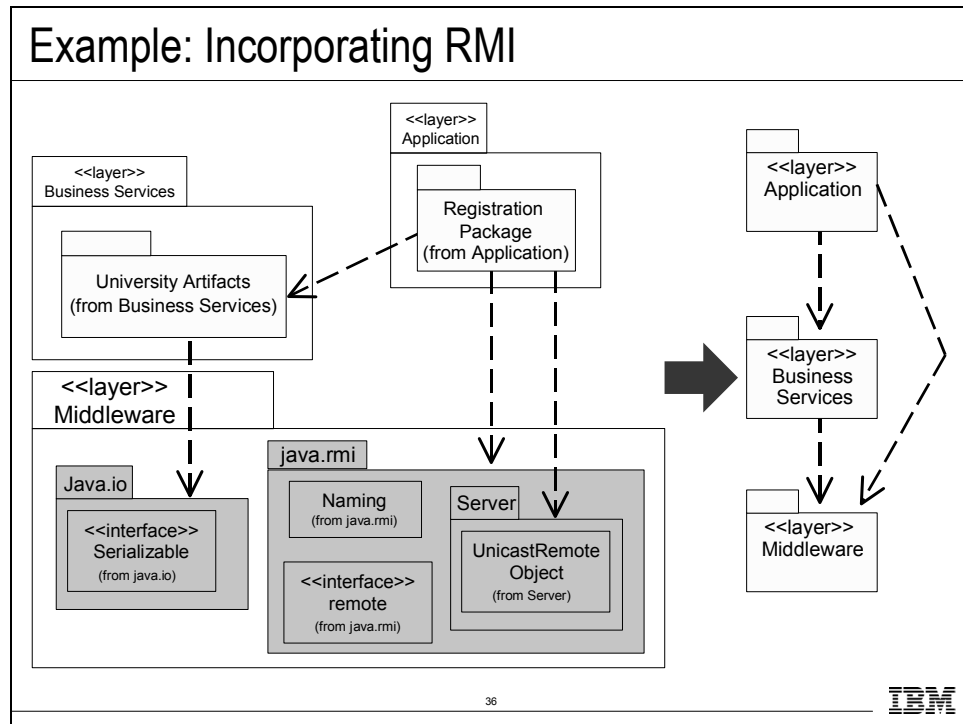
35

IBM

- Clients of distributed classes will need to look up the location of the remote object using the Naming service. The lookup returns a reference to the distributed class interface.
- In the Course Registration System, the distributed control class clients are forms, and the forms were allocated in the Application layer of the architecture in Identify Design Elements. Thus, a dependency must be added from the Application layer to the Middleware layer so the control classes can access the Naming service.
- The definition of the dependency relationship from the distributed class clients and the Naming Service has been deferred until detailed design (for example, Use-Case and Subsystem Design).
- The developer must create Interaction diagrams that model the added distribution functionality. Do not be too concerned with modeling the processing that is provided by the distribution infrastructure (for example, the Remote Stub and Skeleton).

Just model the use of the Naming service and the distributed class interface. The development of any Interaction diagrams has been deferred until detailed design (for example, Use-Case and Subsystem Design).

Example: Incorporating RMI



The above diagram describes the package dependencies needed to support the RMI distribution mechanism. The sample application represents any application package that contains controllers that will need to be distributed along with their associated clients. The following package dependencies were added to support distribution.

The java.rmi package containing the classes that implement the RMI distribution mechanism.

- Dependency from Application packages to java.rmi to provide access to the Remote interface for distributed controller interfaces, and to the Naming service for the distributed controller clients.
- Dependency from the Application packages to the Java Server package to provide access to the UnicastRemoteObject class for distributed controllers.
- Dependency from University Artifacts to java.io to provide access to the Serializable interface for classes whose instances must be passed for distributed objects.

These package dependencies required the following change to the layer dependencies originally defined in Identify Design Elements:

- Dependency from Application layer to Middleware layer to support the new dependency from the Application packages to the java.rmi package.

This is an example of how the architecture might need to be adjusted as design details are added. Such changes are architectural changes that *must* be made by the architect. Note: In the above diagram, only a subset of the packages in the layers are shown. The remaining packages in the layers have been omitted for clarity.

Checkpoints: Deployment View

Checkpoints: Deployment View

- ◆ Have the distributed data update coordination and synchronization issues been addressed and documented?
- ◆ Are services that require more rapid response available locally (LAN vs. WAN)?
- ◆ Have all redundant server issues been addressed and documented (primary vs. secondary)?
- ◆ Are the failure modes documented?



37

IBM

Above are the key things that a designer would look for when assessing the results of the **Describe Distribution** activity. As stated earlier, an architect would have a more detailed list.

Make sure the following points are addressed and well documented:

- Consider how to coordinate and synchronize updates to data that is distributed across multiple nodes. This usually involves some form of a two-phase commit protocol. The situation where multiple copies of the same object can become out of sync with one another needs to be prevented, or recovered from.
- Make sure that services that require more rapid response are available locally via a LAN rather than WAN.
- Consider the situation where more than one server "thinks" it is primary, or the situation where no processors end up being primary needs to be prevented, or recovered from.
- Consider what should happen if a server goes down.

Review

Review: Describe Distribution

- ♦ What is the purpose of the Describe Distribution activity?
- ♦ What is a node? Describe the two different “types” of nodes.
- ♦ Describe some of the considerations when mapping processes to nodes.
- ♦ How do you model the Deployment View? What modeling elements and diagrams are used?

38



Exercise: Describe Distribution

Exercise: Describe Distribution

- ♦ Given the following textual information:
 - Network configuration (e.g., nodes and their connectors)
 - What processes run on what nodes?
 - ♦ Exercise Workbook: *Payroll Architecture Handbook*, Deployment View section



39

IBM

In this exercise, the deployment architecture for the course exercise is given, textually. The architecture is not derived, since such a derivation is out of the scope of this course. The exercise allows the student to produce the visual model of the described deployment architecture.

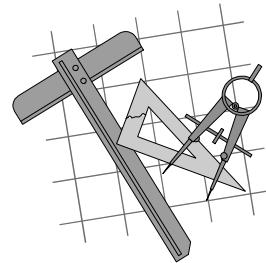
References:

- Exercise Workbook: *Payroll Architecture Handbook*, Deployment View section.

Exercise: Describe Distribution (continued)

Exercise: Describe Distribution (continued)

- ◆ Produce the following:
 - A Deployment diagram depicting:
 - Nodes
 - Connectors
 - What processes run on what nodes



40

IBM

Refer to the following slides if needed:

- What is a Node? – 10-16
- What is a Connector? – 10-17
- Example: Deployment Diagram with Processes – 10-21

Exercise: Review

Exercise: Review

- ♦ Compare your Deployment Model with those developed by the rest of the class.
 - Have nodes and node connections been modeled?
 - Have processes been identified and assigned to nodes? Do the allocations make sense?
 - Are the processes listed beneath the nodes in the Deployment diagram?



Payroll System

41



After completing a model, it is important to step back and review your work. Some helpful questions are the following:

- Have nodes and node connections been modeled?
- Have processes been identified and assigned to nodes? Do the allocations make sense? The processes identified in the Deployment Model should correspond to those in the Process Model. Threads should not appear on the Deployment Model, because threads always execute within the context of their parent process.
- Are the processes listed beneath the nodes in the Deployment diagram?

