Architectural Model of Distributed System (System Architectural Style)

Dr. Risala T Khan

Professor

IIT, JU

System Architectural Style

- System architectural styles cover the physical organization of components and processes over a distributed infrastructure.
- They provide a set of reference models for the deployment of such systems and help engineers identify the major advantages and drawbacks of a given deployment and whether it is applicable for a specific class of applications.
- In this section, we introduce two fundamental reference styles:
  - client/server and
  - peer-to-peer.

Client/Server Model

- This architecture is very popular in distributed computing and is suitable for a wide variety of applications.
- The client/server model features two major components: **a server** and **a client**.
- These two components interact with each other through a network connection using a given protocol.
- The communication is unidirectional: The client issues a request to the server, and after processing the request the server returns a response.
- There could be multiple client components issuing requests to a server that is passively waiting for them.
- Hence, the important operations in the client-server paradigm are request, accept (client side), and listen and response (server side).

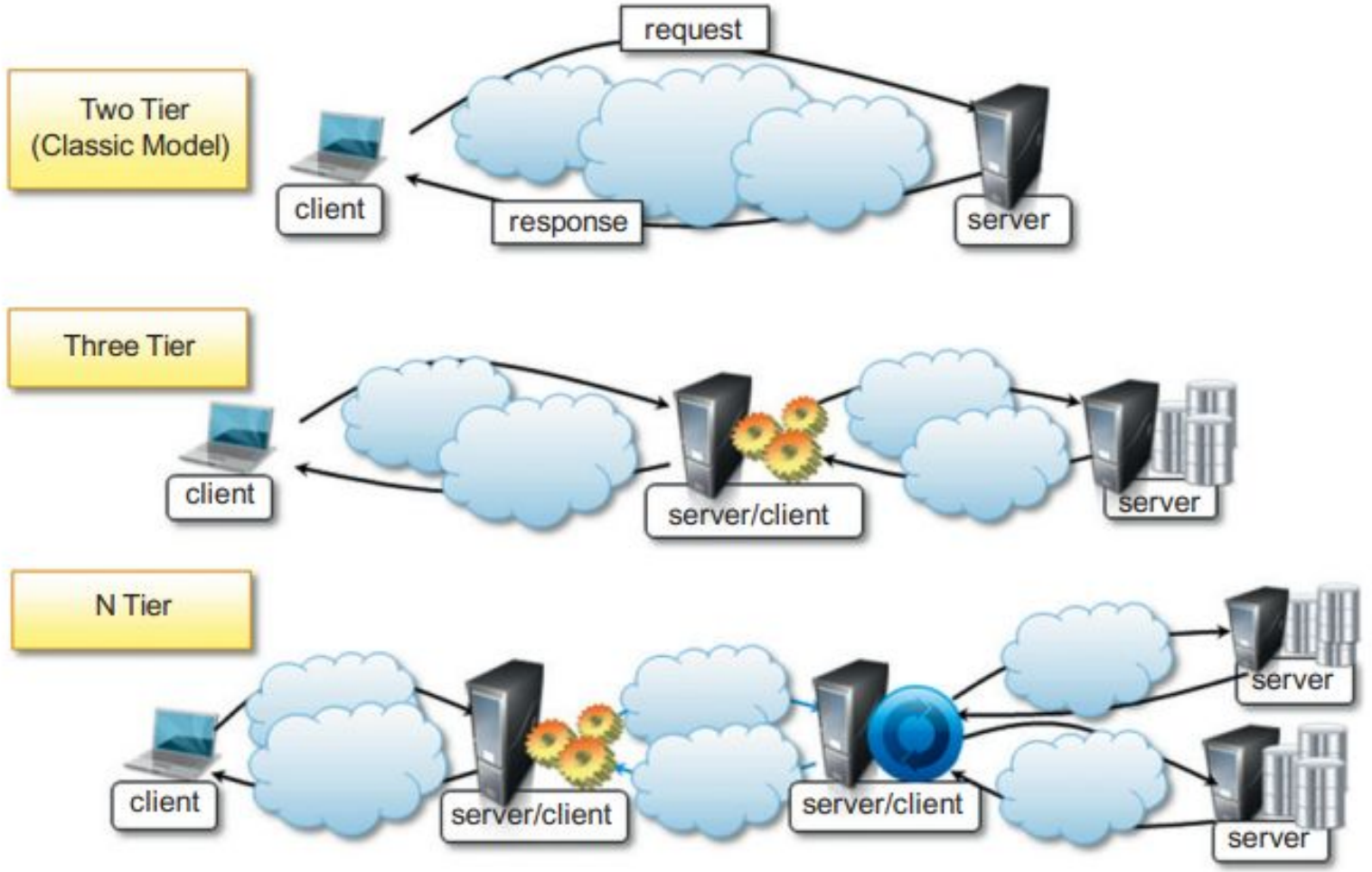


FIGURE 2.12

Client/server architectural styles.

Cont..

- In the client design we identify two major models:
- **Thin-client model:**

  In this model, the load of data processing and transformation is put on the server side, and the client has a light implementation that is mostly concerned with retrieving and returning the data it is being asked for, with no considerable further processing.
- **Fat-client model**
  - In this model, the client component is also responsible for processing and transforming the data before returning it to the user, whereas the server features a relatively light implementation that is mostly concerned with the management of access to the data
- The three major components in the client-server model: **presentation, application logic, and data storage**.
- In the thin-client model, the client embodies only the presentation component, while the server absorbs the other two.
- In the fat-client model, the client encapsulates presentation and most of the application logic, and the server is principally responsible for the data storage and maintenance

Cont…

- Presentation, application logic, and data maintenance can be seen as conceptual layers, which are more appropriately called **tiers**.
- The mapping between the conceptual layers and their physical implementation in modules and components allows differentiating among several types of architectures, which go under the name of multitiered architectures.
- Two major classes exist:
  - Two-tier architecture
  - Three-tier architecture

Two-Tier Architecture

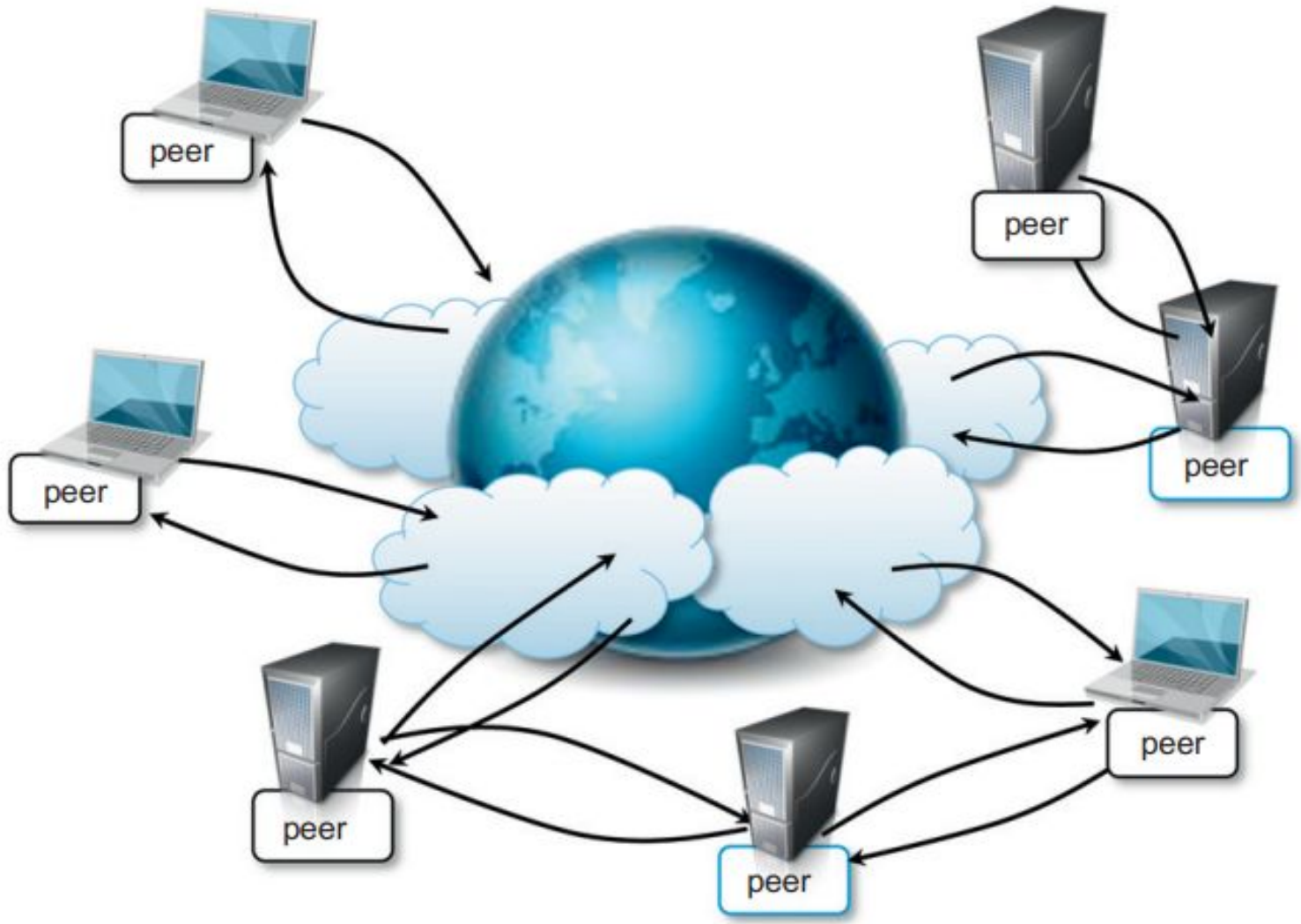
- This architecture partitions the systems into two tiers, which are located one in the client component and the other on the server.
- The client is responsible for the presentation tier by providing a user interface;
- The server concentrates the application logic and the data store into a single tier.
- The server component is generally deployed on a powerful machine that is capable of processing user requests, accessing data, and executing the application logic to provide a client with a response.
- This architecture is suitable for systems of limited size and suffers from scalability issues.
- In particular, as the number of users increases the performance of the server might dramatically decrease.

Three-tier architecture/N-tier architecture:

- The three-tier architecture separates the presentation of data, the application logic, and the data storage into three tiers.
- This architecture is generalized into an N-tier model in case it is necessary to further divide the stages composing the application logic and storage tiers.
- This model is generally more scalable than the two-tier one because it is possible to distribute the tiers into several computing nodes, thus isolating the performance bottlenecks.
- At the same time, these systems are also more complex to understand and manage.
- A classic example of three-tier architecture is constituted by a medium-size Web application that relies on a relational database management system for storing its data.
- In this scenario, the client component is represented by a Web browser that embodies the presentation tier, whereas the application server encapsulates the business logic tier, and a database server machine (possibly replicated for high availability) maintains the data storage.
- Application servers that rely on third-party (or external) services to satisfy client requests are examples of N-tiered architectures.

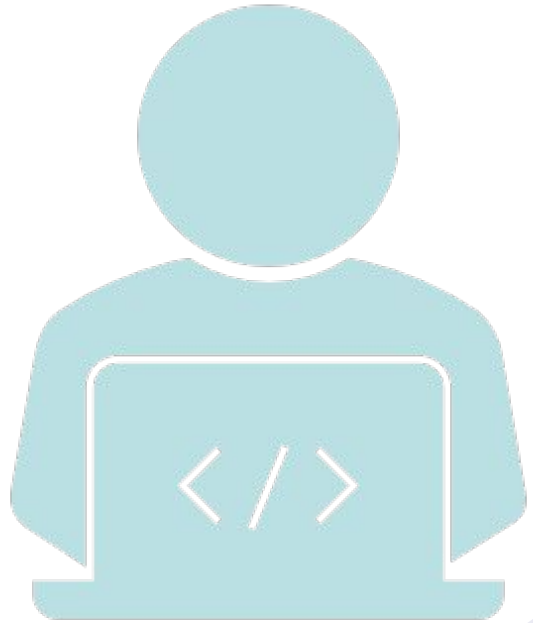
Peer-to-Peer

- The peer-to-peer model, introduces a symmetric architecture in which all the components, called peers, play the same role and incorporate both client and server capabilities of the client/server model.
- More precisely, each peer acts as a server when it processes requests from other peers and as a client when it issues requests to other peers.
- Therefore, this model is quite suitable for highly decentralized architecture, which can scale better along the dimension of the number of peers.
- The disadvantage of this approach is that the management of the implementation of algorithms is more complex than in the client/server model.
- The most relevant example of peer-to-peer systems is constituted by file-sharing applications such as Gnutella, BitTorrent, and Kazaa.
- To address an incredibly large number of peers, different architectures have been designed that divert slightly from the peer-to-peer model.
- For example, in Kazaa not all the peers have the same role, and some of them are used to group the accessibility information of a group of peers.


peer
peer
peer
peer
peer
peer
peer
peer

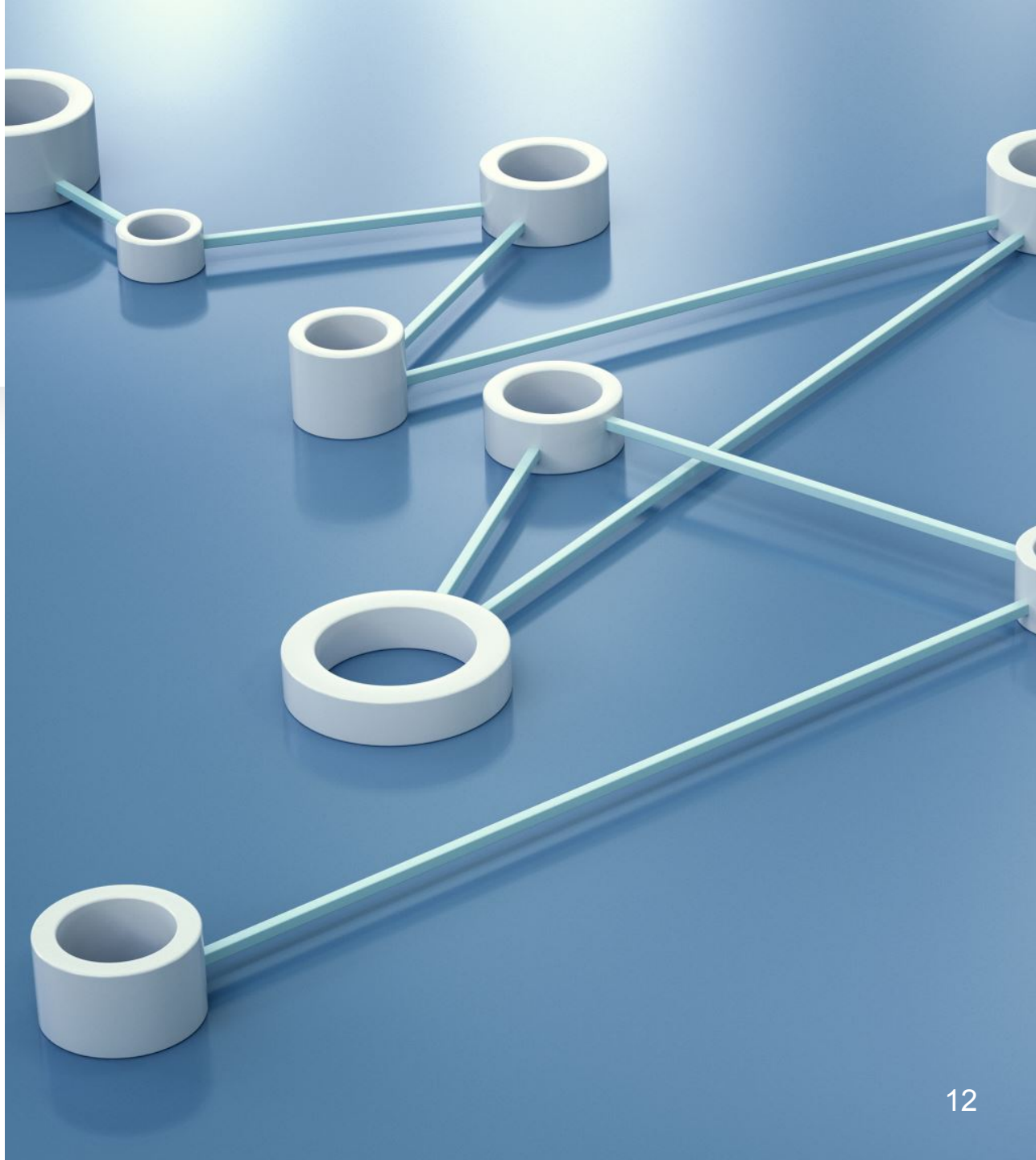
Software Agents

- A software agent is a computer program designed to autonomously perform tasks or functions on behalf of a user or another program.
- These agents are often intelligent, capable of adapting to changes, and can operate without direct human intervention.
- They use inputs from their environment, make decisions, and act to achieve specific goals.



Key Features of Software Agent

1. **Autonomy:** Operates independently to perform tasks.
2. **Reactivity:** Responds to changes in its environment.
3. **Proactiveness:** Initiates actions to achieve predefined goals.
4. **Social Ability:** Can communicate and collaborate with other agents or systems.

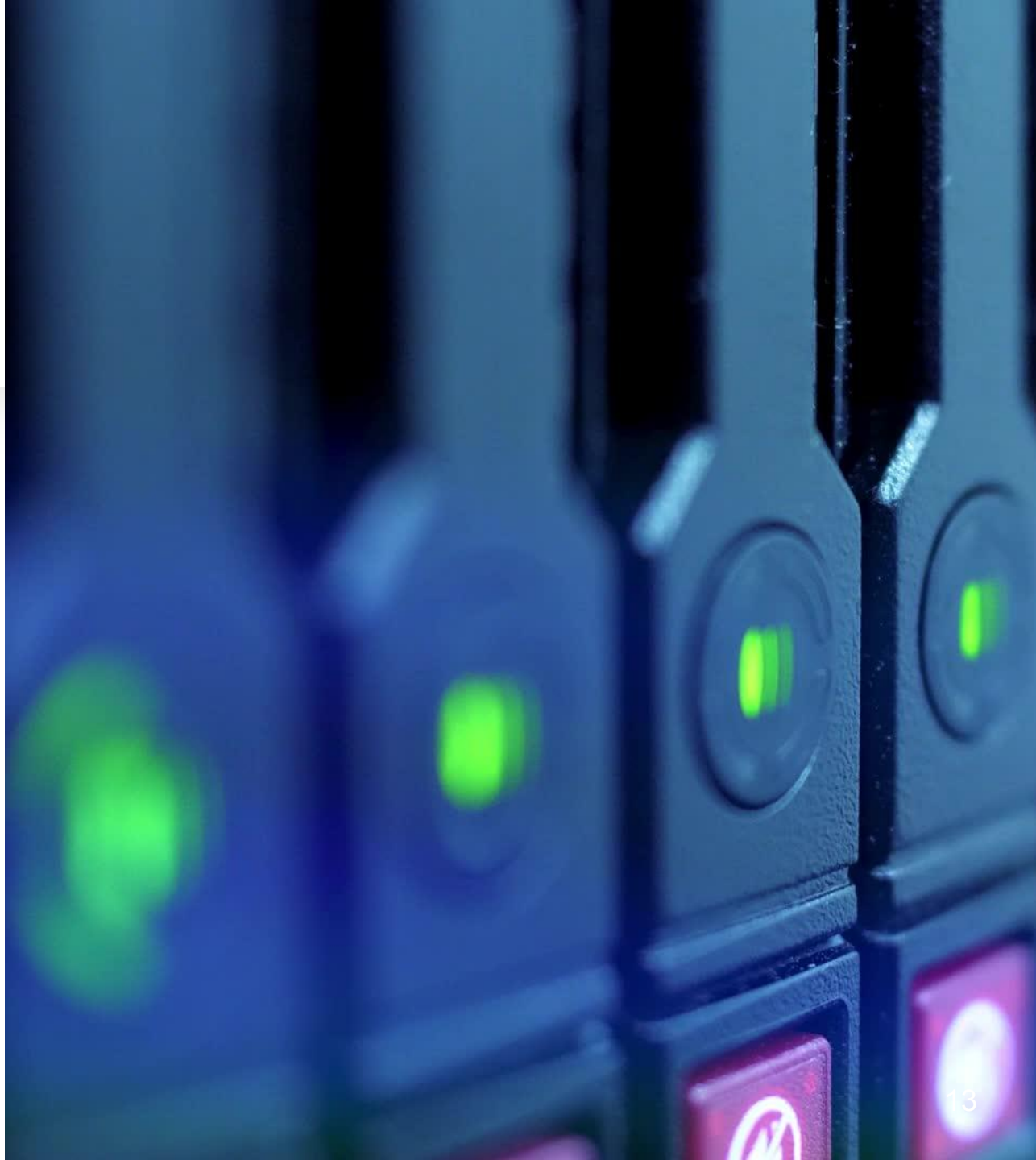


Sensors and Software agent

Sensors themselves are **not software agents**, but they can work in tandem with software agents to gather and process data. Here's the distinction:

For example, in a smart home system:

- Sensors detect room temperature.
- A software agent analyzes the sensor data and decides to adjust the thermostat to optimize comfort and energy use.
- In short, while sensors are vital for providing information, software agents use that information to perform intelligent actions.



Different Types of Agents

- **Collaborative Agent:**
  - Is an agent that forms part of a multiagent system, in which agents seek to achieve some common goal through collaboration.
  - Example: E-commerce systems where agents help buyers find products based on their preferences.

- **Reactive Agent:**
  - These agents act based solely on the current situation and don't retain past experiences.
  - They are simple, fast, and highly specialized.
  - Example: Robots that respond to environmental stimuli like obstacle detection

Different Types of Agents (Continue…)

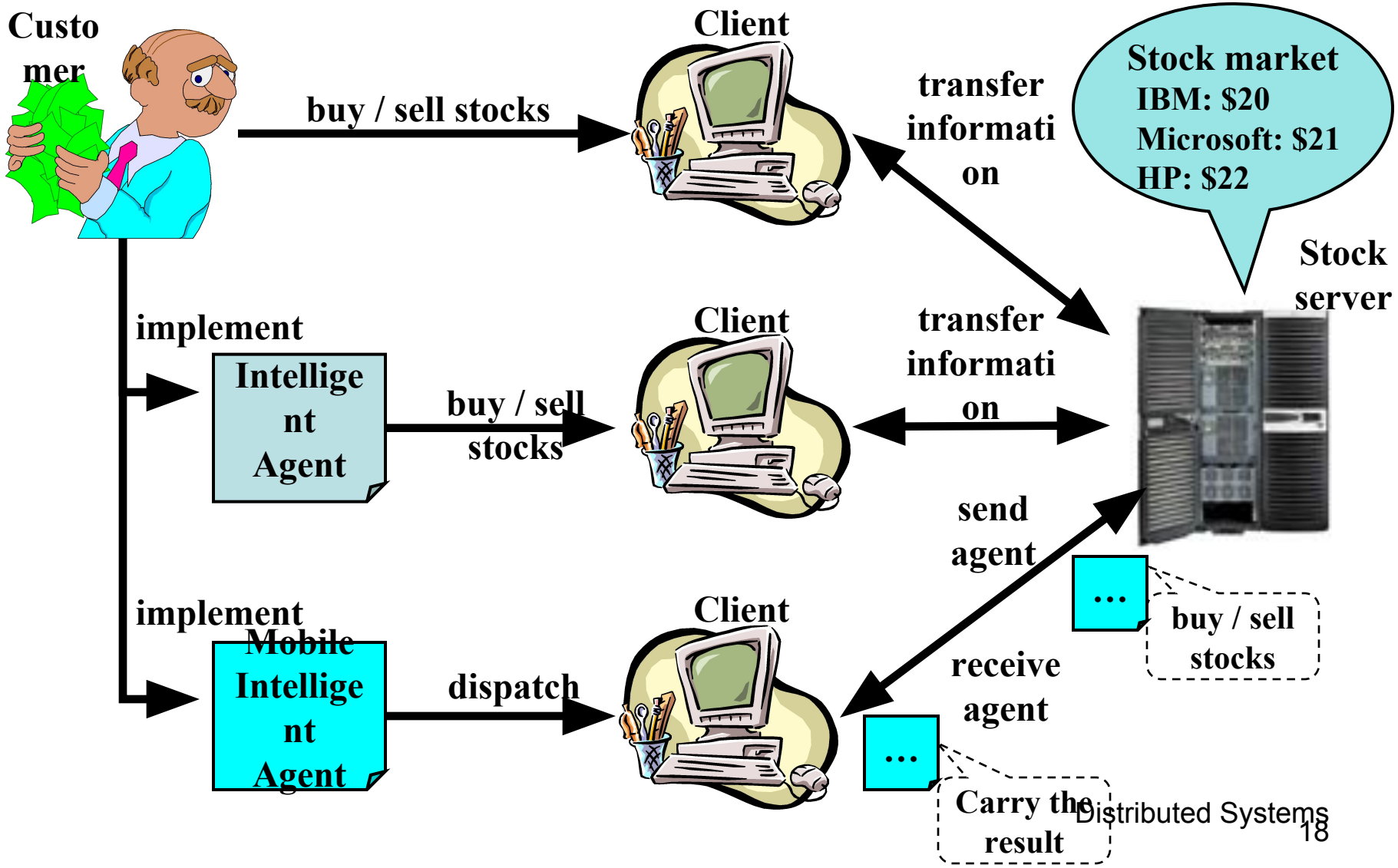
- **Cognitive/Deliberative Agent**
  - These agents have internal models, enabling them to reason and plan actions based on both current and past information.
  - Example: Virtual personal assistants, like Siri, which analyze queries and decide on an appropriate response.
- **Learning Agent:**
  - Equipped with machine learning capabilities, these agents improve their performance by learning from data and experiences.
  - Example: AI-powered recommendation systems for streaming platforms.

Continue…

- **Mobile Agent:**
  - A self-contained process that can autonomously migrate from host to host in order to perform its task on Internet.
  - The motto of Mobile Agents is:

    move the computations to the data rather than the data to the computations

Why do we need mobile agents?



Why do we need mobile agents?

- We defined the scenario, such that a customer wants to trade his stocks in the remote stock market through the Internet. The remote stock market is indicated by a stock server, with maintains all the stock information.

- Choice 1: sit in front of the client machine, logon to the website of the stock market, monitor the latest stock prices and pick up the right moment for trading.
  - Disadvantage: need to sit there the whole day.
  - Network connection between the client machine and the stock server need to be active all the time.
  - Network traffic is heavy because the stock prices need to be updated instantly.

- Choice 2: implement an intelligent agent, and let the agent monitor the stock market for the customer.
  - Disadvantages: the network connection still needs to be keep active all the time.
  - Disadvantages: the network traffic for instant stock prices updating remains the same.

Why do we need mobile agents?

- Choice 3: implement an mobile intelligent agent, which can be delivered to the remote server. Therefore the missions of stock trading can be fulfilled on the remote stock server. This agent will be sent back to the client machine carrying the trade results after finishing all the operations.
  - Advantages: the network connection only need to be available during the periods of agent sending and returning back, which makes the entire system more reliable.
  - Advantages: the network traffic will become much less. Because the network resources are only charged for delivering the mobile intelligent agent, which normally will be just a small piece of code (few k bytes).

 This example shows that the mobile agent system does give us some benefits when it is used in particular situations.

Different Types of Agents (Continue…)

- **Interface Agent:**
  - Are agents that assist an end user in the use of one or more applications.
  - An interface agent has **learning capabilities**.
  - The more often it interacts with the user, the better its assistance become.
  - For example: special interface agent exists that actively seek to bring buyers and sellers together.
- **Information Agent:**
  - The main function of these agents is to manage information from many different sources.
  - Managing information includes ordering, filtering, collating and so on.
  - For example: an e-mail agent may be capable of filtering unwanted mail from its owner's mailbox or automatically distributing incoming mail into appropriate subject-specific mailboxes.

Software Agents in Distributed Systems

Property	Common to all agents?	Description
Autonomous	Yes	Can act on its own
Reactive	Yes	Responds timely to changes in its environment
Proactive	Yes	Initiates actions that affects its environment
Communicative	Yes	Can exchange information with users and other agents
Continuous	No	Has a relatively long lifespan
Mobile	No	Can migrate from one site to another
Adaptive	No	Capable of learning

Some important properties by which different types of agents can be distinguished.

Design Requirements for distributed architectures

Performance Issues

- Performance issues arises from the limited processing and communication capacities of computers.
- Following parameters are the matrices to measure the performance of a system.
  - Responsiveness
    - Users of interactive applications require a fast and consistent response of interaction; but client program often need to access shared resources.
    - When a remote service is invoked, the speed at which the response is generated is determined not just by the load and performance of the server and the network but also by delays in all the software components involved.
    - E.g. a web browser can access the cached pages faster than the non-cached pages.

Design Requirements for distributed architectures

- Throughput
  - Throughput is the rate at which the computation is done.
  - The ability of a distributed system to perform work for all its users is affected by processing speeds at clients and servers and by data transfer rate.
  - Data that is located on a remote server must be transferred from the server process to the client process, passing through several software layers in both computers. The throughput of the intervening software layers is important as well as that of the network.

Design Requirements for distributed architectures

- Load Balancing:
  - One of the purposes of distributed systems is to enable applications and service processes to proceed concurrently without competing for the same resources and to exploit the available computational resources.
  - For example, the ability to run applets on client computers removes load form the web server, enabling it to provide a better service.

Design Requirements for distributed architectures

- Quality of service
  - The ability of systems to meet deadlines.
  - It depends on availability of the necessary computing and network resources at the appropriate time.
  - This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time.
    - E.g. the task of displaying a frame of video
  - The networks commonly used today, for example to browse the Web, may have good performance characteristics, but when they are heavily loaded their performance degrades significantly----in no way can they be said to provide QoS.

Design Requirements for distributed architectures(Q0S)

- The main properties of the quality of the service are:
  - Reliability
  - Security
  - Performance
  - Adaptability
- Reliability & Security issues are critical in the design of most computer systems. They are strongly related to two of the fundamental models: **the failure model** and **the security model**
- Adaptability is to meet changing system configurations and resource availability.
- Performance indicates responsiveness and computational throughput.

Design Requirements for distributed architectures(QoS)

- Use of caching and replication
  - Distributed systems overcome the performance issues by the use of data replication and caching.
- Dependability issues
  - Dependability is the requirement in most application domain.
  - Dependability of computer systems is defined as:
    - Correctness
      - The development of techniques for checking or ensuring the correctness of distributed and concurrent programs is the subject of much current and recent research.
    - Security
      - Security is locating sensitive data and other resources only in computers that can be secured effectively against attack. E.g. a hospital database
    - Fault tolerance
      - Dependable applications should continue to function in the presence of faults in hardware, software, and networks.
      - Reliability is achieved by redundancy.

Models for Inter Process Communication (IPC)

- Distributed systems are composed of a collection of concurrent processes interacting with each other by means of a network connection.
- Therefore, IPC is a fundamental aspect of distributed systems design and implementation.
- IPC is used to either exchange data and information or coordinate the activity of processes.
- There are several different models in which processes can interact with each other; these map to different abstractions for IPC.
- Among the most relevant that we can mention are shared memory, remote procedure call (RPC), and message passing

Message-based communication (IPC Reference Model)

- The abstraction of message has played an important role in the evolution of the models and technologies enabling distributed computing.
- A distributed system is “one in which components located at networked computers communicate and coordinate their actions only by passing messages.”
- The term **message**, in this case, identifies any discrete amount of information that is passed from one entity to another.
- It encompasses any form of data representation that is limited in size and time.

Different Message-based Communication

- **Message passing:**
  - This paradigm introduces the concept of a message as the main abstraction of the model.
  - The entities exchanging information explicitly encode in the form of a message the data to be exchanged.
  - The structure and the content of a message vary according to the model.
  - Examples of this model are the Message-Passing Interface (MPI) and OpenMP.
- **Remote procedure call (RPC):**
  - This paradigm extends the concept of procedure call beyond the boundaries of a single process, thus triggering the execution of code in remote processes.
  - In this case, underlying client/server architecture is implied.
  - A remote process hosts a server component, thus allowing client processes to request the invocation of methods, and returns the result of the execution.
  - Messages, automatically created by the RPC implementation, convey the information about the procedure to execute along with the required parameters and the return values.
  - The use of messages within this context is also referred as **marshaling** of parameters and return values.

Different Message-based Communication(Cont…)

- **Distributed objects:**
  - This is an implementation of the RPC model for the object-oriented paradigm and contextualizes this feature for the remote invocation of methods exposed by objects.
  - Each process registers a set of interfaces that are accessible remotely.
  - Client processes can request a pointer to these interfaces and invoke the methods available through them.
  - The underlying runtime infrastructure is in charge of transforming the local method invocation into a request to a remote process and collecting the result of the execution.
  - The communication between the caller and the remote process is made through messages.
  - Examples of distributed object infrastructures are Common Object Request Broker Architecture (CORBA), Component Object Model (COM, DCOM, and COM1), Java Remote Method Invocation (RMI), and .NET Remoting

Different Message-based Communication (Cont…)

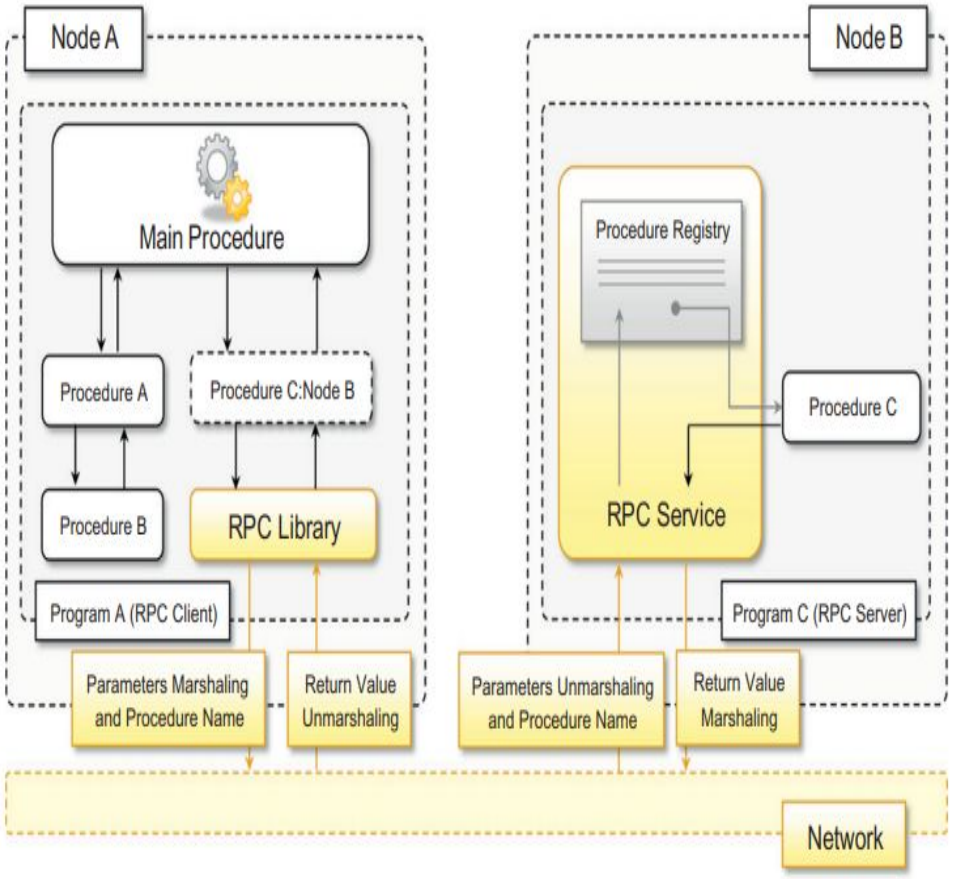
- **Distributed agents and active objects**.
  - In this model the objects have their own control thread, which allows them to carry out their activity.
  - These models often make explicit use of messages to trigger the execution of methods, and a more complex semantics is attached to the messages.
- **Web services:**
  - Web service technology provides an implementation of the RPC concept over HTTP, thus allowing the interaction of components that are developed with different technologies.
  - A Web service is exposed as a remote object hosted on a Web server, and method invocations are transformed in HTTP requests, packaged using specific protocols such as Simple Object Access Protocol (SOAP) or Representational State Transfer (REST).

Technologies for distributed computing (Remote Procedure Call)

- RPC is the fundamental abstraction enabling the execution of procedures on client's request.
- RPC allows extending the concept of a procedure call beyond the boundaries of a process and a single memory address space.
- The called procedure and calling procedure may be on the same system or they may be on different systems in a network.
- The concept of RPC has been discussed since 1976 and completely formalized by Nelson and Birrell in the early 1980s.
- From there on, it has not changed in its major components.
- Even though it is a quite old technology, RPC is still used today as a fundamental component for IPC in more complex systems

RPC (Cont…)

- Figure illustrates the major components that enable an RPC system.
- The system is based on a client/server model.
- The server process maintains a registry of all the available procedures that can be remotely invoked and listens for requests from clients that specify which procedure to invoke, together with the values of the parameters required by the procedure.
- The calling process thread remains blocked until the procedure on the server process has completed its execution and the result (if any) is returned to the client.



RPC (Cont…)

- An important aspect of RPC is **marshaling**, which identifies the process of converting parameter and return values into a form that is more suitable to be transported over a network through a sequence of bytes.
- The term **unmarshaling** refers to the opposite procedure.
- Marshaling and unmarshaling are performed by the RPC runtime infrastructure, and the client and server user code does not necessarily have to perform these tasks.
- The RPC runtime, on the other hand, is not only responsible for parameter packing and unpacking but also for handling the request-reply interaction that happens between the client and the server process in a completely transparent manner.

- Therefore, developing a system using RPC for IPC consists of the following steps:
  - Design and implementation of the server procedures that will be exposed for remote invocation.
  - Registration of remote procedures with the RPC server on the node where they will be made available.
  - Design and implementation of the client code that invokes the remote procedure(s)
- RPC has been a dominant technology for IPC for quite a long time, and several programming languages and environments support this interaction pattern in the form of libraries and additional packages.
- For instance, RPyC is an RPC implementation for Python. There also exist platform independent solutions such as XML-RPC and JSON-RPC, which provide RPC facilities over XML and JSON, respectively

Remote Procedure Calls (1)

- A remote procedure call occurs in the following steps:

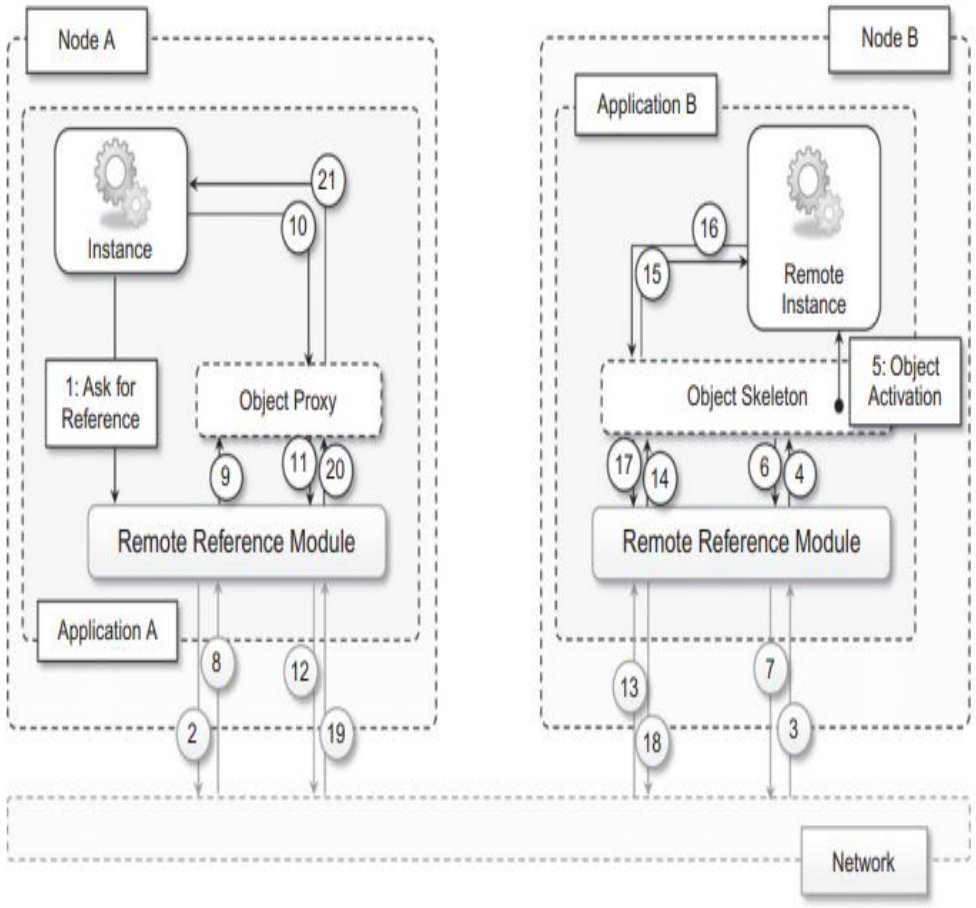
1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Technologies for distributed computing (Distributed object frameworks)

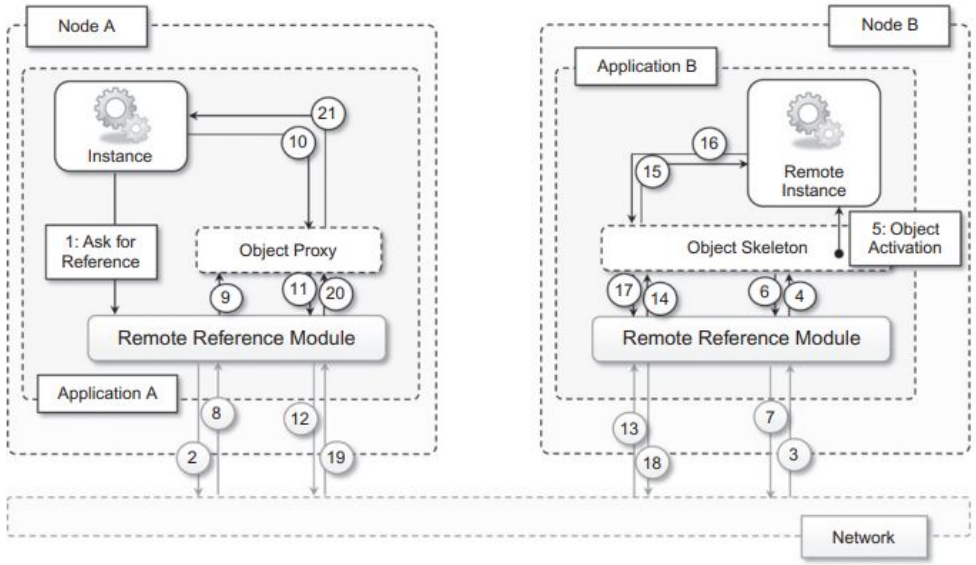
- Distributed object frameworks extend object-oriented programming systems by allowing objects to be distributed across a heterogeneous network and provide facilities so that they can coherently act as though they were in the same address space.
- Distributed object frameworks use the basic mechanism introduced with RPC and extend it to enable the remote invocation of object methods and to keep track of references to objects made available through a network connection.
- Therefore, the common interaction pattern is the following:

1. The server process maintains a registry of active objects that are made available to other processes. According to the specific implementation, active objects can be published using interface definitions or class definitions.
2. The client process, by using a given addressing scheme, obtains a reference to the active remote object. This reference is represented by a pointer to an instance that is of a shared type of interface and class definition.
3. The client process invokes the methods on the active object by calling them through the reference previously obtained. Parameters and return values are marshaled as happens in the case of RPC.

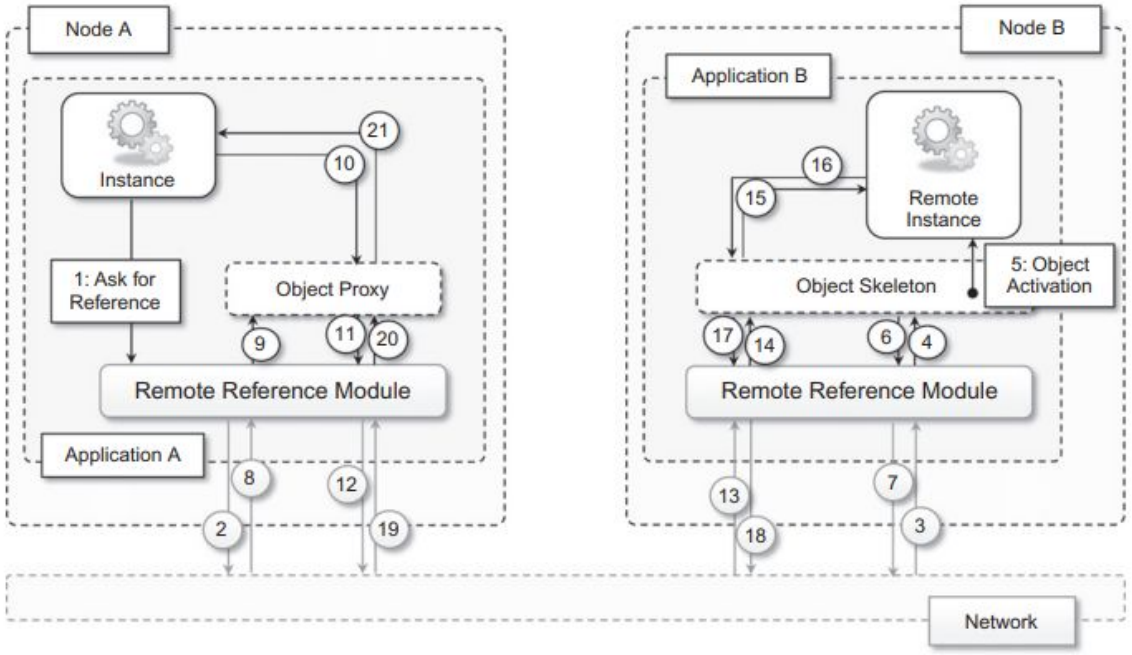
- Distributed object frameworks give the illusion of interaction with a local instance while invoking remote methods.
- This is done by a mechanism called a **proxy skeleton.**
- Figure gives an overview of how this infrastructure works.
- Proxy and skeleton always constitute a pair:
  - the server process maintains the skeleton component, which is in charge of executing the methods that are remotely invoked, while the client maintains the proxy component, allowing its hosting environment to remotely invoke methods through the proxy interface.



- The transparency of remote method invocation is achieved using one of the fundamental properties of object-oriented programming: **inheritance and subclassing**.
- Both the proxy and the active remote object expose the same interface, defining the set of methods that can be remotely called.



- On the client side, a runtime object published by the server is generated.
- This object translates the local method invocation into an RPC call for the corresponding method on the remote active object.
- On the server side, whenever an RPC request is received, it is unpacked and the method call is dispatched to the skeleton that is paired with the client that issued the request.
- Once the method execution on the server is completed, the return values are packed and sent back to the client, and the local method call on the proxy returns.



Common object request broker architecture (CORBA)

- CORBA is a specification introduced by the Object Management Group (OMG) for providing cross platform and cross-language interoperability among distributed components.
- The specification was originally designed to provide an interoperation standard that could be effectively used at the industrial level.
- The current release of the CORBA specification is version 3.0 and currently the technology is not very popular, mostly because the development phase is a considerably complex task and the interoperability among components developed in different languages has never reached the proposed level of transparency.
- A fundamental component in the CORBA architecture is the Object Request Broker (ORB), which acts as a central object bus.
- A CORBA object registers the interface it is exposing in the ORB, and clients can obtain a reference to that interface and invoke methods on it.
- The ORB is responsible for returning the reference to the client and managing all the low-level operations required to perform the remote method invocation
- CORBA is not supported by Microsoft, which instead has developed its own distributed object management architecture called DCOM

How it Works

- In a CORBA environment, programs request services through an object request broker (ORB), which allows components of distributed applications to find each other and communicate without knowing where applications are located on the network or what kind of interface they use.
- ORBs are the **middleware** that enable client and server programs to establish sessions with each other, independent of their location on the network or their programming interface.
- The process of a client invoking a call to an application programming interface (API) on a server object is transparent.
- The client issues the call, which is intercepted by the ORB.
- The ORB takes the call and is responsible for locating a server object that is able to implement the request.
- Once it has located such an object, the ORB invokes the object's method and passes it any parameters submitted by the client.
- The results are then returned to the client.
- ORBs communicate among themselves using the **General Inter-ORB Protocol (GIOP)** or the **Internet Inter-ORB Protocol (IIOP)** so that any ORB can fulfill any client request on the network.

- To simplify cross-platform interoperability, interfaces are defined in **Interface Definition Language (IDL)**, which provides a platform-independent specification of a component.
- An IDL specification is then translated into a **stub-skeleton pair** by specific CORBA compilers that generate the required client (stub) and server (skeleton) components in a specific programming language.
- These templates are completed with an appropriate implementation in the selected programming language.
- This allows CORBA components to be used across different runtime environment by simply using the stub and the skeleton that match the development language used.