Unit – I

Introduction to Distributed System

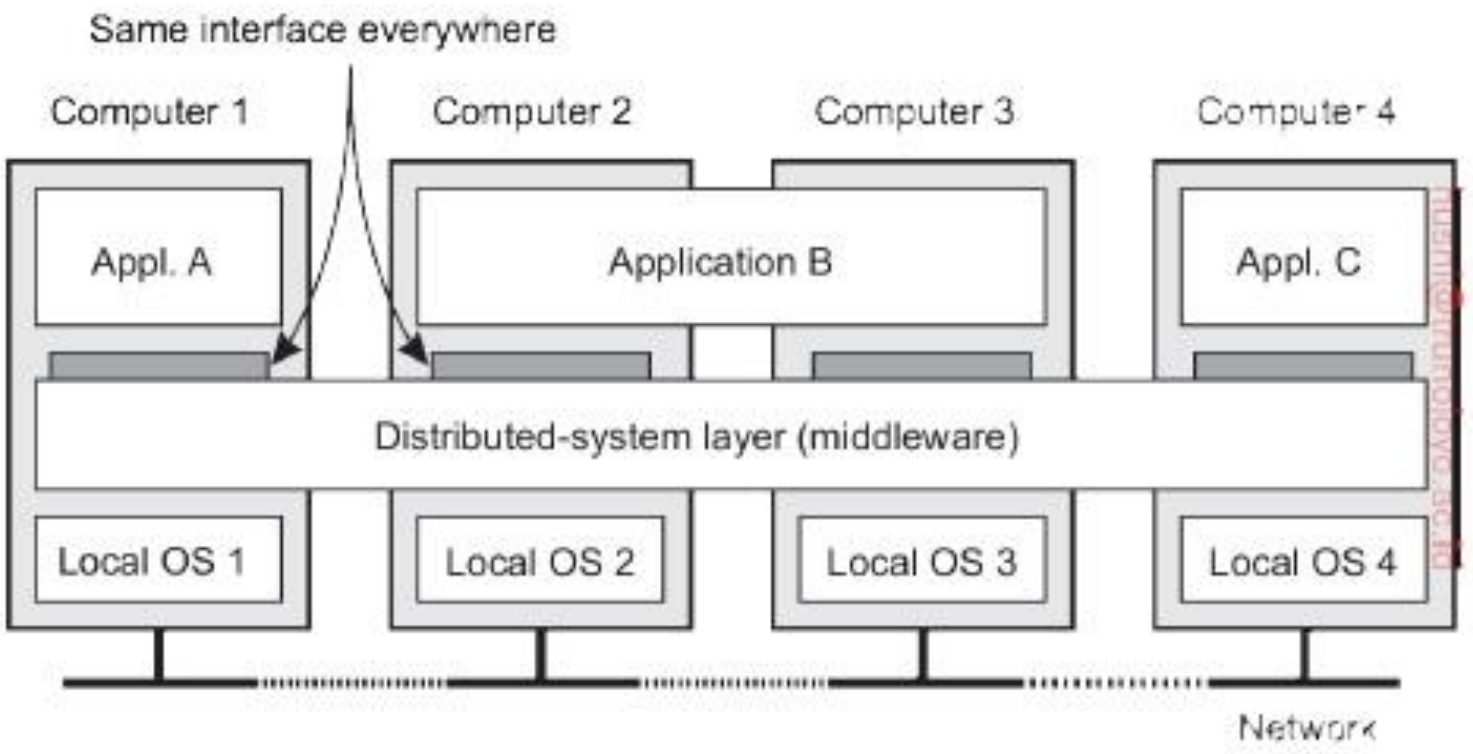
Introduction to Distributed System

- A Distributed System is one in which components located at networked computers communicate and coordinate their actions only by passing messages. **OR**
- A DS is a collection of independent computers that appears to its users as a single coherent system.
- The sharing of resources is a main motivation for constructing distributed systems.
- Resources may be managed by servers and accessed by clients.
- Components/Computers in DS, that are autonomous.
- Users (people/program) think that, they are dealing with a single system.
- This means that one way or the other the autonomous components need to collaborate.
- How this collaboration lies between them, there is need to develop DS accordingly.
- To develop a DS, no assumptions are made concerning the type of computers/components., and the way that computers are interconnected.

Introduction to Distributed System

- To have a better DS, the way in which various computers communicate with each others, should be hidden from users.
- The Users and Applications can interact with a DS in a consistent and uniform way, regardless of where and when interaction takes place.
- The DS should also be relatively easy to expand or scale.
- A DS will normally be continuously available, unless & until some parts may be temporarily out of order.
- So Users and applications should not notice that parts are being replaced or fixed or that new parts are added to serve more users or applications.
- In order to support heterogeneous computers and networks while offering a single-system view, DS are often organized by means of layer of software.
- It logically placed between a higher-level of users and applications and OS and basic communication facilities, so also called as Middleware.

Introduction to Distributed System



-

Examples of Distributed System

- **Internet**
  - ---- Interconnected, WWW, large DS,
  - ---- Communicates by message passing,
  - ---- Sharing resources,
  - ---- Uses Internet protocols,
- **Intranet**
  - ---- A portion of Internet,
  - ---- Has a boundary & configured to enforce local security policies,
  - ---- Use of Firewall,
  - ---- A moderate DS
  - ---- Same as Internet,
  - ---- Services like, - File sharing, Printer, Use of ERP

Examples of Distributed System

- **Mobile Computing**

---- Highly supports wireless networks,

---- Integrates wired and wireless N/Ws,

---- e.g. Laptops, PDAs, Cell phones, pagers, video cameras, digital cameras,

---- Wearable devices like, smart watches, health related gadgets,

---- Embedded devices like, washing machines, wi-fi systems, cars etc,

---- Portability of these devices, together makes Mobile Computing easier,

- **Ubiquitous Computing**

---- Services for many small, cheap computational devices that are present in our physical environments,

---- The computational behavior of these devices will b e transparent and intimately tied up with their physical functions,

---- The presence of computers everywhere only becomes useful when they can communicate with each other.

Goals of Distributed System

- A DS should make resources easily accessible.
- It should reasonably hide the facts that resources are distributed across a network.
- It should be open.
- It should be scalable.
- It should be run on Web.
- To become a backbone for Web services – HTTP, URL, HTML TCP/IP, etc.

Trends in DS

- Distributed systems are undergoing a period of significant change and this can be traced back to a number of influenced trends:

✓ The emergence of pervasive networking technology;

✓ The emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems;

✓ The increasing demand for multimedia services;

✓ The view of distributed systems as a utility such as Physical resources for data centers, Software services, Cloud computing, Cluster computing etc.

DS Focuses on Sharing Resources

- Users should always get to the benefits of resource sharing that they may easily overlook their significance.
- We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.
- H/w sharing in the N/w.
- Data sharing in the form of shared DB, or set of web pages.
- Similarly, users think in terms of shared resources such as a search engine or a currency converter, without regard for the server or servers that provide these.
- Patterns of resource sharing vary widely in their scope and in how closely users work together.
- At one extreme, a search engine on the Web provides a facility to users throughout the world, users who need never come into contact with one another directly.
- Directly share resources such as documents in a small, closed group.

DS Focuses on Sharing Resources

- The pattern of sharing and the geographic distribution of particular users determines what mechanisms the system must supply to coordinate users' actions.
- The only access we have to the various services is via the set of operations that it exports. For example, a file service provides *read, write and delete operations on files.*
- So DS restrict resource access to a well-defined set of operations is in the part i.e. standard software engineering practice.
- But it also reflects the physical organization of DS.
- Resources in a DS are physically encapsulated within computers and can only be accessed from other computers by means of communication.
- For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.
- This would be done through the Client-Server Computing.

DS Focuses on Sharing Resources

- When the client sends a request for an operation to be carried out, we say that the client *invokes an operation* upon the server.
- A complete interaction between a client and a server, from the point when the client sends its request to when it receives the server's response, is called a *remote invocation.*
- Not only the clients but, servers sometimes invoke operations on other servers.

Challenges in DS

- To the successful DS, there are some significant challenges were encountered and should be overcome.
- As the scope and scale of DS and applications is extended the same and other challenges are likely to be encountered.

Heterogeneity *(variety and difference)**:***

- The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks.
- Heterogeneity applies to all of the following:

  Networks, Computer Hardware, Operating System,

  Programming Languages, Implementations by different developers

Challenges in DS

- The term *middleware applies to a software layer that provides a* programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages.
- In addition to solving the problems of heterogeneity, middleware provides a uniform computational model for use by the programmers of servers and distributed applications. Possible models include remote object invocation, remote event notification, remote SQL access and distributed transaction processing.
- E.g. CORBA provides remote object invocation, which allows an object in a program running on one computer to invoke a method of an object in a program running on another computer. Its implementation hides the fact that messages are passed over a network in order to send the invocation request and its reply.

Challenges in DS

Openness:

- The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways.
- The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.
- Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers.
- The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people.
- Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.

Challenges in DS

- Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

Security:

- Many of the information resources that are made available and maintained in distributed systems have a high important value to their users.
- Their security is therefore of considerable importance in DS.
- Security for information resources has basically three components:

  -- Confidentiality (protection against disclosure to unauthorized individuals),

  -- Integrity (protection against alteration or corruption),

  -- Availability (protection against interference with the means to access the resources).

Challenges in DS

- In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network.
- E.g.1 A doctor might request access to hospital patient data or send additions to that data.
- E.g.2 In electronic commerce and banking, users send their credit card numbers across the Internet.
- In both examples, the challenge is to send sensitive information in a message over a network in a secure manner.
- Both of these challenges can be met by the use of encryption techniques developed for this purpose.

Challenges in DS

Scalability:

• Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet.

• A system is described as *scalable if it will remain* effective when there is a significant increase in the number of resources and the number of users.

• The number of computers and servers in the Internet has increased dramatically.

• One web server may also increasingly be hosted on multiple computers.

• The design of scalable distributed systems presents the following challenges,

✓ **Controlling the cost of physical resources:** demand for resources grows, extend the systems, bandwidth sharing, increased cost

✓ **Controlling the performance loss:** number of system in network increases, bandwidth get shared, effects on performance.

Challenges in DS

✓ **Preventing software resources running out:** An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet). Change in 32-bit address to 128-bit address, requires modifications to many software components.

✓ **Avoiding performance bottlenecks:** algorithms should be decentralized to avoid having performance bottlenecks.

Failure Handling:

• Computer systems sometimes fail.

• When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation.

• Failures in a DS are partial – that is, some components fail while others continue to function.

• Therefore the handling of failures is particularly difficult.

Challenges in DS

• The following techniques for dealing with failures are used,

✓ **Detecting Failures:** Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file.

It is difficult or even impossible to detect some other failures, such as a remote crashed server in the Internet.

The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

✓ **Masking Failures:** Some failures that have been detected can be hidden or made less severe. E.g. Messages can be re-transmitted when they fail to arrive, File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

✓ **Tolerating Failure:** Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components.

Challenges in DS

E.g. when a web browser cannot contact a web server, it does not make the user wait for ever while it keeps on trying – it informs the user about the problem, leaving them free to try again later.

✓ **Recovery from Failures:** Recovery involves the design of software so that the state of permanent data can be recovered or 'rolled back' after a server has crashed.

✓ **Redundancy:** Services can be made to tolerate failures by the use of redundant components.

There should always be at least two different routes between any two routers in the Internet.

In the Domain Name System, every name table is replicated in at least two different servers.

A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server.

Challenges in DS

Concurrency:

• Both services and applications provide resources that can be shared by clients in a distributed system.

• There is therefore a possibility that several clients will attempt to access a shared resource at the same time.

• The process that manages a shared resource could take one client request at a time.

• But that approach limits throughput.

• Therefore services and applications generally allow multiple client requests to be processed concurrently.

• To make this more concrete, suppose that each resource is encapsulated as an object and that invocations are executed in concurrent threads.

Challenges in DS

Transparency:

• Transparency is defined as hiding the separation of components in a DS from the user and the application programmer.

• So that the system looks as a whole rather than as a collection of independent components.

✓ **Access transparency** enables local and remote resources to be accessed using identical operations.

✓ **Location transparency** enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

✓ **Concurrency transparency** enables several processes to operate concurrently using shared resources without interference between them.

✓ **Replication transparency** enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.

Challenges in DS

✓ **Failure transparency** enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

✓ **Mobility transparency** allows the movement of resources and clients within a system without affecting the operation of users or programs.

✓ **Performance transparency** allows the system to be reconfigured to improve performance as loads vary.

✓**Scaling transparency** allows the system and applications to expand in scale without change to the system structure or the application algorithms.

Challenges in DS

Quality of Service:

• Once users are provided with the functionality that they require of a service, such as the file service in a DS, we can go on to ask about the quality of the service provided.

• The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are reliability, security and performance.

• Reliability and security issues are critical in the design of most computer systems.

• The performance aspect of quality of service was originally defined in terms of responsiveness and computational throughput, but it has been redefined in terms of ability to meet timeliness guarantees.

System Models

• Distributed Systems that are intended for use in real-world environments should be designed to function correctly in the widest possible range of circumstances.

• That may face possible difficulties and threats.

• DS of different types share important underlying properties and give rise to common design problems.

• For any DS how the properties and design issues can be captured and discussed through the use of descriptive models.

• Each type of model is intended to provide an abstract, simplified but consistent description of a relevant aspect of DS design,

1. Physical Models
2. Architectural Models
3. Fundamental Models

System Models

Quick Check

Physical Models:

• Physical models are the most explicit way in which to describe a system.

• They capture the hardware composition of a system in terms of the computers and their interconnecting networks on any geographical situations.

Architectural Models:

• Architectural models describe a system in terms of the computational and communication tasks performed by its computational elements.

• The computational elements being individual computers or collection of them supported by appropriate network interconnections.

System Models

Fundamental Models:

• Fundamental models take an abstract perspective in order to examine individual aspects of a distributed system.

• Important aspects of Functional Models are:

1. **Interaction Models**: which consider the structure and sequencing of the communication between the elements of the system.
2. **Failure Models:** which consider the ways in which a system may fail to operate correctly.
3. **Security Models:** which consider how the system is protected against attempts to interfere with its correct operation or to steal its data.

Difficulties and Threats for DS

• Here are some of the problems that the designers of DS face.

Widely varying modes of use:

• The component parts of systems are subject to wide variations in workload – for example, some web pages are accessed several million times a day.

• Some parts of a system may be disconnected, or poorly connected some of the time – for example, when mobile computers are included in a system.

• Some applications have special requirements for high communication bandwidth and low latency – for example, multimedia applications.

Internal problems:

• Non-synchronized clocks, conflicting data updates and many modes of hardware and software failure involving the individual system components.

Difficulties and Threats for DS

External threats:

• Attacks on data integrity and secrecy, denial of service attacks.

Wide range of system environments:

• A DS must accommodate heterogeneous hardware, operating systems and networks.

• The networks may differ widely in performance – wireless networks operate at a fraction of the speed as compare with local networks.

• Systems of widely differing scales, ranging from tens of computers to millions of computers, must be supported.

Physical Models

• A physical model is a representation of the underlying hardware elements of a DS that abstracts away from specific details of the computer and networking technologies employed.

Baseline physical model:

• A DS in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.

• This leads to a minimal physical model of a DS as an extensible set of computer nodes interconnected by a computer network for the required passing of messages.

• Beyond this baseline model, we can usefully identify three generations of DS.

Physical Models

Early distributed systems:

- Such systems emerged in the late 1970s and early 1980s in response to the emergence of local area networking technology, usually Ethernet.
- These systems typically consisted of between 10 and 100 nodes interconnected by a local area network, with limited Internet connectivity.
- This were supported a small range of services such as shared local printers and file servers as well as email and file transfer across the Internet.
- Individual systems were largely homogeneous and openness was not a primary concern.
- Providing quality of service was still very much difficult.

Physical Models

Internet-scale distributed systems:

• Building on this foundation, larger-scale distributed systems started to emerge in the 1990s in response to the dramatic growth of the Internet during this time.

• E.g., the Google search engine was first launched in 1996.

• In such systems, the underlying physical infrastructure consists of a physical model that is, an extensible set of nodes interconnected by a *network of networks (the Internet).*

• Such systems exploit the infrastructure offered by the Internet to become truly global.

• They incorporate large numbers of nodes and provide distributed system services for global organizations and across organizational boundaries.

• The level of heterogeneity in such systems is significant in terms of networks, computer architecture, operating systems, languages employed and the development teams involved.

Physical Models

• This has led to an increasing emphasis on open standards and associated middleware technologies such as CORBA and more recently, web services.

• Additional services were employed to provide end-to-end quality of service properties in such global systems.

Contemporary distributed systems:

• In these systems, nodes were typically desktop computers and therefore relatively static, discrete and autonomous.

• The emergence of mobile computing has led to physical models

• Where nodes such as laptops or smart phones may move from location to location in a distributed system are going to attach.

• Leading to the need for added capabilities such as service discovery and support for spontaneous interoperation.

Physical Models

- The emergence of ubiquitous computing has led to a move from discrete nodes to architectures.
- Where computers are embedded in everyday objects and in the surrounding environment
- E.g., in washing machines or in smart homes more generally.

- The emergence of cloud computing and cluster architectures has led to a move from autonomous nodes.
- Performing a given role to pools of nodes that together provide a given service.
- E.g. a search service as offered by Google.

Physical Models

Distributed systems of systems :

• A recent report discusses the emergence of ultra large- scale (ULS) DS.

• The report captures the complexity of modern DS by referring to physical architectures as systems of systems.

• Mirroring the view of the Internet as a network of networks.

• A system of systems can be defined as a complex system consisting of a series of subsystems that are systems in their own right and that come together to perform a particular task or tasks.

• An example of a system of systems, consider an environmental management system for flood prediction.

• In such a scenario, there will be sensor networks deployed to monitor the state of various environmental parameters relating to rivers, flood plains, tidal effects and so on.

Physical Models

- This can then be coupled with systems that are responsible for predicting the floods, by running simulation systems on, E.g. Cluster Computers.
- Other systems may be established to maintain and analyze historical data or to provide early warning systems to key stakeholders via mobile phones.

Architectural Models

- The architecture of a system is nothing but structure.
- This structure is in terms of separately specified components and their interrelationships.
- The overall goal is to ensure that the structure will meet present and likely future demands on it.
- Major concerns are to make the system reliable, manageable, adaptable and cost-effective.
- E. g - The architectural design of a building has similar aspects – it determines not only its appearance but also its general structure and architectural style (classical, modern) and provides a consistent frame of reference for the design.
- In this section we describe the main architectural models employed in DS and the architectural styles of DS.

Architectural Models

• To understand the fundamental building blocks of a DS, it is necessary to consider four key questions:

✓ What are the entities that are communicating in the DS?

✓ How do they communicate, or, more specifically, what communication paradigm is used?

✓ What roles and responsibilities do they have in the overall Architecture?

✓ How are they mapped on to the physical distributed infrastructure (what is their placement)?

• The first two questions are absolutely central to an understanding of DS; what is communicating and how those entities communicate together define a rich design space for the DS developer to consider.

• It is helpful to address the first question from a system-oriented and a problem-oriented perspective.

Architectural Models

• From a system perspective, the answer is normally very clear in that the entities that communicate in a DS are typically processes.

• In most DS environments, processes are supplemented by threads, so it is threads that are the endpoints of communication.

• From a programming perspective of any DS, more problem-oriented abstractions have been proposed:

Communicating Entities:

Objects:

• Objects have been introduced to enable and encourage the use of object oriented approaches in distributed systems.

• In distributed object-based approaches, a computation consists of a number of interacting objects.

• Objects are accessed via interfaces, with an associated interface definition language (IDL) providing a specification of the methods defined on an object.

Architectural Models

Components:

• Components differs from objects in that they offer problem-oriented abstractions for building distributed systems and are also accessed through interfaces.

• The key difference is that components specify not only their (provided) interfaces but also the assumptions they make.

• Assumptions in terms of other components/interfaces that must be present for a component to fulfill its function.

• Making all dependencies explicit and providing a more complete contract for system construction.

• A purer compositional approach to constructing DS by removing hidden Dependencies.

Architectural Models

Web Services:

• Web services represent the third important paradigm for the development of DS.

• Web services are closely related to objects and components with approach based encapsulation of behavior and access through interface.

• Web services are basically integrated into the World Wide Web, using web standards to represent and discover services.

Communication Paradigms:

• How entities communicate in a DS gives us three types of communication paradigms,

✓ Interprocess Communication,

✓ Remote Invocation,

✓ Indirect Communication

Architectural Models

• **Interprocess communication** refers to the relatively low-level support for communication between processes in DS including message-passing primitives, direct access to the API offered by Internet protocols (socket programming) and support for multicast communication.

• **Remote invocation** represents the most common communication paradigm in DS, covering a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method.

- ✓ **Request-Reply Protocols:** Client-Server Communication
- ✓ **Remote Procedure Calls:** Hides important aspects of distribution
- ✓ **Remote Method Invocation:** A calling object can invoke a method in a remote object.

Architectural Models

• **Indirect communication** allows a strong degree of decoupling between senders and receivers.

• Senders do not need to know who they are sending to (*space uncoupling).*

• Senders and receivers do not need to exist at the same time (*time uncoupling).*

• Key techniques for indirect communication include,

✓ **Group Communication**: A multiparty communication paradigm supporting one-to-many communication.

✓ **Publish-subscribe systems**: An information spreading systems, e.g. Share/Financial Trading.

✓ **Message Queues:** Offer a point-to-point service whereby producer processes can send messages to a specified queue and consumer processes can receive messages from the queue.

Architectural Models

✓ **Tuple Space**: Processes can place arbitrary items of structured data, called tuples. Since the tuple space is persistent, readers and writers do not need to exist at the same time.

✓ **Distributed Shared Memory**: Provide an abstraction for sharing data between processes that do not share physical memory.

Architectural Models

Roles and Responsibilities:

• In a DS processes, interact with each other to perform a useful activity.

• In doing so, the processes take on given roles, and these roles are fundamental in establishing the overall architecture to be adopted.

• There are two architectural styles stemming from the role of individual processes,

- ✓ Client-Server
- ✓ Peer to Peer

Architectural Models

Placement:

- The final issue to be considered is how entities such as objects or services map on to the underlying physical distributed infrastructure.
- Which will consist of a potentially large number of machines interconnected by a network of arbitrary complexity.
- Placement is crucial in terms of determining the properties of the DS, most obviously with regard to performance but also to other aspects, such as reliability and security.
- The question of where to place a given client or server in terms of machines and processes within machines is a matter of careful design.
- Placement needs to take into account the patterns of communication between entities.
- The reliability of given machines and their current loading, the quality of communication between different machines and so on.

Architectural Models

• Placement must be determined with strong application knowledge, and there are few universal guidelines to obtaining an optimal solution.

✓ **Mapping of services to multiple servers:** The servers may partition the set of objects on which the service is based and distribute those objects between themselves, or they may maintain replicated copies of them on several hosts.

✓ **Caching:** A cache is a store of recently used data objects that is closer to one client or a particular set of clients than the objects themselves. When a new object is received from a server it is added to the local cache store, replacing some existing objects if necessary. When an object is needed by a client process, the caching service first checks the cache and supplies the object from there if an up-to-date copy is available. If not, an up-to-date copy is fetched.

✓ Mobile Code

✓ Mobile Agents

Architectural Models

✓ **Mobile Code:** Applets are a well-known and widely used example of mobile code – the user running a browser selects a link to an applet whose code is stored on a web server; the code is downloaded to the browser and runs there. An advantage of running the downloaded code locally is that it can give good interactive response since it does not suffer from the delays or variability of bandwidth associated with network communication.

✓ **Mobile Agents**: A mobile agent is a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf, such as collecting information, and eventually returning with the results. A mobile agent may make many invocations to local resources at each site it visits. E.g. accessing individual database entries.

Architectural Models

Architectural Patterns:

• They not provide a complete solutions but rather offer a partial insights, when combined with other patterns.

• That leads the designer to a solution for a given problem domain.

• Several key architectural patterns in DS are,

✓ **Layering:**

• The concept of layering is a familiar and closely related to abstraction.

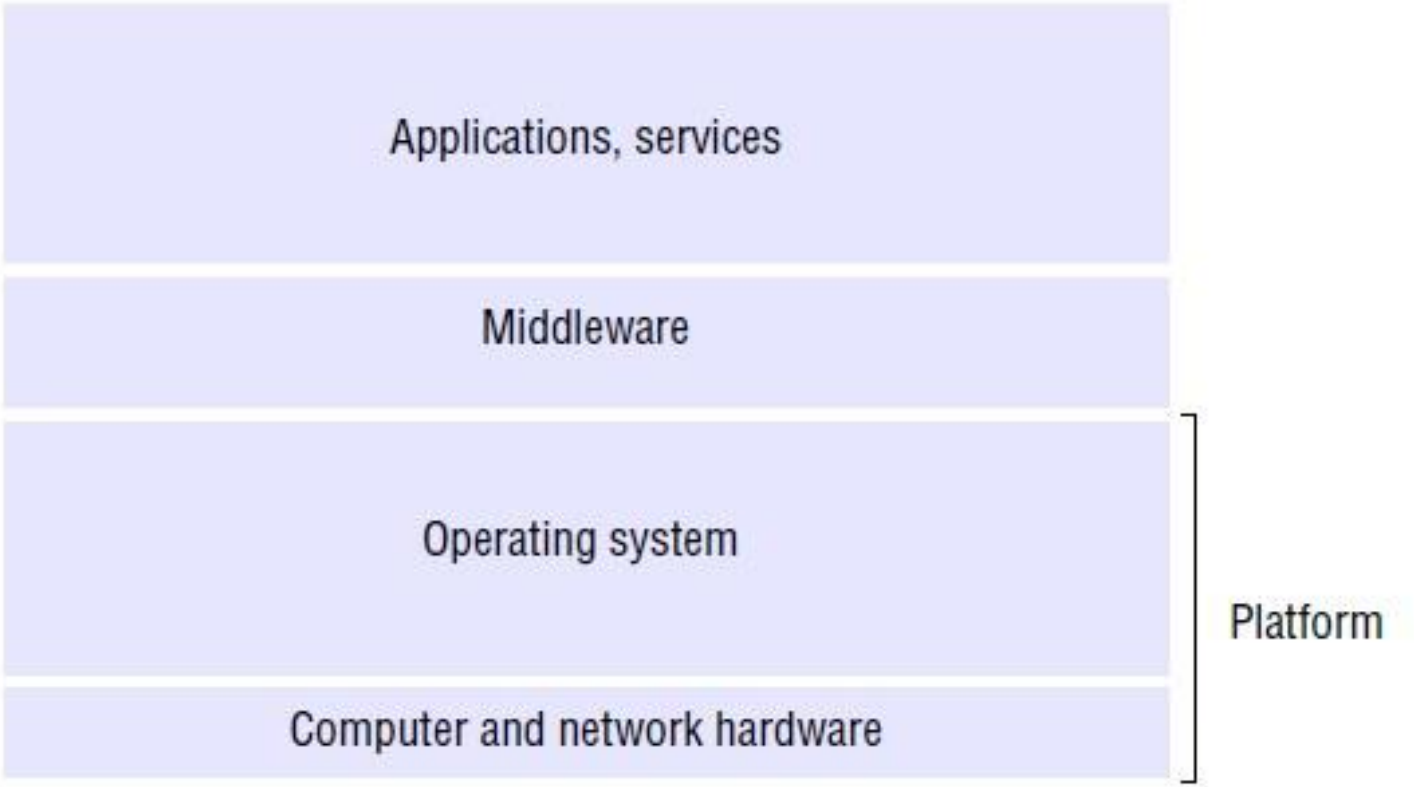
• In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below.

• A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details.

• In terms of DS, this relates to a vertical organization of services into service layers.

Architectural Models

- Software and hardware service layers in distributed systems



Architectural Models

✓ **Tiered Architecture:**

• Tiered architectures are complementary to layering.

• Whereas layering deals with the vertical organization of services into layers of abstraction,

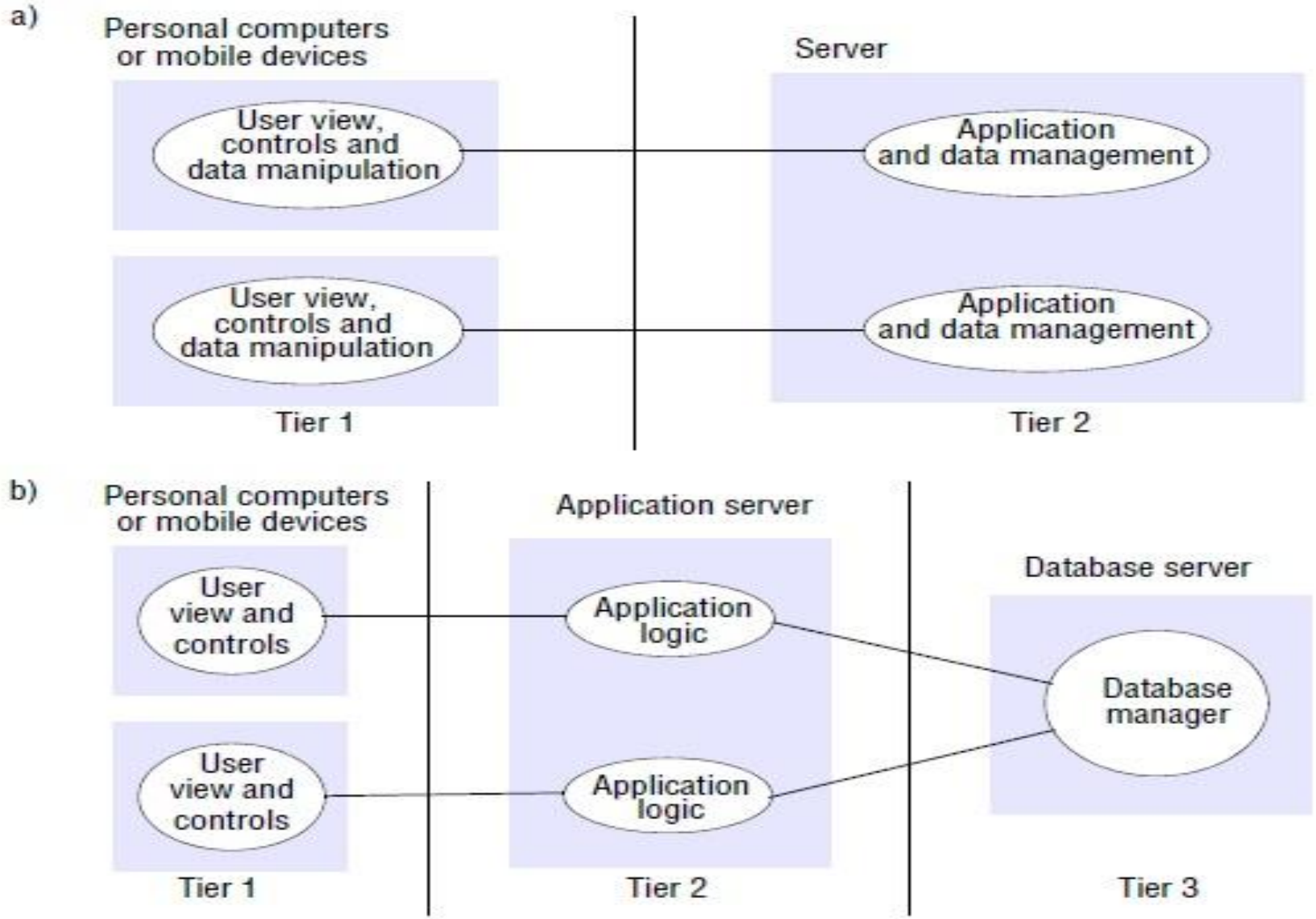
• Tiering is a technique to organize functionality of a given layer and place this functionality into appropriate servers.

• This technique is most commonly associated with the organization of applications and Services.

• But it also applies to all layers of a DS architecture.

Highlight text

Two-tier and three-tier architectures



Architectural Models

✓ **Thin Clients:**

• The trend in distributed computing is moving towards reducing complexity away from the end-user on the Internet.

• This is most apparent in the moving towards cloud computing and also be seen in tiered architectures.

• This trend has given rise to interest in the concept of a thin client, enabling access to sophisticated networked services provided.

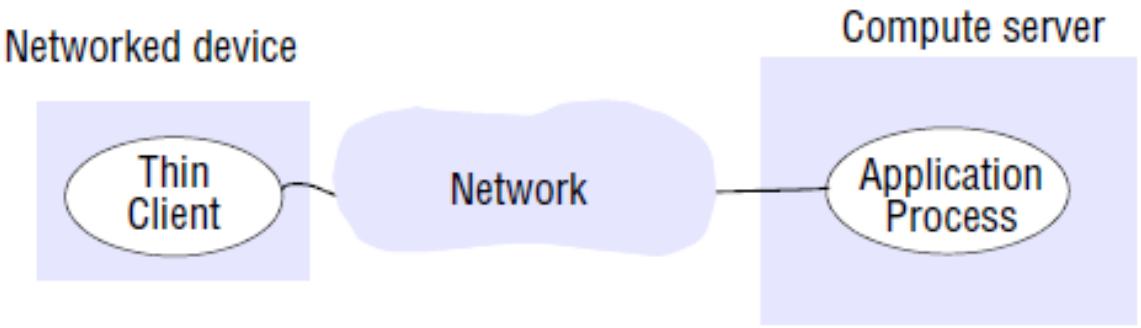
• E.g. By a cloud solution, with few assumptions or demands on the client device.

• More specifically, the term thin client refers to a software layer that supports a window-based user interface that is local to the user while executing application programs.

• Rather more generally, accessing services on a remote computer.

Architectural Models

- Thin clients and computer servers



- The advantage of this approach is that potentially simple local devices can be significantly enhanced with a plethora of networked services and capabilities.
- The main drawback of the thin client architecture is in highly interactive graphical activities such as CAD and image processing, where the delays experienced by users.

Architectural Models

Associated Middleware Solutions:

• The task of middleware is to provide a higher-level programming abstraction for the development of DS.

• And through layering, to abstract over heterogeneity in the underlying infrastructure to promote interoperability and portability.

• Middleware solutions are based on the architectural models and also support more complex architectural patterns.

Fundamental Models

- All the previous models of systems share some fundamental properties.
- In particular, all of them are composed of processes that communicate with one another by sending messages over a computer network.
- All of the models share the design requirements of achieving the performance and reliability characteristics of processes and networks.
- And also by ensuring the security of the resources in the system.
- Fundamental models are based on the fundamental properties that allow us to be more specific about their characteristics and the failures and security risks they might exhibit.
- In general, such a fundamental model should contain only the essential ingredients that we need to consider in order to understand and reason about some aspects of a system's behavior.

Fundamental Models

• The purpose of such a model is:

✓ To make explicit all the relevant assumptions about the systems we are modeling.

✓ To make generalizations concerning what is possible or impossible, given those assumptions.

• The aspects of distributed systems that we wish to capture in our fundamental models are intended to help us to discuss and reason about:

✓ **Interaction**:

• Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes.

Fundamental Models

- In the analysis and design of distributed systems we are concerned especially with these interactions.
- The interaction model must reflect the facts that communication takes place with delays.
- That are often of considerable duration, and that the accuracy with which independent processes can be coordinated is limited by these delays.
- And by the difficulty of maintaining the same notion of time across all the computers in a DS.

Fundamental Models

✓ **Failure**:

• The correct operation of a DS is threatened whenever a fault occurs in any of the computers on which it runs (including software faults) or in the network that connects them.

• Our model defines and classifies the faults.

• This provides a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.

✓ **Security**:

• The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents.

• Our security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

Interaction Model

• System Architectures that fundamentally distributed systems are composed of many processes, interacting in complex ways, e.g. :

✓ Multiple server processes may cooperate with one another to provide a service; the examples mentioned above were the Domain Name System, which partitions and replicates its data at servers throughout the Internet.

✓ A set of peer processes may cooperate with one another to achieve a common goal: for example, a voice conferencing system that distributes streams of audio data in a similar manner, but with strict real-time constraints.

• A normal program is executed as a single process.

• DS composed of multiple processes such as those which are more complex.

• Their behavior and state can be described by a *distributed algorithm – a definition of the steps to be taken by each of the* processes of which the system is composed.

Interaction Model

- Those including the transmission of messages between them.
- Messages are transmitted between processes to transfer information between them and to coordinate their activity.
- The rate at which each process proceeds and the timing of the transmission of messages between them cannot in general be predicted.
- It is also difficult to describe all the states of a distributed algorithm, because it must deal with the failures of one or more of the processes involved or the failure of message transmissions.
- Interacting processes perform all of the activity in a distributed system.
- Each process has its own state, consisting of the set of data that it can access and update, including the variables in its program.
- The state belonging to each process is completely private – that is, it cannot be accessed or updated by any other process.

Interaction Model

• There are some factors affecting interacting processes in a DS:

✓ Performance of Communication Channels: Bandwidth, Jitter, Latency, Load on OS.

✓ Computer Clocks and Timing Events: Reference Clock, Clock Drift, Auto correct by GPS.

✓ Two Variants of the Interaction Model: Synchronous DS, Asynchronous DS.

✓ Event Ordering: Avoid concurrency.

Failure Model

- In a DS both processes and communication channels may fail.
- That is, they may depart from what is considered to be correct or desirable behavior.
- The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures.
- According to the research of Hadzilacos and Toueg provide a taxonomy that distinguishes between the failures of processes and communication channels.
- These are presented under the headings omission failures, arbitrary failures and timing failures.

✓ Omission failures :

- The faults classified as omission failures refer to cases when a process or communication channel fails to perform actions that it is supposed to do.

Failure Model

✓ Arbitrary failures :

• The term arbitrary failure is used to describe the worst possible failure semantics, in which any type of error may occur.

• E.g., a process may set wrong values in its data items, or it may return a wrong value in response to an invocation.

• An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps.

• Arbitrary failures in processes cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply.

Failure Model

✓ Timing failures :

• Timing failures are applicable in synchronous DS where time limits are set on process execution time, message delivery time and clock drift rate.

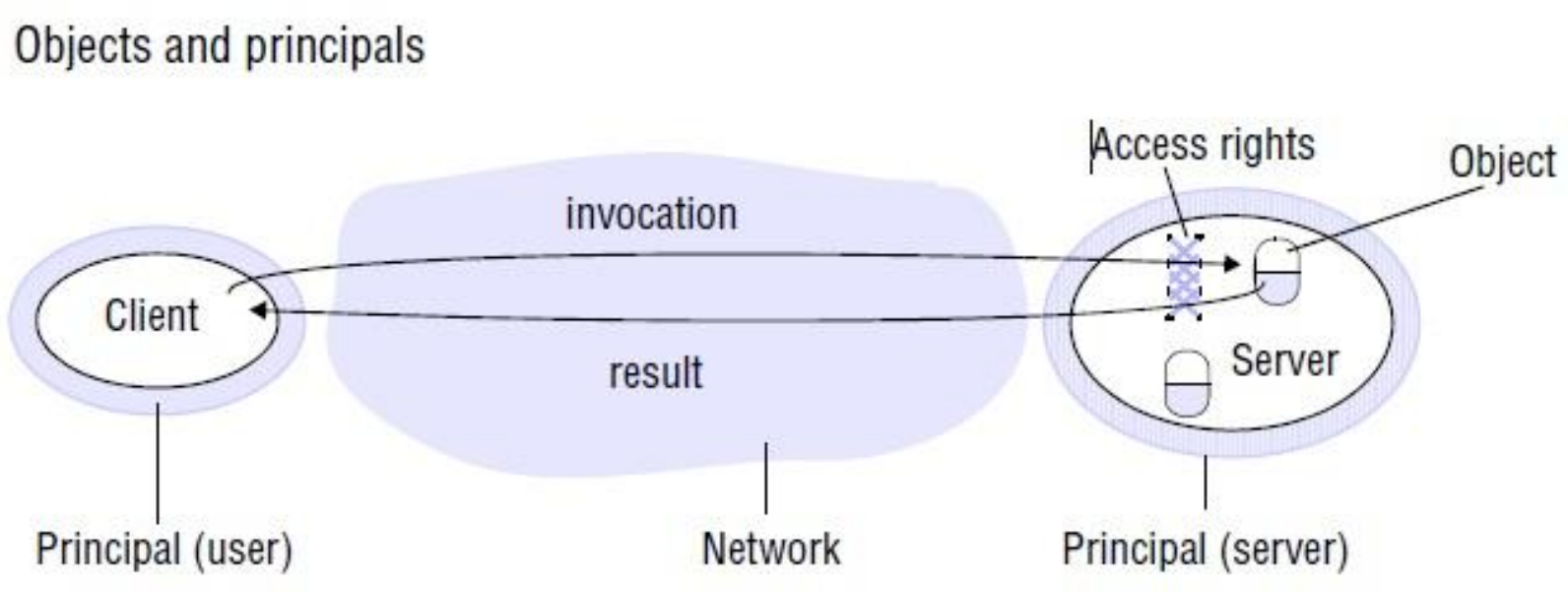
• Any of the failures may result in responses being unavailable to clients within a specified time interval.

• In an asynchronous DS, an overloaded server may respond too slowly, but we cannot say that it has a timing failure since no guarantee has been offered.

Security Model

• The security of a DS can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

• Protection is described in terms of objects.

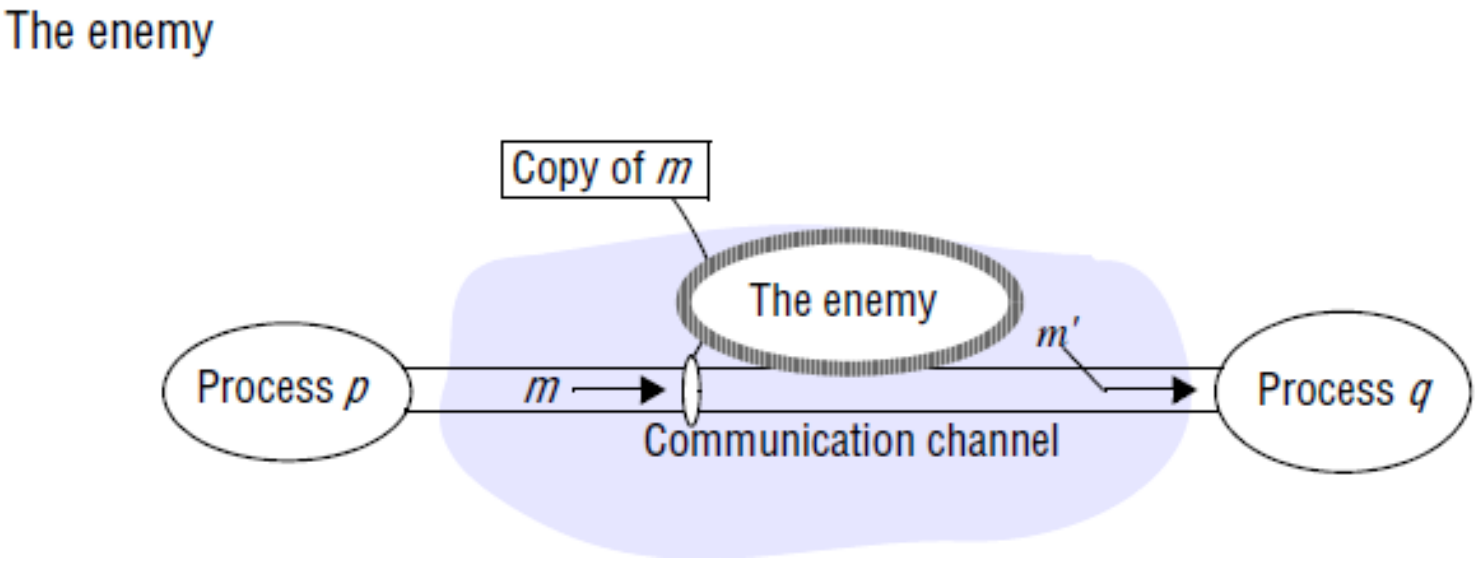


Security Model

✓ Protecting objects: Access Rights, Verification and Validation Process

✓ Securing processes and their interactions: Openness of N/W & Communication channels, exposed interface by servers.

✓ The Enemy: Threats by Enemy,



Security Model

Defeating security threats :

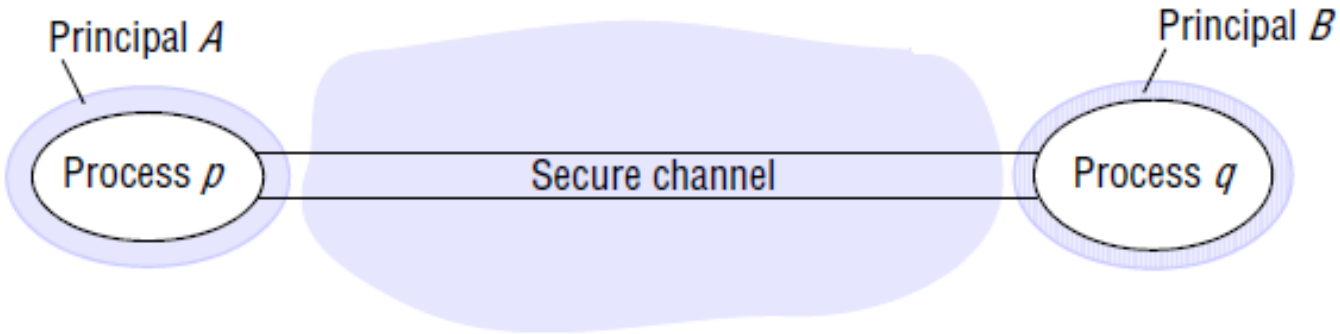
• Here we introduce the main techniques on which secure systems are based.

✓ Cryptography and shared secrets: share a secret code, public & private key.

✓ Authentication: File or message encryption/decryption, observation of portion of encrypted message to guarantee of authenticity.

✓Secure Channel: Encryption and Authentication used to build Secure Channel. VPN, SSL. Ensures Privacy and Integrity i.e. Protection against tampering.

Secure channels



Security Model

Other Possible threats :

• Denial of Service attacks and the deployment of Mobile Code are the possible opportunities for the enemy to disturb the activities of processes.

✓ **Denial of Service Attacks:**

• This is a form of attack in which the enemy interferes with the activities of authorized users.

• By making excessive and pointless invocations on services or message transmissions in a network, resulting in overloading of physical resources like N/w bandwidth, server processing capacity etc.

• Such attacks are usually made with the intention of delaying or preventing actions by other users.

Security Model

• E.g. The operation of electronic door locks in a building might be disabled by an attack that saturates the computer controlling the electronic locks with invalid requests.

✓ **Mobile Code:**

• Mobile code raises new and interesting security problems for any process that receives and executes program code from elsewhere, such as the email attachment.

• Such code may easily play a Trojan horse role.

• These Trojans though claim for innocent purpose, but in fact including code that accesses or modifies resources.

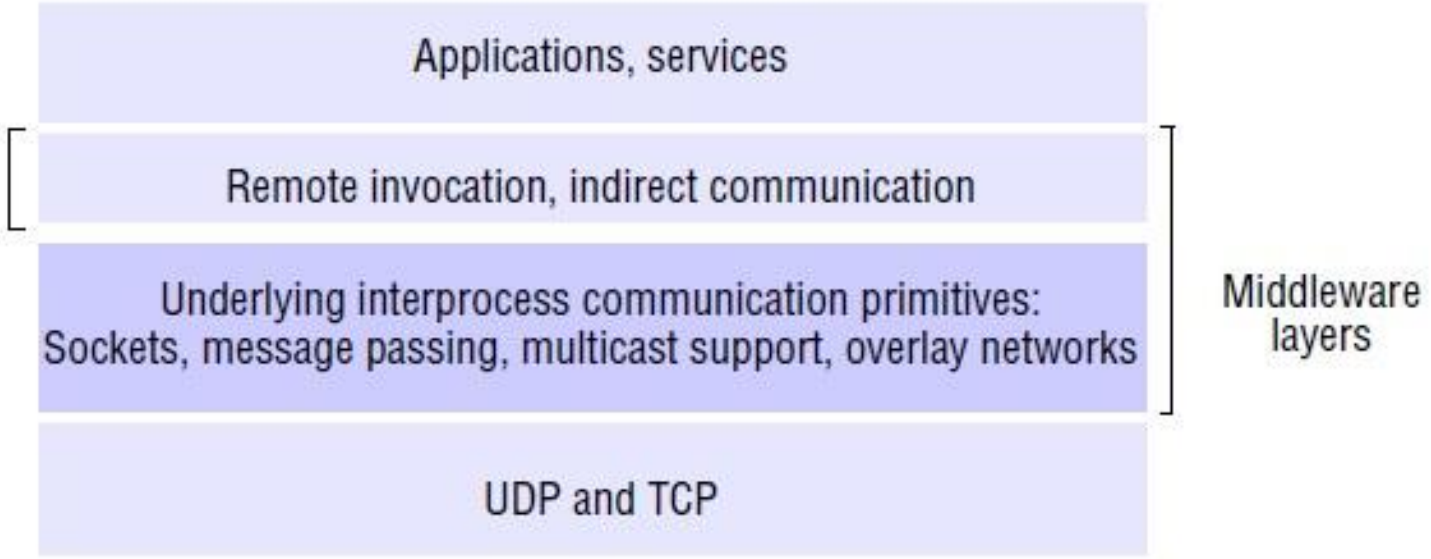
• The methods by which such attacks might be carried out are many and varied, and the host environment must be very carefully constructed in order to avoid them.

Unit – I

Remote Invocation

Remote Invocation

• This topic is concerned with how processes or entities at a higher level of abstraction such as objects or services communicate in a DS.



• The diagram showing the study of Middleware Concepts, by focusing on the layer above Interprocess Communication.

• This is also showing different styles of communication happens in DS.

Remote Invocation

• **Request-reply protocols** represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing.

• In particular, such protocols provide relatively low-level support for requesting the execution of a remote operation.

• The earliest and perhaps the best-known example of a more programmer-friendly model was the extension of the conventional procedure call model to DS ie. **Remote Procedure Call**.

• RPC allows client programs to call procedures transparently in server programs running in separate processes and generally in different computers from the client.

Remote Invocation

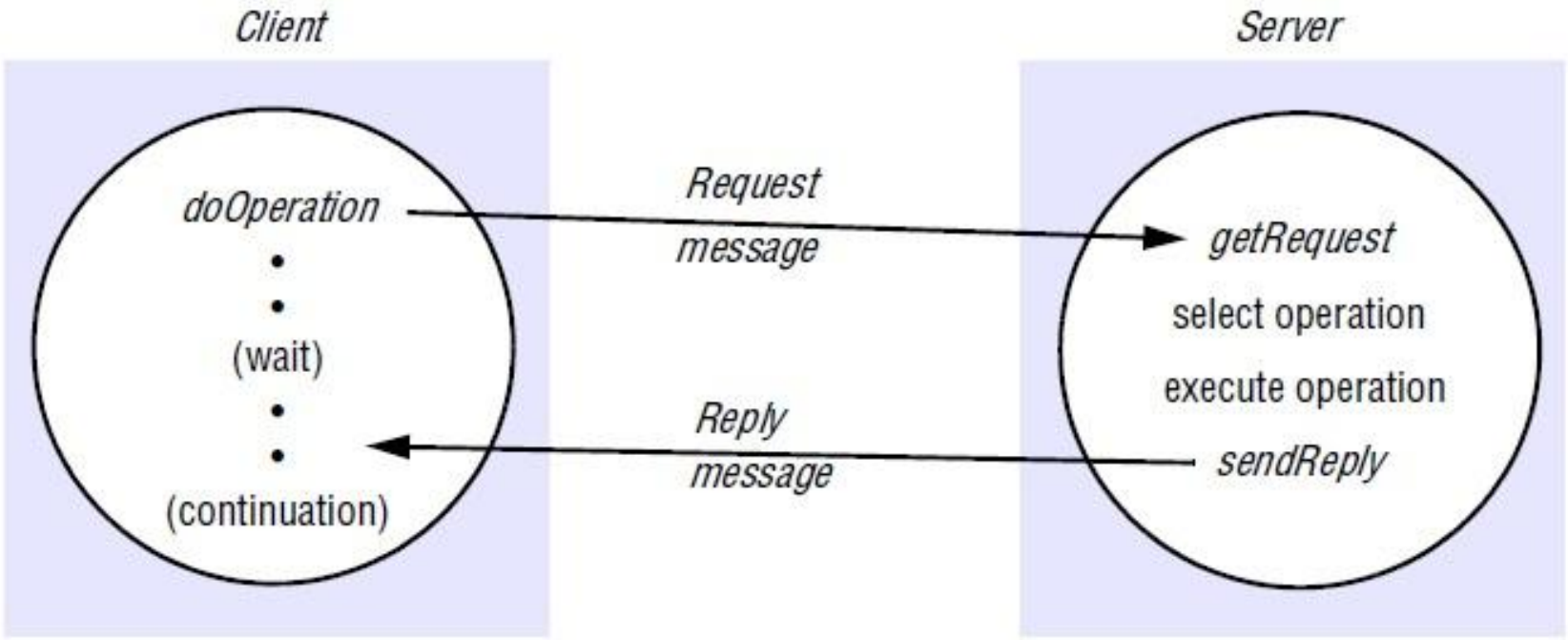
- In the 1990s, the object-based programming model was extended to allow objects in different processes to communicate with one another by means of **Remote Method Invocation.**
- RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.

Request – Reply Protocols

- This form of communication is designed to support the roles and message exchanges in typical client-server interactions.
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- It can also be reliable because the reply from the server is effectively an acknowledgement to the client.
- Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later.
- The client-server exchanges are in terms of the send and receive operations.
- The protocol we describe here is based on a trio of communication primitives, *doOperation, getRequest* and *sendReply*, as shown in diagram.

Request – Reply Protocols

- Request-reply communication



- This request-reply protocol matches requests to replies.
- It may be designed to provide certain delivery guarantees.

Request – Reply Protocols

- Operations of the request-reply protocol

public byte[ ] doOperation (RemoteRef s, int operationId, byte[ ] arguments)

Sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[ ] getRequest ();

Acquires a client request via the server port.

public void sendReply (byte[ ] reply, InetAddress clientHost, int clientPort);

Sends the reply message *reply* to the client at its Internet address and port.

Request – Reply Protocols

- The *doOperation* method is used by clients to invoke remote operations.
- Its arguments specify the remote server and which operation to invoke, together with additional information required by the operation.
- Its result is a byte array containing the reply.
- The *doOperation* method sends a request message to the server whose Internet address and port are specified in the remote reference given as an argument.
- After sending the request message, *doOperation* invokes receive to get a reply message, from which it extracts the result and returns it to the caller.
- The caller of *doOperation* is blocked until the server performs the requested operation
- Then transmits a reply message to the client process.

Request – Reply Protocols

• *getRequest* is used by a server process to acquire service requests.

• When the server has invoked the specified operation, it then uses *sendReply* to send the reply message to the client.

• When the reply message is received by the client the original *doOperation* is unblocked and execution of the client program continues.

• The information to be transmitted in a request message or a reply message is shown in diagram,

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

Request – Reply Protocols

• The first field indicates whether the message is a Request or a Reply message.

• The second field, *requestId*, contains a message identifier.

• A *doOperation* in the client generates a *requestId* for each request message, and the server copies these IDs into the corresponding reply messages.

• This enables *doOperation* to check that a reply message is the result of the current request, not a delayed earlier call.

• The third field is a remote reference.

• The fourth field is an identifier for the operation to be invoked.

•The fifth field is argument, formed with a array of bytes, which contain actual message is to be passed.

Request – Reply Protocols

• Request – Reply protocols work very well over the TCP. Very less chance of failure in the communication.

• If the three primitives *doOperation, getRequest and sendReply* are implemented over UDP datagrams, then they suffer from the same communication failures. That is:

- ✓ They suffer from omission failures.
- ✓ Messages are not guaranteed to be delivered in sender order.

• In addition, the protocol can suffer from the failure of processes.

• We assume that processes have crash failures.

• To allow for occasions when a server has failed or a request or reply message is dropped, *doOperation* uses a timeout when it is waiting to get the server's reply message.

• The action taken when a timeout occurs depends upon the delivery guarantees being offered.

Request – Reply Protocols

• Failures can be avoided by using some of these techniques,

Timeouts:

• There are various options as to what *doOperation* can do after a timeout.

• The simplest option is to return immediately from *doOperation* with an indication to the client that the *doOperation* has failed.

• The timeout may have been due to the request or reply message getting lost and in the latter case, the operation will have been performed.

• To compensate for the possibility of lost messages, *doOperation* sends the request message repeatedly until either it gets a reply or it i*s* reasonably sure that the delay is due to lack of response from the server rather than to lost messages.

Request – Reply Protocols

Discarding Duplicate Request Messages:

- In cases when the request message is retransmitted, the server may receive it more than once.
- This can lead to the server executing an operation more than once for the same request.
- To avoid this, the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter out duplicates.
- If the server has not yet sent the reply, it need take no special action – it will transmit the reply when it has finished executing the operation.

Request – Reply Protocols

Lost Reply Messages:

• If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result, unless it has stored the result of the original execution.

• Some servers can execute their operations more than once and obtain the same results each time.

• An *idempotent operation* is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once.

•A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once.

Request – Reply Protocols

History:

• For servers that require retransmission of replies without re-execution of operations, a history may be used.

• The term 'history' is used to refer to a structure that contains a record of (reply) messages that have been transmitted.

• An entry in a history contains a request identifier, a message and an identifier of the client to which it was sent.

• Its purpose is to allow the server to retransmit reply messages when client processes request them.

Remote Procedure Call

- The concept of a remote procedure call (RPC) represents a major intellectual breakthrough in distributed computing, ie, achieving a high level of distribution transparency.
- This unification is achieved in a very simple manner, by extending the abstraction of a procedure call to distributed environments.
- In RPC, procedures on remote machines can be called as if they are procedures in the local address space.
- The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, passing of the messages.

Design Issues for RPC

• As per design consideration of RPC, there are three issues that are important in understanding this concept.

✓ **The style of programming promoted by RPC – programming with interfaces;**

✓ **The call semantics associated with RPC;**

✓ **The key issue of transparency and how it relates to remote procedure calls.**

Programming with interfaces:

• Most modern programming languages provide a means of organizing a program as a set of modules that can communicate with one another.

• Communication between modules can be by means of procedure calls between modules or by direct access to the variables in another module.

• In order to control the possible interactions between modules, an explicit *interface* is defined for each module.

Design Issues for RPC

• The interface of a module specifies the procedures and the variables that can be accessed
from other modules.

• Modules are implemented so as to hide all the information about them except that which is available through its interface.

• So long as its interface remains the same, the implementation may be changed without affecting the users of the module.

RPC call semantics:

• Request-reply protocols where we showed that *doOperation* can be implemented in different ways to provide different delivery guarantees.

✓ **Retry request message:** Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

Design Issues for RPC

✓ **Duplicate filtering:** Controls when retransmissions are used and whether to filter out duplicate requests at the server.

✓ **Retransmission of results:** Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

• Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker.

• The choices of RPC invocation semantics are defined as follows.

✓ **Maybe semantics:**

• With maybe semantics, the remote procedure call may be executed once or not at all.

Design Issues for RPC

• Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:

Omission failures if the request or result message is lost;

Crash failures when the server containing the remote operation fails.

✓ **At-least-once semantics:**

• With at-least-once semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received.

• At-least-once semantics can be achieved by the retransmission of request messages, which masks the omission failures of the request or result message.

• At-least-once semantics can suffer from the following types of failure:

Crash failures when the server containing the remote procedure fails;

Arbitrary failures – in cases when the request message is retransmitted.

Design Issues for RPC

✓ **At-most-once semantics:**

• With at-most-once semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received.

• In which case the procedure will have been executed either once or not at all.

• At-most-once semantics can be achieved by using all of the fault-tolerance measures.

Transparency:

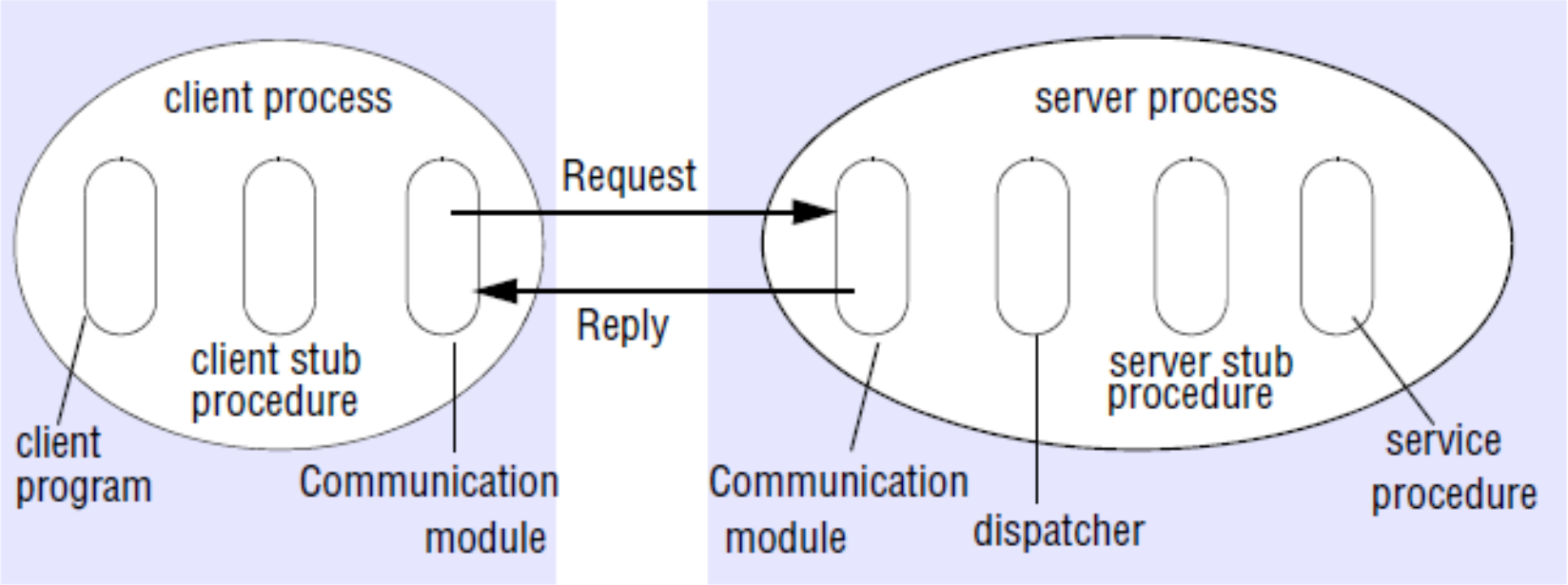
• The originators of RPC, aimed to make remote procedure calls as much like local procedure calls as possible.

• With no distinction in syntax between a local and a remote procedure call.

• All the necessary calls and message-passing procedures were hidden from the programmer making the call.

Implementation of RPC

- Role of client and server stub procedures in RPC



• The software components required to implement RPC are shown in Figure.

• The client that accesses a service includes one stub procedure for each procedure in the service interface.

Implementation of RPC

- The stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server.
- When the reply message arrives, it unmarshals the results.
- The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface.
- The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message.
- The server stub procedure then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message.
- The service procedures implement the procedures in the service interface.
- The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition of the service.

Implementation of RPC

- RPC is generally implemented over a request-reply protocol.
- The contents of request and reply messages are the same as those illustrated for request-reply protocols.
- RPC may be implemented to have one of the choices of invocation semantics discussed in earlier topic.
- At-Least-Once or At-Most-Once is generally chosen.
- To achieve this, the communication module will implement the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results.

Remote Method Invocation

• Remote Method Invocation (RMI) is closely related to RPC but extended into the world of distributed objects.

• In RMI, a calling object can invoke a method in a potentially remote object.

• As with RPC, the underlying details are generally hidden from the user.

• The common things between RMI and RPC are,

✓ They both support programming with interfaces, with the resultant benefits that comes from this approach.

✓ They are both typically constructed on top of request-reply protocols and can offer a range of call semantics such as *at-least-once and at-most-once.*

✓ They both offer a similar level of transparency – that is, local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions.

Remote Method Invocation

- The following differences lead to added expressiveness when it comes to the programming of complex distributed applications and services.

✓ The programmer is able to use the full expressive power of object-oriented programming in the development of DS software, including the use of objects, classes and inheritance, and can also employ related object oriented design methodologies and associated tools.

✓ Building on the concept of object identity in object-oriented systems, all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.

Remote Method Invocation

• The issue of parameter passing is particularly important in DS.

• RMI allows the programmer to pass parameters not only by value, as input or output parameters, but also by object reference.

• The remote end, on receiving an object reference, can then access this object using remote method invocation, instead of having to transmit the object value across the network.

Design Issues in RMI

• RMI shares the same design issues as RPC in terms of programming with interfaces, call semantics and level of transparency.

• The key added design issue relates to the object model and achieving the transition from objects to distributed objects.

• As per the RMI there are two object models,

Single Object Model

Distributed Object Model

Single Object Model:

• An object-oriented program consists of a collection of interacting objects, each of which consists of a set of data and a set of methods.

• An object communicates with other objects by invoking their methods, generally passing arguments and receiving results.

• Objects can encapsulate their data and the code of their methods.

Design Issues in RMI

• Some languages, for example Java and C++, allow programmers to define objects whose instance variables can be accessed directly.

• But for use in a distributed object system, an object's data should be accessible only via its methods.

• We need to concentrate on following issues in case of Object Model,

✓ **Object References:**

• Objects can be accessed via object references.

• All Variables and methods hold by an object needs a Object Reference.

✓ **Interface:**

• An interface provides a definition of the signatures of a set of methods without specifying their implementation.

Design Issues in RMI

✓ **Actions:**

• Action in an object-oriented program is initiated by an object invoking a method in another object.

• The receiver executes the appropriate method and then returns control to the invoking object.

• As an invocation can lead to further invocations of methods in other objects, an action is a chain of related method invocations, each of which eventually returns.

✓ **Exceptions:**

• Programs can encounter many sorts of errors and unexpected conditions of varying seriousness.

• During the execution of a method, many different problems may be discovered.

• E.g. inconsistent values in the object's variables, or failures in attempts to read or write to files or network sockets.

Design Issues in RMI

✓ **Garbage Collection:**

• It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed.

Distributed Object Model:

• This model is used as extensions to the object model to make it applicable to distributed objects.

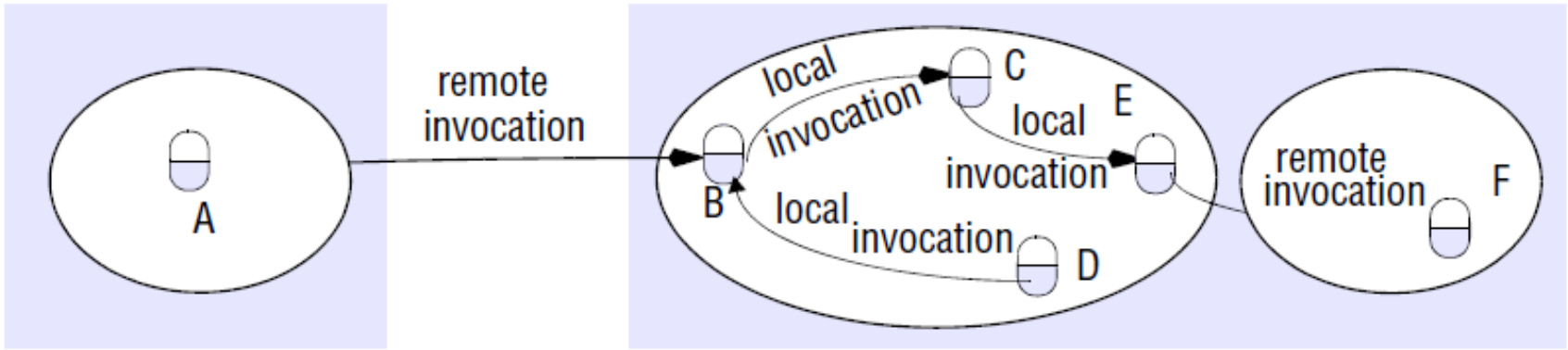
• Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations.

• Method invocations between objects in different processes, whether in the same computer or not, are known as remote method invocations.

• Method invocations between objects in the same process are local method invocations.

Design Issues in RMI

- Remote and local method invocations



- We refer to objects that can receive remote invocations as *remote objects.*
- The objects B and F are remote objects.
- All objects can receive local invocations, although they can receive them only from other objects that hold references to them.
- Object C must have a reference to object E so that it can invoke one of its methods.
- For any Distributed Object Model following two concepts are very important.

Design Issues in RMI

✓ **Remote Object References:**

• Other objects can invoke the methods of a remote object if they have access to its remote object reference.

• E.g. a remote object reference for B in diagram must be available to A.

✓ **Remote Interfaces:**

• Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

• E.g. the objects B and F in diagram must have remote interfaces.

Design Issues in RMI

✓ **Actions in a Distributed System:**

• As in the non-distributed case, an action is initiated by a method invocation, which may result in further invocations on methods in other objects.

• But in the distributed case, the objects involved in a chain of related invocations may be located in different processes or different computers.

• When an invocation crosses the boundary of a process or computer, RMI is used, and the remote reference of the object must be available to the invoker.

• In diagram, object A needs to hold a remote object reference to object B.

• Remote object references may be obtained as the results of remote method invocations.

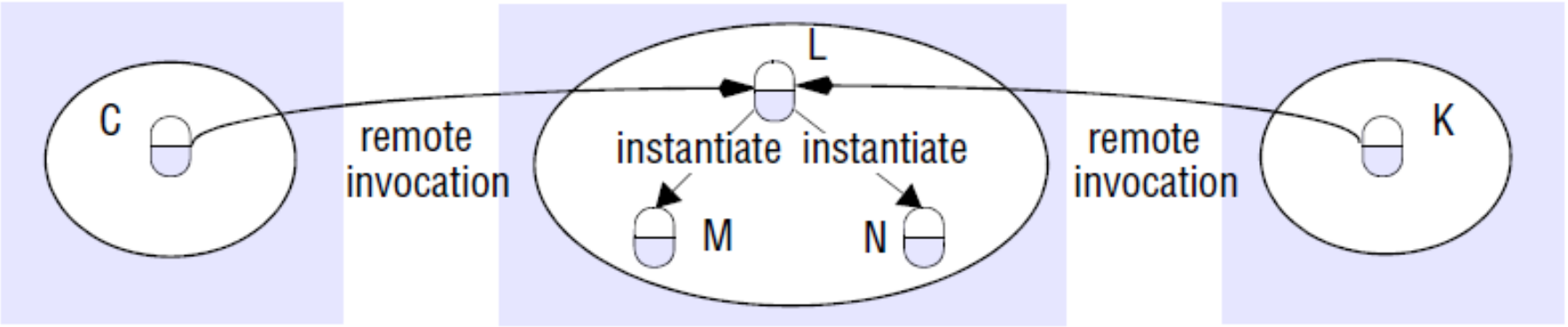
• E.g., object A in diagram might obtain a remote reference to object F from object B.

• When an action leads to the instantiation of a new object, that object will normally live within the process where instantiation is requested where constructor was used.

Design Issues in RMI

• If the newly instantiated object has a remote interface, it will be a remote object with a remote object reference.

Instantiation of remote objects



• Distributed applications may provide remote objects with methods for instantiating objects that can be accessed by RMI, thus effectively providing the effect of remote instantiation of objects.

• E.g., if the object L in diagram contains a method for creating remote objects, then the remote invocations from C and K could lead to the instantiation of the objects M and N, respectively.

Design Issues in RMI

✓ **Garbage Collection in a Distributed-Object System:**

• If a language, e.g. Java, supports garbage collection, then any associated RMI system should allow garbage collection of remote objects.

• Distributed garbage collection is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting.

• If garbage collection is not available, then remote objects that are no longer required should be deleted.

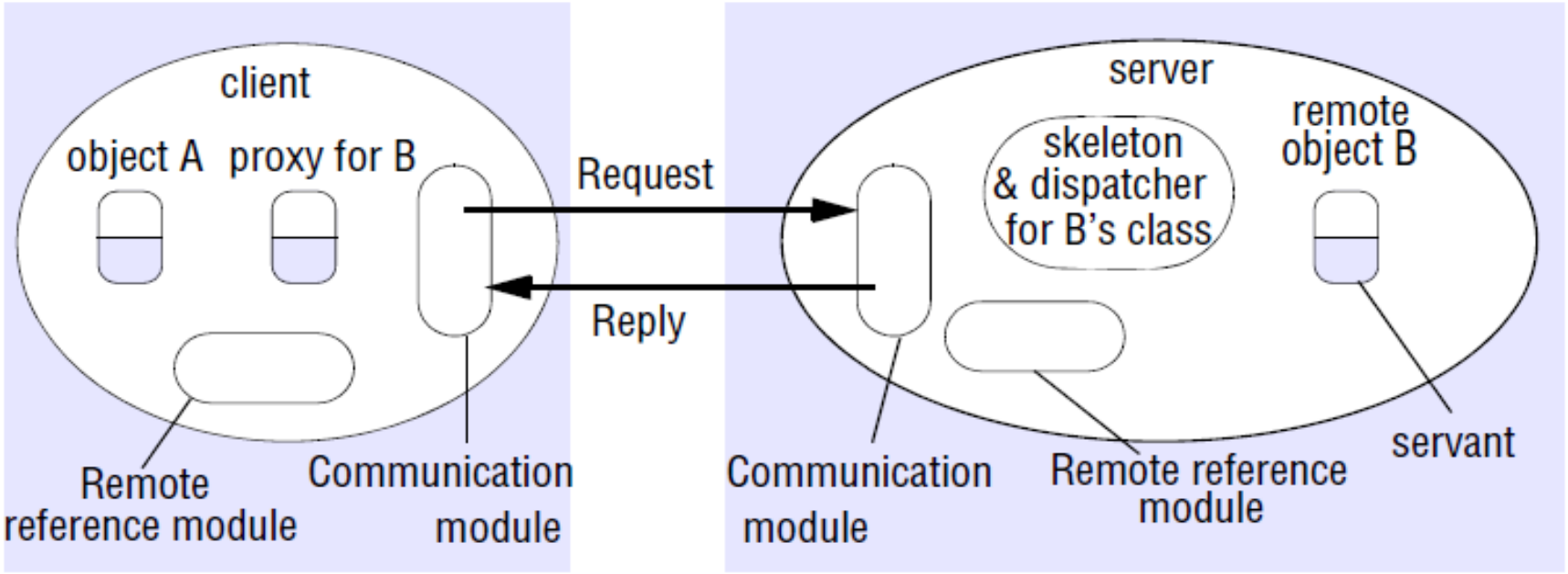
✓ **Exceptions:**

•Any remote invocation may fail for reasons related to the invoked object being in a different process or computer from the invoker.

• E.g., the process containing the remote object may have crashed or may be too busy to reply, or the invocation or result message may be lost.

Implementation of RMI

The role of proxy and skeleton in remote method invocation



• Several separate objects and modules are involved in achieving a remote method invocation.

• These are shown in diagram, in which an application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference.

Implementation of RMI

• This section discusses the roles of each of the components shown in that figure, dealing first with the communication and remote reference modules and then with the RMI software that runs over them.

Communication Module:

• The two cooperating communication modules carry out the *request-reply protocol*, which transmits *request* and *reply* messages between the client and server.

• The communication module uses only the first three items, which specify the message type, its *requestId* and the remote reference of the object to be invoked.

• The *operationId* and all the *marshalling* and *unmarshalling* are the concern of the RMI software.

• The communication modules are together responsible for providing a specified invocation semantics, e.g. *at-most-once.*

Implementation of RMI

• The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on its local reference, which it gets from the remote reference module in return for the remote object identifier in the *request message.*

Remote Reference Module:

• A remote reference module is responsible for translating between local and remote object references and for creating remote object references.

• To support its responsibilities, the remote reference module in each process has a *remote object table* that records the correspondence between local object references in that process and remote object references.

• This module is called by components of the RMI software when they are *marshalling* and *unmarshalling* remote object references.

• E.g., when a *request* message arrives, the table is used to find out which local object is to be invoked.

Implementation of RMI

- Remote Object Table generally contains,
  - ✓ An entry for all the remote objects held by the process.
  - ✓ An entry for each local proxy.
  - ✓ When a remote object is to be passed as an argument or a result for the first time, the remote reference module is asked to create a remote object reference, which it adds to its table.
  - ✓ When a remote object reference arrives in a *request or reply* message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object. In the case that the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.

Implementation of RMI

Servants:

• A servant is an instance of a class that provides the body of a remote object.

• It is the servant that eventually handles the remote requests passed on by the corresponding skeleton.

• Servants live within a server process.

• They are created when remote objects are instantiated and remain in use until they are no longer needed, finally being garbage collected or deleted.

The RMI Software:

• This consists of a layer of software between the application-level objects and the communication and remote reference modules.

• It has different roles working as a middleware objects, ie., Proxy, Dispatcher & Skeleton.

Implementation of RMI

Proxy:

• The role of a proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker;

• But instead of executing an invocation, it forwards it in a message to a remote object.

• It hides the details of the remote object reference, the marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client.

• There is one proxy for each remote object for which a process holds a remote object reference.

Dispatcher:

• A server has one dispatcher and one skeleton for each class representing a remote object.

• E.g., the server has a dispatcher and a skeleton for the class of remote object B.

Implementation of RMI

• The dispatcher receives request messages from the communication module.

• It uses the *operationId* to select the appropriate method in the skeleton, passing on the request message.

• The dispatcher and the proxy use the same allocation of *operationIds* to the methods of the remote interface.

Skeleton:

• The class of a remote object has a skeleton, which implements the methods in the remote interface.

• They are implemented quite differently from the methods in the servant that incarnates a remote object.

• A skeleton method *unmarshals* the arguments in the *request* message and invokes the corresponding method in the servant.

• It waits for the invocation to complete and then *marshals* the result, together with any exceptions, in a *reply* message to the sending proxy's method.

Implementation of RMI

Server and Client Program:

• The server program contains the classes for the dispatchers and skeletons.

• Both together with the implementations of the classes of all of the servants that it supports.

• The server program contains an *initialization* section e.g. a *main method in Java or C++.*

• *The initialization section is* responsible for creating and initializing at least one of the servants to be hosted by the server.

• The client program will contain the classes of the proxies for all of the remote objects that it will invoke.

• It can use a binder to look up remote object references.

• A *binder in a* DS is a separate service that maintains a table containing mappings from textual names to remote object references.

Case Study

JAVA RMI

Java RMI

- Java RMI extends the Java object model to provide support for distributed objects in the Java language.
- It allows objects to invoke methods on remote objects using the same syntax as for local invocations.
- The programming of distributed applications in Java RMI should be relatively simple because it is a single-language system.
- Remote interfaces are defined in the Java language.
- If a multiple-language system such as CORBA is used, the programmer needs to learn an IDL (Interface Definition Language) and to understand how it maps onto the implementation language.
- In this case study, we are discussing the CORBA, Web Services and we use a Shared Whiteboard as an example.

Java RMI

- This is a distributed program that allows a group of users to share a common view of a drawing surface containing graphical objects, such as rectangles, lines and circles, each of which has been drawn by one of the users.
- The server maintains the current state of a drawing by providing an operation for clients to inform it about the latest shape one of their users has drawn and keeping a record of all the shapes it has received.
- The server also provides operations allowing clients to retrieve the latest shapes drawn by other users by polling the server.
- The server has a version number (an integer counter) that it increments each time a new shape arrives and attaches to the new shape.
- The server provides operations allowing clients to enquire about its version number and the version number of each shape, so that they may avoid fetching shapes that they already have.

Remote Interfaces in Java

- Remote interfaces are defined by extending an interface called *Remote* provided in the *java.rmi package.*
- The methods must throw *RemoteException.*

Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Remote Interfaces in Java

• Above program snippet shows an example of two remote interfaces called *Shape and ShapeList.*

• In this example, *GraphicalObject* is a class that holds the state of a graphical object.

• E.g., its type, its position, enclosing rectangle, line color and fill color – and provides operations for accessing and updating its state.

Parameter and Result Passing

• In Java RMI, the parameters of a method are assumed to be *input* parameters and the result of a method is a single *output* parameter.

• Java serialization, which is used for marshalling arguments and results in Java RMI.

• Any object that is serializable – that is, that implements the *Serializable* interface – can be passed as an argument or result in Java RMI.

• All primitive types and remote objects are serializable.

• Thus, in our example, the client uses the method *newShape* to pass an instance of *GraphicalObject* to the server; the server makes a remote object of type *Shape* containing the state of the *GraphicalObject* and returns a remote object reference to it.

• The arguments and return values in a remote invocation are serialized to a stream using the method with the following modifications.

Parameter and Result Passing

1. Whenever an object that implements the *Remote interface is serialized, it is* replaced by its remote object reference, which contains the name of its (the remote object's) class.
2. When any object is serialized, its class information is annotated with the location of the class (as a URL), enabling the class to be downloaded by the receiver.

Downloading of Classes

- Java is designed to allow classes to be downloaded from one virtual machine to another.
- This is particularly relevant to distributed objects that communicate by means of remote invocation.
- The non-remote objects are passed by value and remote objects are passed by reference as arguments and results of RMIs.
- If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically.
- Similarly, if the recipient of a remote object reference does not already possess the class for a proxy, its code is downloaded automatically.

- As an example, consider the *whiteboard program* and suppose that its initial implementation of *GraphicalObject* does not allow for text.

Downloading of Classes

• A client with a textual object can implement a subclass of *GraphicalObject* that deals with text and pass an instance to the server as an argument of the *newShape method*.

• After that, other clients may retrieve the instance using the *getAllState method*.

• The code of the new class will be downloaded automatically from the first client to the server and then to other clients as needed.

RMI-Registry

- The RMIregistry is the binder for Java RMI.
- An instance of RMIregistry should normally run on every server computer that hosts remote objects.
- It maintains a table mapping textual, URL-style names to references to remote objects hosted on that computer.
- It is accessed by methods of the *Naming class, whose methods take as an* argument a URL-formatted string of the form,

// computerName:port/objectName

- Where *computerName and port refer to the location of the RMIregistry. If they are* omitted, the local computer and default port are assumed.

Building Client and Server Programs

• This basically outlines the steps necessary to produce client and server programs that use the *Remote* interfaces *Shape and ShapeList.*

• The server program is a simplified version of a whiteboard server that implements the two interfaces *Shape and ShapeList.*

• We consider a simple polling client program and then introduce the callback technique that can be used to avoid the need to poll the server.

Server Program:

• The server is a whiteboard server: it represents each shape as a remote object instantiated by a servant that implements the *Shape* interface and holds the state of a graphical object as well as its version number.

• It represents its collection of shapes by using another servant that implements the *ShapeList* interface and holds a collection of shapes in a *Vector.*

Building Client and Server Programs

• The server program consists of a *main method and a servant class* to implement each of its remote interfaces.

Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();                    1
            ShapeList stub =                                                  2
                (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);3
            Naming.rebind("//bruno.ShapeList", stub );                        4
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Building Client and Server Programs

• The *main method* of the server class is shown in above code snippet with the key steps contained in the lines marked 1 to 4.

• In line 1, the server creates an instance of *ShapeListServant.*

• Lines 2 and 3 use the method *exportObject (defined on UnicastRemoteObject)* to make this object available to the RMI runtime, thereby making it available to receive incoming invocations. The second parameter of *exportObject* specifies the TCP port to be used for incoming invocations. It is normal practice to set this to zero, implying that an anonymous port will be used (one that is generated by the RMI runtime). Using *UnicastRemoteObject* ensures that the resultant object lives only as long as the process in which it is created.

• Line 4 binds the remote object to a name in the *RMIregistry*. Note that the value bound to the name is a remote object reference, and its type is the type of its remote interface – *ShapeList.*

Building Client and Server Programs

• The two servant classes are *ShapeListServant*, which implements the *ShapeList* interface, and *ShapeServant*, which implements the *Shape* interface.

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.util.Vector;
public class ShapeListServant implements ShapeList {
     private Vector theList;              // contains the list of Shapes
    private int version;
    public ShapeListServant(){...}
    public Shape newShape(GraphicalObject g)  {
        version++;
        Shape s = new ShapeServant( g, version);
        theList.addElement(s);
        return s;
    }
    public  Vector allShapes(){...}
    public int getVersion()  { ... }
}
```

Building Client and Server Programs

- The implementation of the methods of the remote interface in a servant class is completely straightforward because it can be done without any concern for the details of communication.
- The *main method* of a server needs to create a security manager to enable Java security to apply the protection appropriate for an RMI server.
- A default security manager called *RMISecurityManager* is provided.
- It protects the local resources to ensure that the classes that are loaded from remote sites cannot have any effect on resources such as files.
- But it differs from the standard Java security manager in allowing the program to provide its own class loader and to use reflection.

Building Client and Server Programs

Client Program:

• A simplified client for the *ShapeList* server is illustrated in below code snippet.

Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
      public static void main(String args[]){
            System.setSecurityManager(new RMISecurityManager());
            ShapeList aShapeList = null;
            try{
                  aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");     1
                  Vector sList = aShapeList.allShapes();                            2
            } catch(RemoteException e) {System.out.println(e.getMessage());
            }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
      }
}
```

Building Client and Server Programs

• Any client program needs to get started by using a binder to look up a remote object reference.

• Our client sets a security manager and then looks up a remote object reference for the remote object using the *lookup operation of the RMIregistry (line 1).*

• Having obtained an initial remote object reference, the client continues by sending RMIs to that remote object or to others discovered during its execution according to the needs of its application.

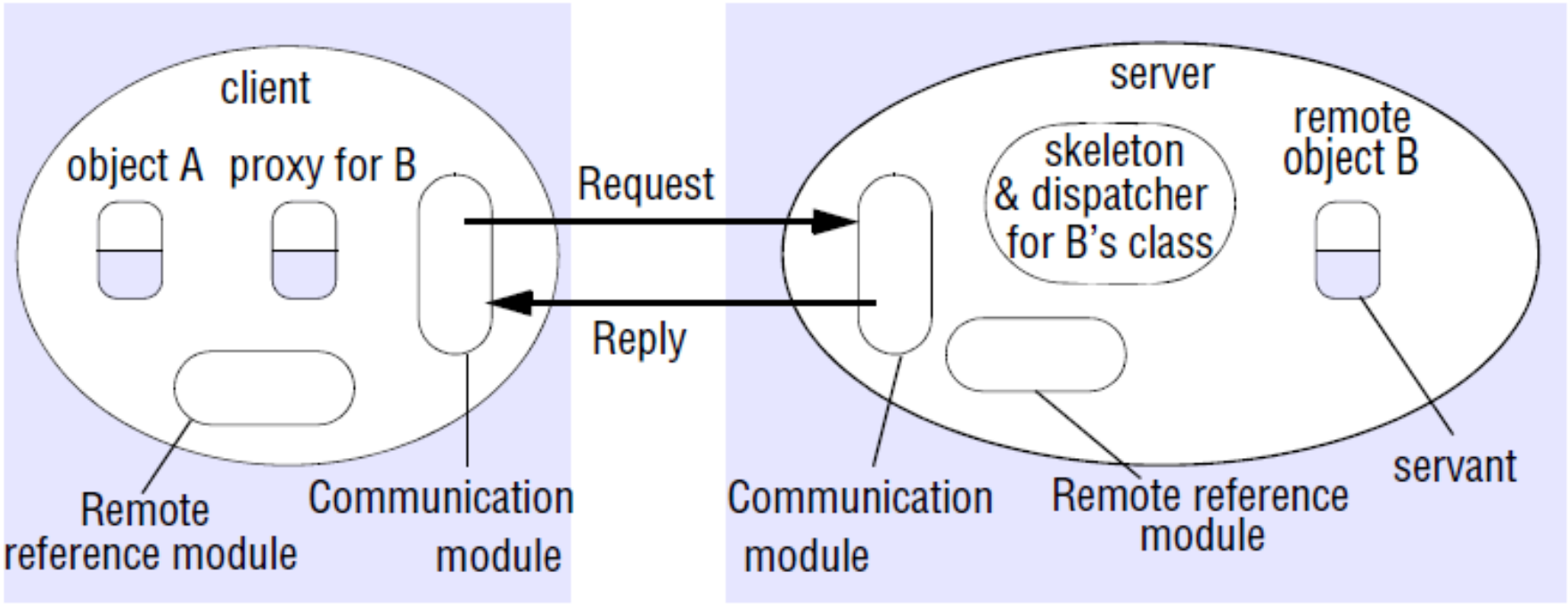
• In our example, the client invokes the method *allShapes* in the remote object (line 2) and receives a vector of remote object references to all of the shapes currently stored in the server.

• If the client was implementing a whiteboard display, it would use the server's *getAllState* method in the *Shape* interface to retrieve each of the graphical objects in the vector and display them in a window.

Building Client and Server Programs

- Each time the user finishes drawing a graphical object, it will invoke the method *newShape* in the server, passing the new graphical object as its argument.
- The client will keep a record of the latest version number at the server, and from time to time it will invoke *getVersion* at the server to find out whether any new shapes have been added by other users.
- If so, it will retrieve and display them.

Design & Implementation of Java RMI



- The original Java RMI system used all of the components shown in diagram.
- But in Java 1.2, the reflection facilities were used to make a generic dispatcher and to avoid the need for skeletons.
- However, this step is no longer necessary with recent versions of J2SE, which contain support for the dynamic generation of stub classes at runtime.

Design & Implementation of Java RMI

Use of Reflection:

• Reflection is used to pass information in *request* messages about the method to be invoked.

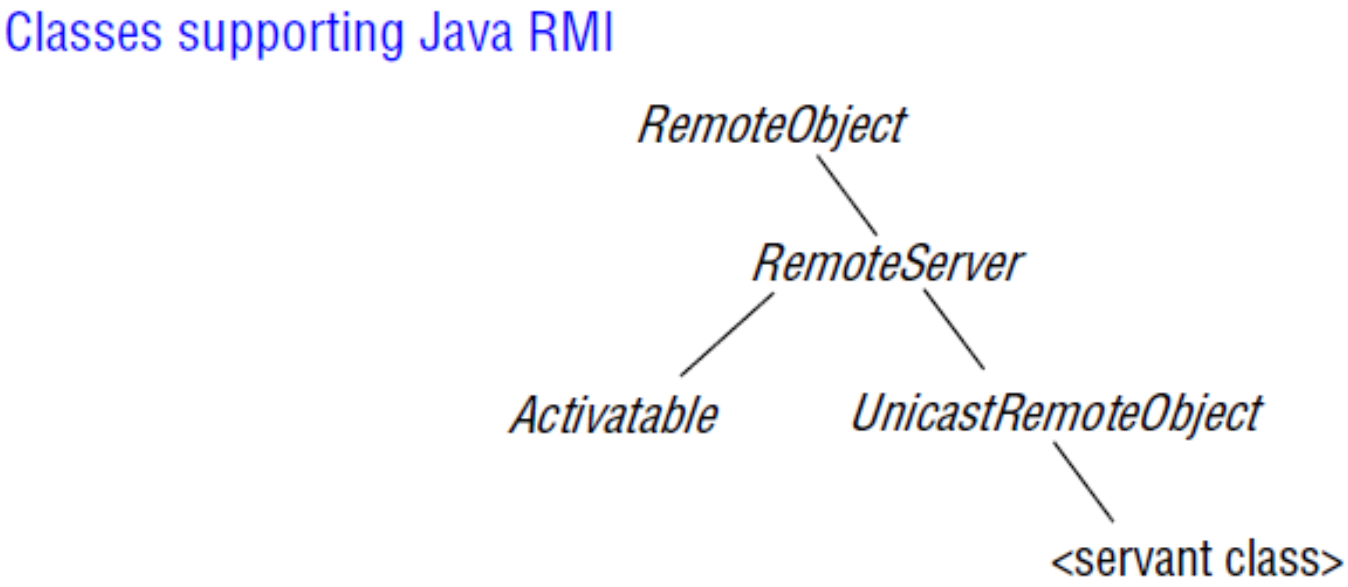
• This is achieved with the help of the class *Method* in the *reflection* package.

• Each instance of *Method* represents the characteristics of a particular method, including its class and the types of its arguments, return value and exceptions.

• The most interesting feature of this class is that an instance of *Method* can be invoked on an object of a suitable class by means of its *invoke method.*

• The *invoke method* requires two arguments: the first specifies the object to receive the invocation and the second is an array of *Object* containing the arguments. The result is returned as type *Object.*

Classes Supporting Java RMI



- Diagram shows the inheritance structure of the classes supporting Java RMI servers.
- The only class that the programmer need be aware of is *UnicastRemoteObject,* which every simple servant class needs to extend.
- The class *UnicastRemoteObject* extends an abstract class called *RemoteServer,* which provides abstract versions of the methods required by remote servers.

Classes Supporting Java RMI

- *UnicastRemoteObject* was the first example of *RemoteServer* to be provided.
- Another called *Activatable* is available for providing *activatable* objects.
- Further alternatives might provide for replicated objects.
- The class *RemoteServer* is a subclass of *RemoteObject* that has an instance variable holding the remote object reference and provides the following methods:

  equals: This method compares remote object references.

  toString: This method gives the contents of the remote object reference as a String.

  readObject, writeObject: These methods deserialize/serialize remote objects.

  the *instanceOf* operator can be used to test remote objects.