Distributed Systems Architectures

BOOK REFERENCED: IAN SOMMERVILLE, 8TH EDITION, CHAPTER 12

Objectives

- To explain the advantages and disadvantages of different distributed systems architectures
- To discuss client-server and distributed object architectures

Topics Covered

- Distributed Systems
- Client-server architectures
- Distributed object architectures

Distributed Systems

- Virtually all large computer-based systems are now distributed systems.
- Information processing is distributed over several computers rather than confined to a single machine.
- Distributed software engineering is therefore very important for enterprise computing systems.

Distributed System Characteristics

- Resource sharing
  - Sharing of hardware and software resources.
- Openness
  - Use of equipment and software from different vendors.
- Concurrency
  - Concurrent processing to enhance performance.
- Scalability
  - Increased throughput by adding new resources.
- Fault tolerance
  - The ability to continue in operation after a fault has occurred.

Distributed System Disadvantages

- Complexity
  - Typically, distributed systems are more complex than centralised systems.
- Security
  - More susceptible to external attack.
- Manageability
  - More effort required for system management.
- Unpredictability
  - Unpredictable responses depending on the system organisation and network load.

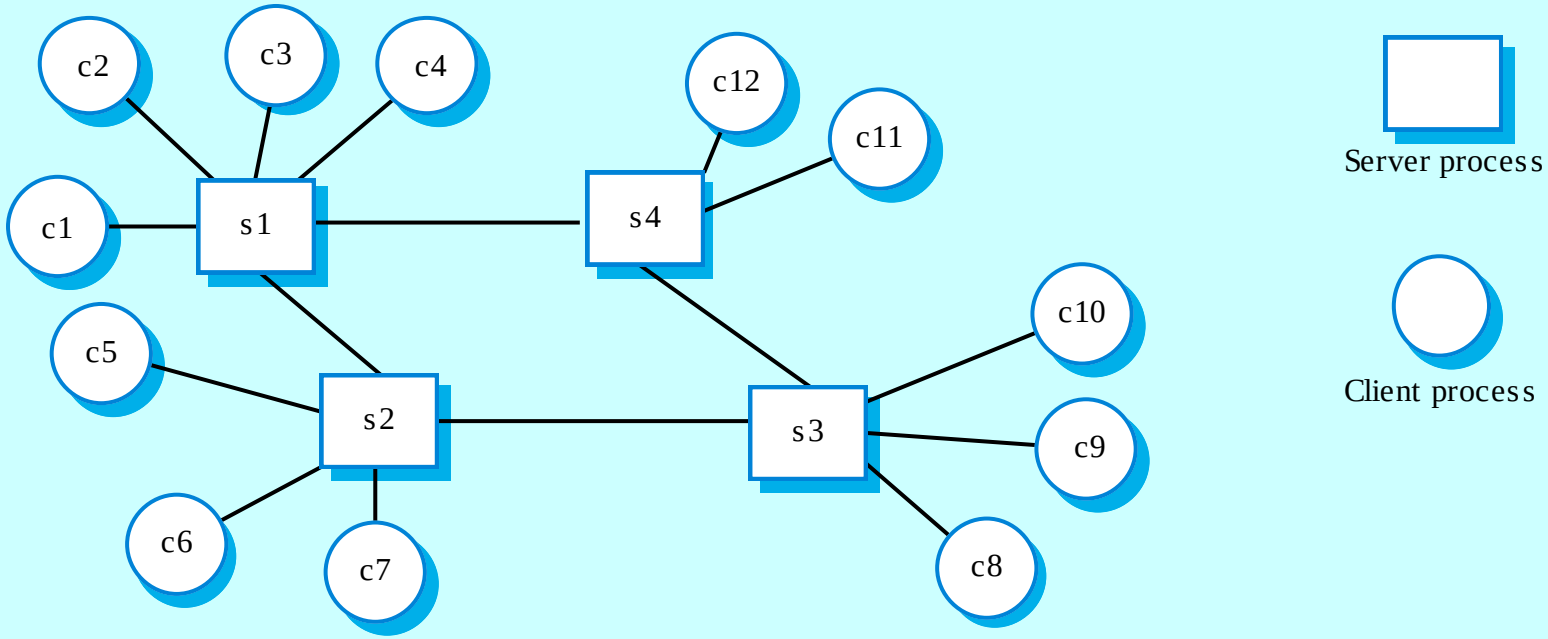
Distributed Systems Architectures

- Client-server architectures
  - Distributed services which are called on by clients. Servers that provide services are treated differently from clients that use services.
- Distributed object architectures
  - No distinction between clients and servers. System may be thought of set of interacting objects irrelevant of location. Any object on the system may provide and use services from other objects.

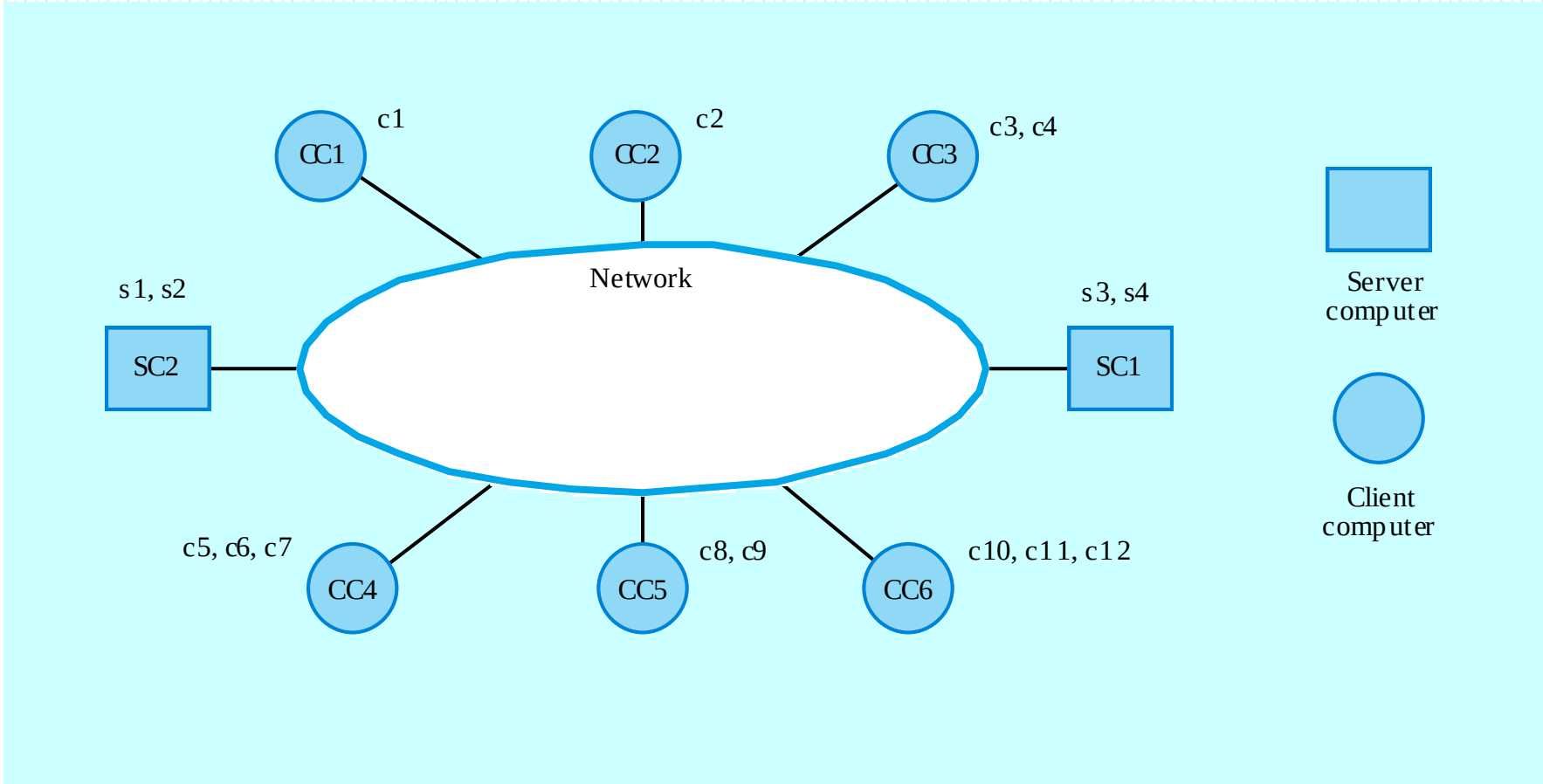
Client Server Architecture

- The application is modelled as a set of services that are provided by servers and a set of clients that use these services.
- Clients know of servers but servers need not know of clients.
- Clients and servers are logical processes and need not be physical computers
- Mapping between processes and processors need not be 1:1

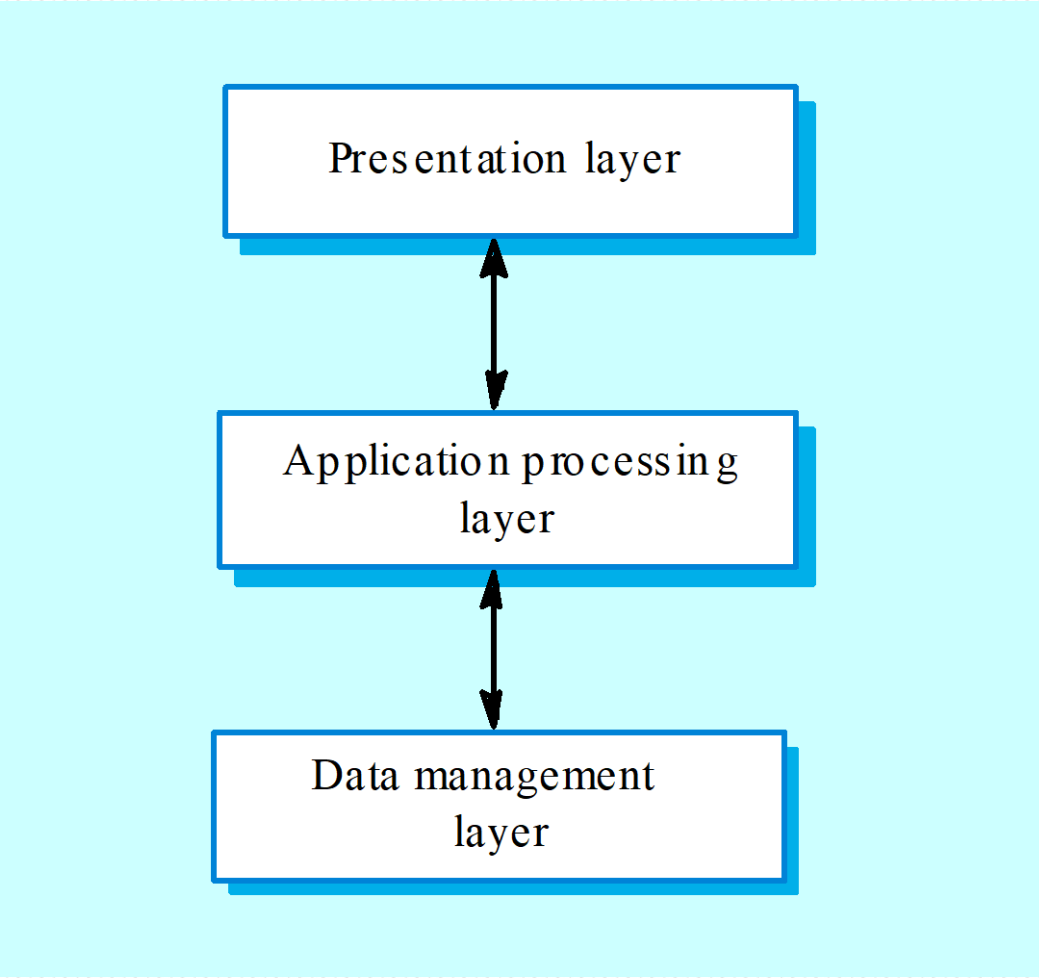
A client-server system



Computers in a Client - server network



Application Layers



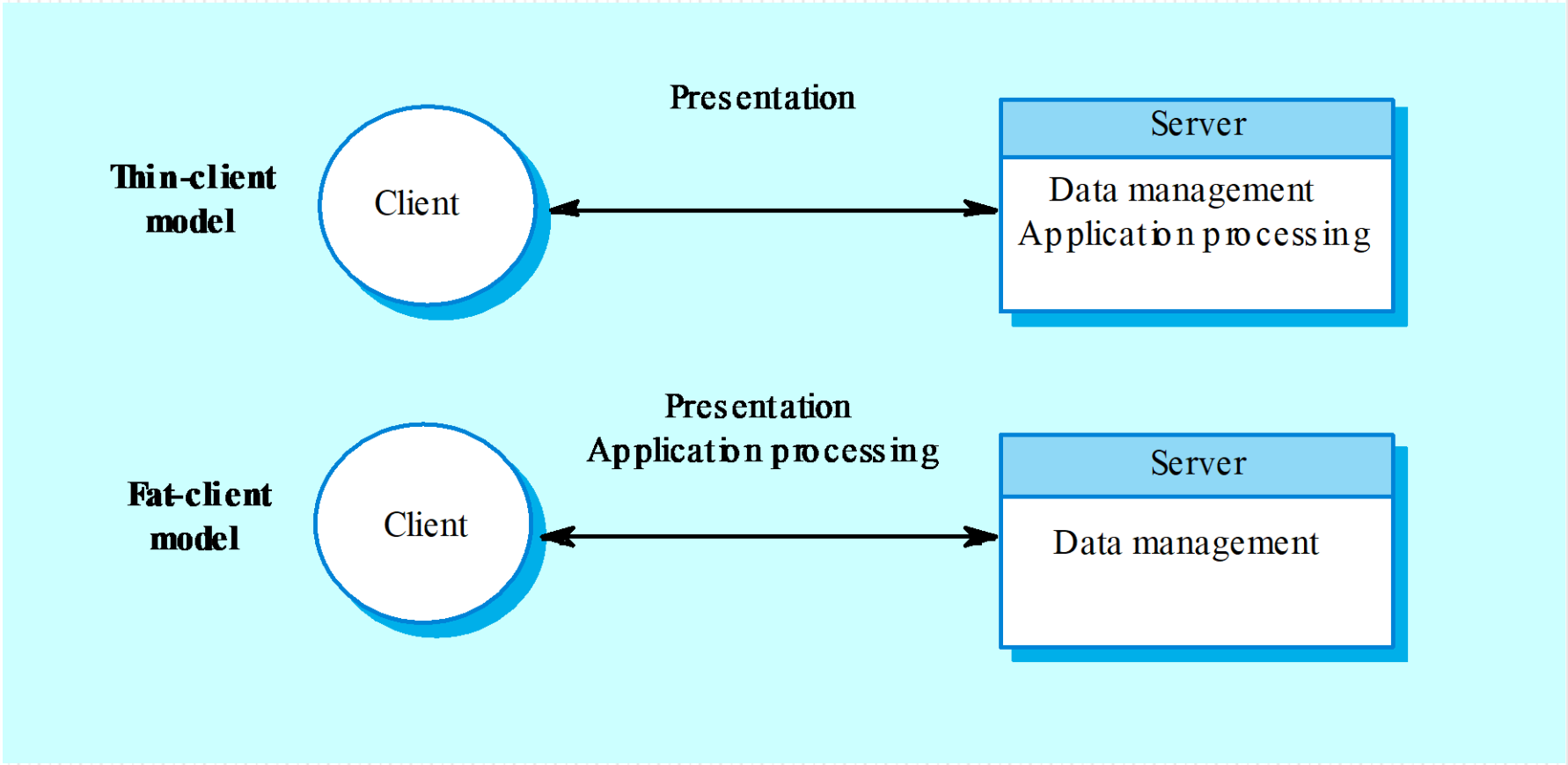
Layered Application Architecture

- Presentation layer
  - Concerned with presenting the results of a computation to system users and with collecting user inputs.
- Application processing layer
  - Concerned with providing application specific functionality e.g., in a banking system, banking functions such as open account, close account, etc.
- Data management layer
  - Concerned with managing the system databases.

Thin and fat clients

- The simplest client–server architecture is called a *two-tier client–server architecture,* where an application is organised as a server (or multiple identical servers) and a set of clients.
- Thin-client model
  - In a thin-client model, all of the application processing and data management is carried out on the server. The client is simply responsible for running the presentation software.
- Fat-client model
  - In this model, the server is only responsible for data management. The software on the client implements the application logic and the interactions with the system user.

Thin and fat clients



Thin client model

- A "thin" client is one that requires no additional software be downloaded.
- Just acts like a graphical user interface like a browser.
- Easy Distribution : A major advantage of the thin client is the ability to make changes to the application without having to push software to every desktop that uses it.
- Less Expensive Terminals : Thin client applications tend to have much of their complex business logic on the remote server.
- Poorer Response Times : Because the thin client leaves the majority of the business logic on the server, it must call that server for any change.
- Web browsers and web-based apps such as WordPress, Google Docs, and web-based online games are also examples of a thin client. Devices used for media streaming such as Chromecast and Apple TV installed with streaming apps such as Netflix or Spotify are technically examples of thin clients.

Fat client model

- A "thick" client is one where that part of the application must be downloaded to the desktop.
- More processing is delegated to the client as the application processing is locally executed, ability to locally process their own data and/or programs.
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.

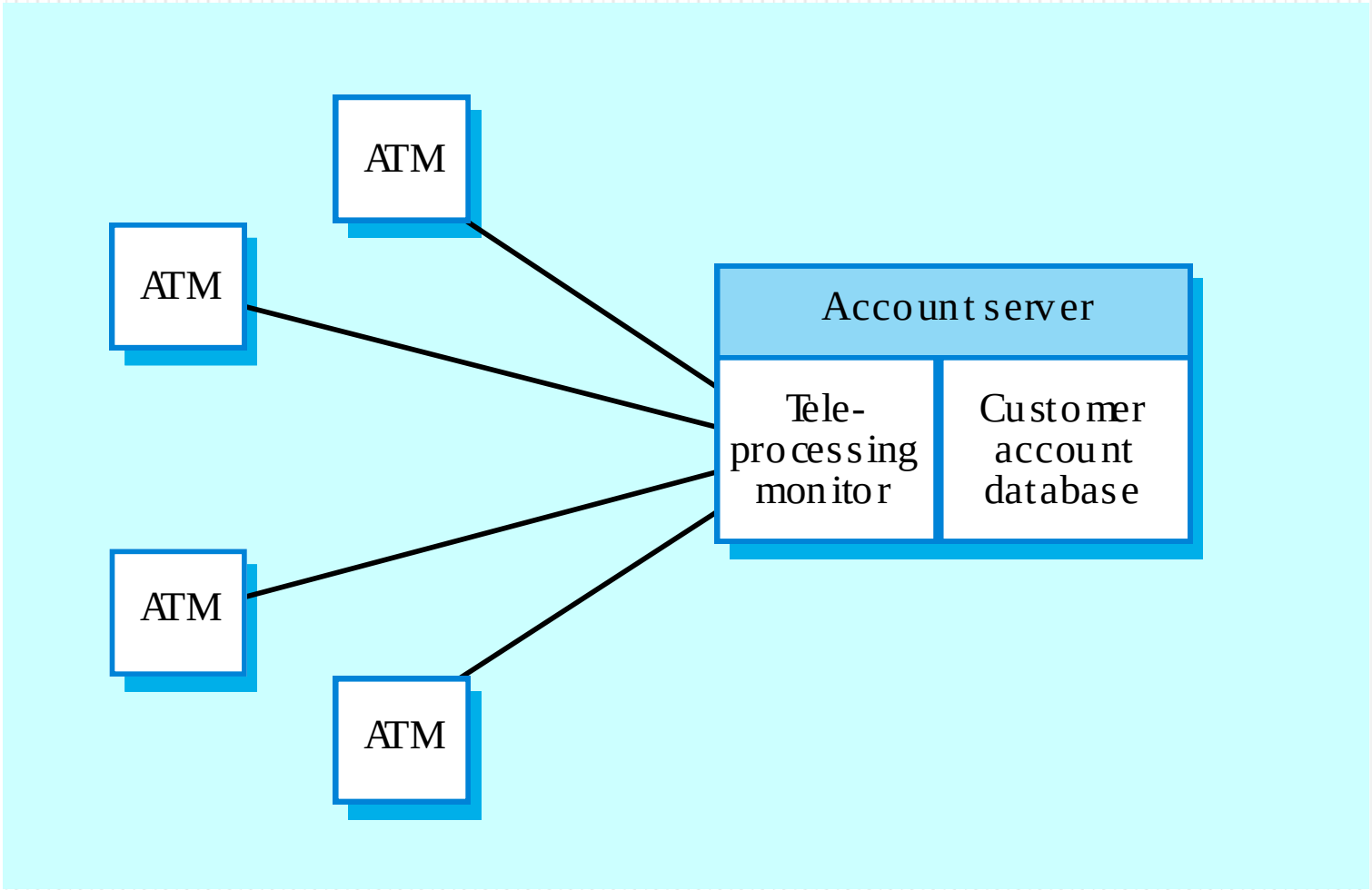
- Better data and program processing
- Server performance efficiency
- Can work offline

- Need of hardware resources
- Need of software maintenance
- Complex to manage

Think client vs. thick (fat) client

Thin Clients	Thick Clients
- Easy to deploy as they require no extra or specialized software installation - Needs to validate with the server after data capture - If the server goes down, data collection is halted as the client needs constant communication with the server - Cannot be interfaced with other equipment (in plants or factory settings for example) - Clients run only and exactly as specified by the server - More downtime - Portability in that all applications are on the server so any workstation can access - Opportunity to use older, outdated PCs as clients - Reduced security threat	- More expensive to deploy and more work for IT to deploy - Data verified by client not server (immediate validation) - Robust technology provides better uptime - Only needs intermittent communication with server - Require more resources but less servers - Can store local files and applications - Reduced server demands - Increased security issues

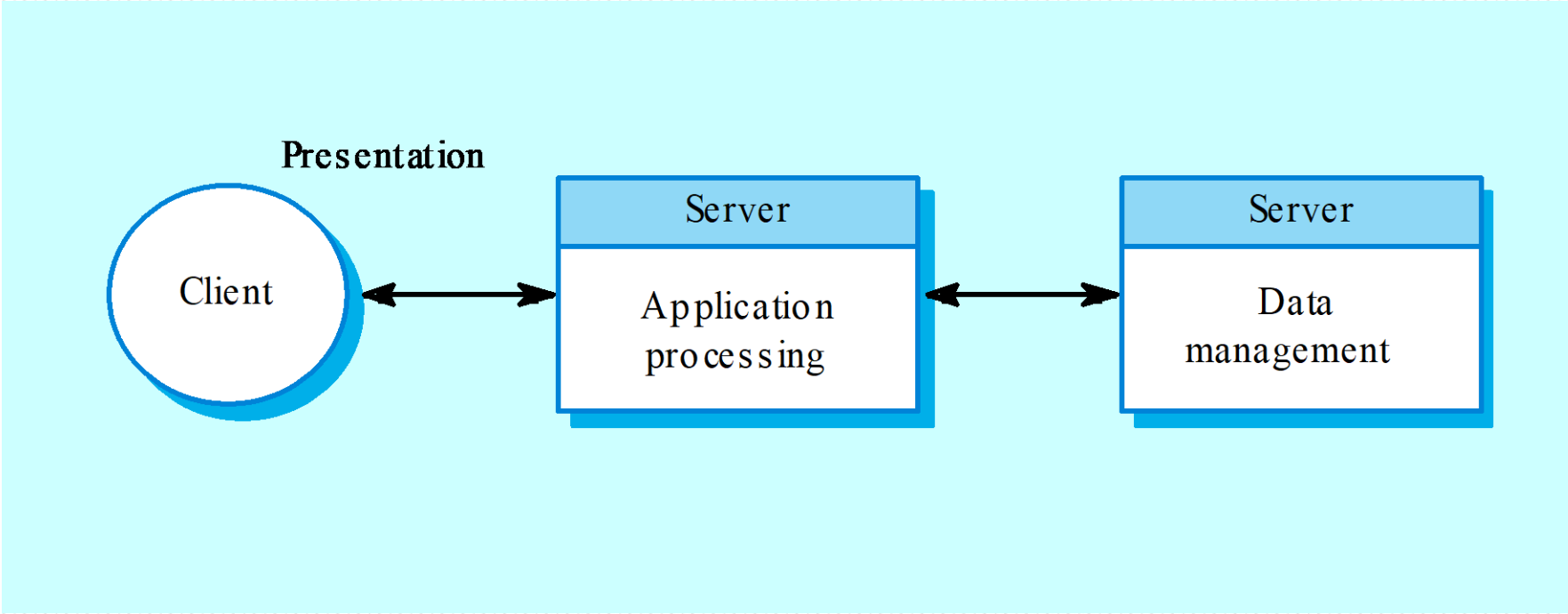
A client-server ATM system



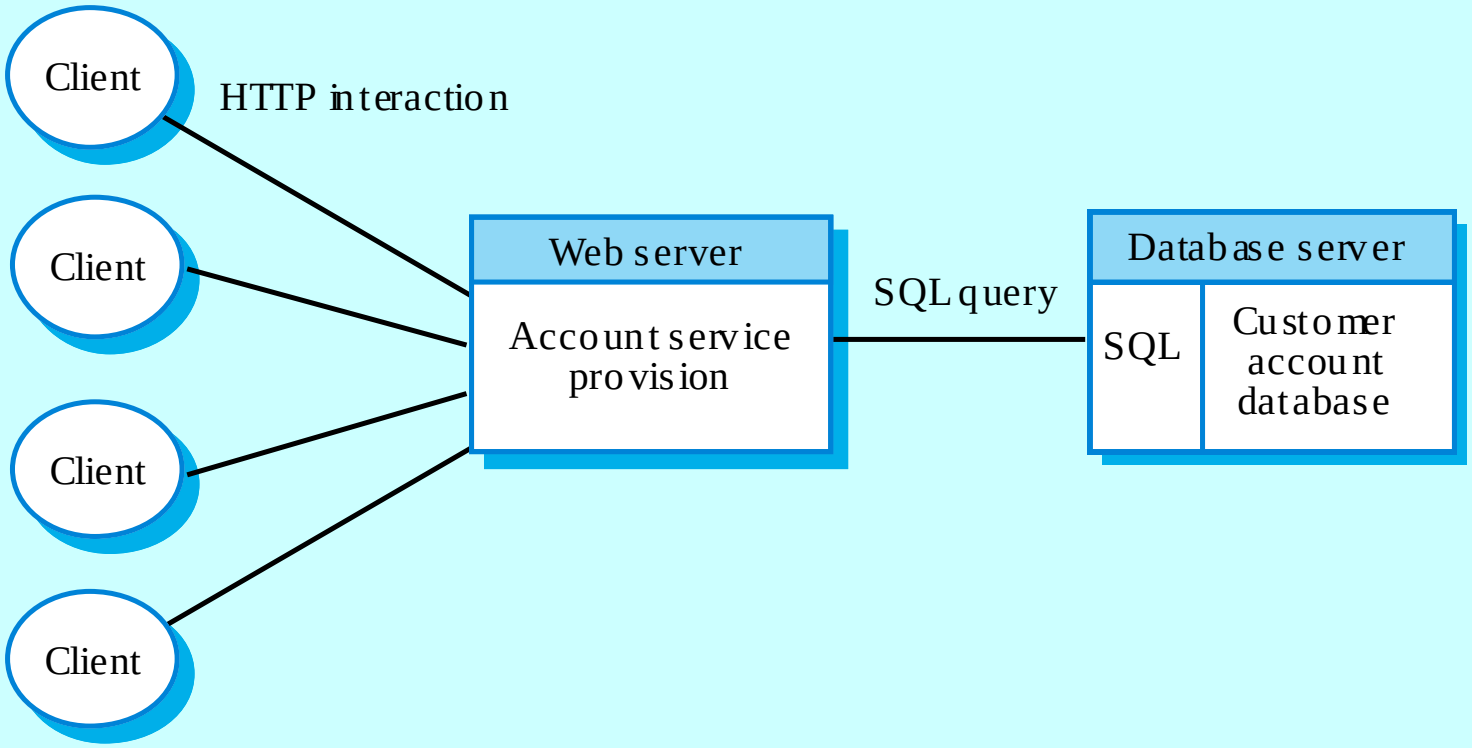
Three-tier architectures

- In a three-tier architecture, each of the application architecture layers may execute on a separate processor.
- Allows for better performance than a thin-client approach and is simpler to manage than a fat-client approach.
- A more scalable architecture - as demands increase, extra servers can be added.

A 3-tier C/S architecture



An internet banking system



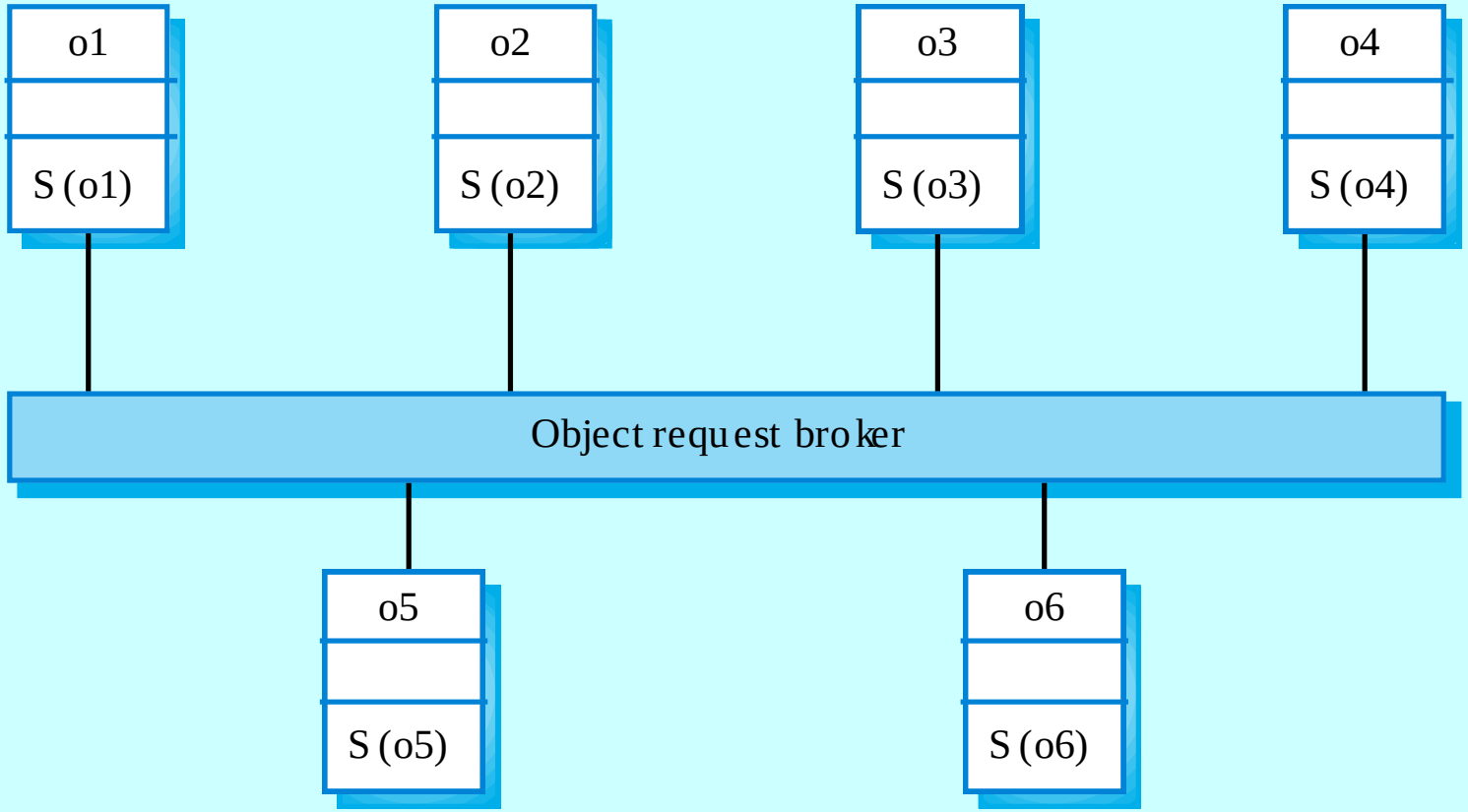
Where to use C/S architectures?

Architecture	Applications
Two-tier C/S architecture with thin clients	Legacy system applications where separating application processing and data management is impractical. Computationally-intensive applications such as compilers with little or no data management. Data-intensive applications (browsing and querying) with little or no application processing.
Two-tier C/S architecture with fat clients	Applications where application processing is provided by off-the-shelf software (e.g. Microsoft Excel) on the client. Applications where computationally-intensive processing of data (e.g. data visualisation) is required. Applications with relatively stable end-user functionality used in an environment with well-established system management.
Three-tier or multi-tier C/S architecture	Large-scale applications with hundreds or thousands of clients. Applications where both the data and the application are volatile. Applications where data from multiple sources are integrated.

Distributed Object Architecture

- There is no distinction in a distributed object architectures between clients and servers.
- The fundamental system components are objects that provide an interface to a set of services that they provide. Other objects call on these services with no logical distinction between a client (a receiver of a service) and a server (a provider of a service).
- Each distributable entity is an object that provides services to other objects and receives services from other objects.
- Object communication is through a middleware system called an object request broker.
- However, distributed object architectures are more complex to design than C/S systems.

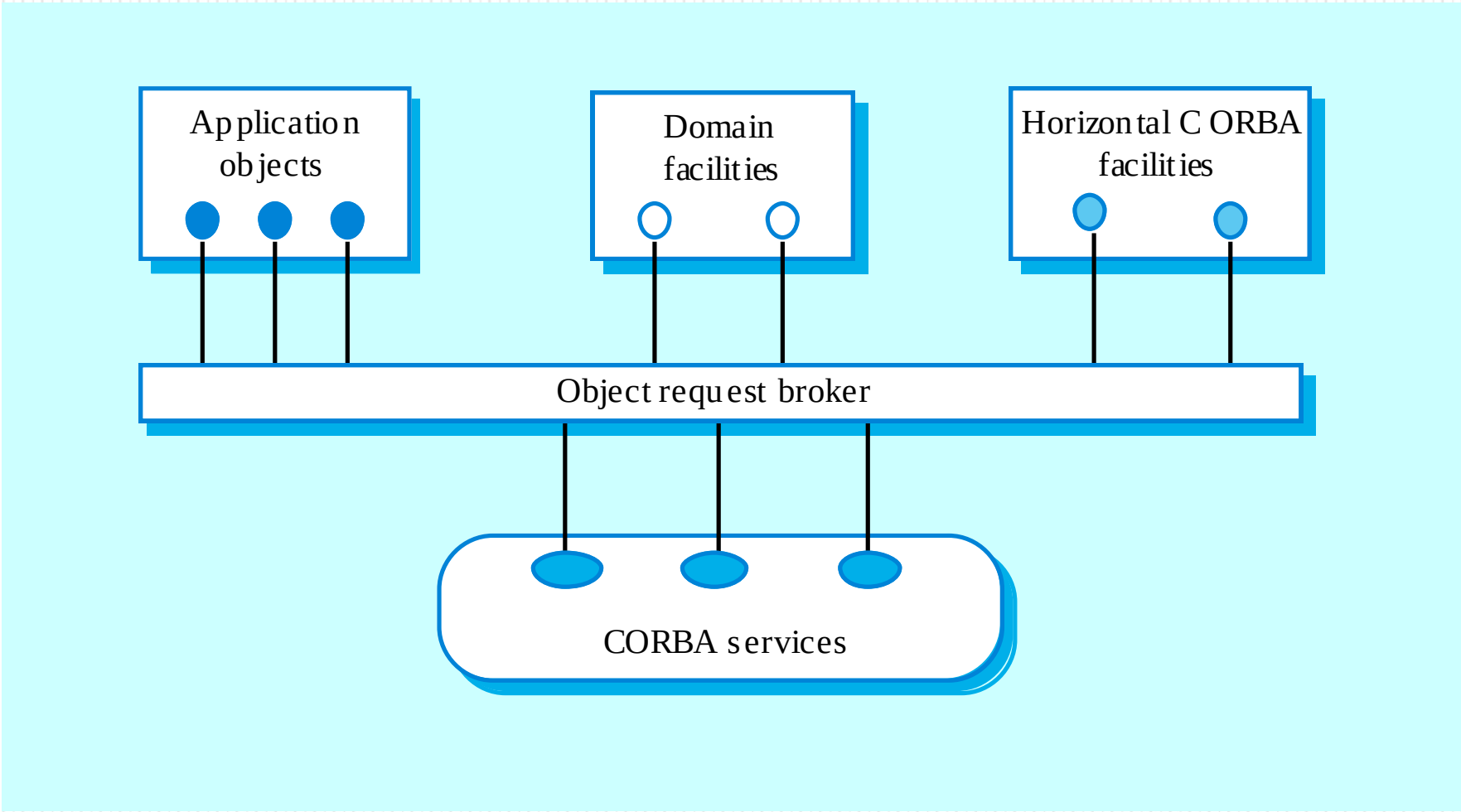
Distributed Object Architecture



CORBA

- CORBA (Common Object Request Broker Architecture) is an international standard for an Object Request Broker - middleware to manage communications between distributed objects.
- The objects in the system may be implemented using different programming languages, the objects may run on different platforms and their names need not be known to all other objects in the system.
- The middleware therefore has to do a lot of work to ensure seamless object communications.
- Middleware for distributed computing is required at 2 levels:
  - At the logical communication level, the middleware allows objects on different computers to exchange data and control information;
  - At the component level, the middleware provides a basis for developing compatible components. CORBA component standards have been defined.

CORBA application structure



Application structure

- Application objects that are designed and implemented for this application.
- Standard/Application objects (vertical), defined by the OMG (Object management group), for a specific domain e.g. insurance.
- Fundamental CORBA services that provide basic distributed computing services such as directories and security management.
- Horizontal CORBA facilities such as user interface facilities.

CORBA standards

- An object model for application objects
  - A CORBA object is an encapsulation of state with a well-defined, language-neutral interface defined in an IDL (interface definition language).
- An object request broker that manages requests for object services.
- A set of general object services of use to many distributed applications. For example, directory services, transaction services etc.
- A set of common components built on top of these services. These may be vertical components (domain specific) or horizontal components (general purpose).

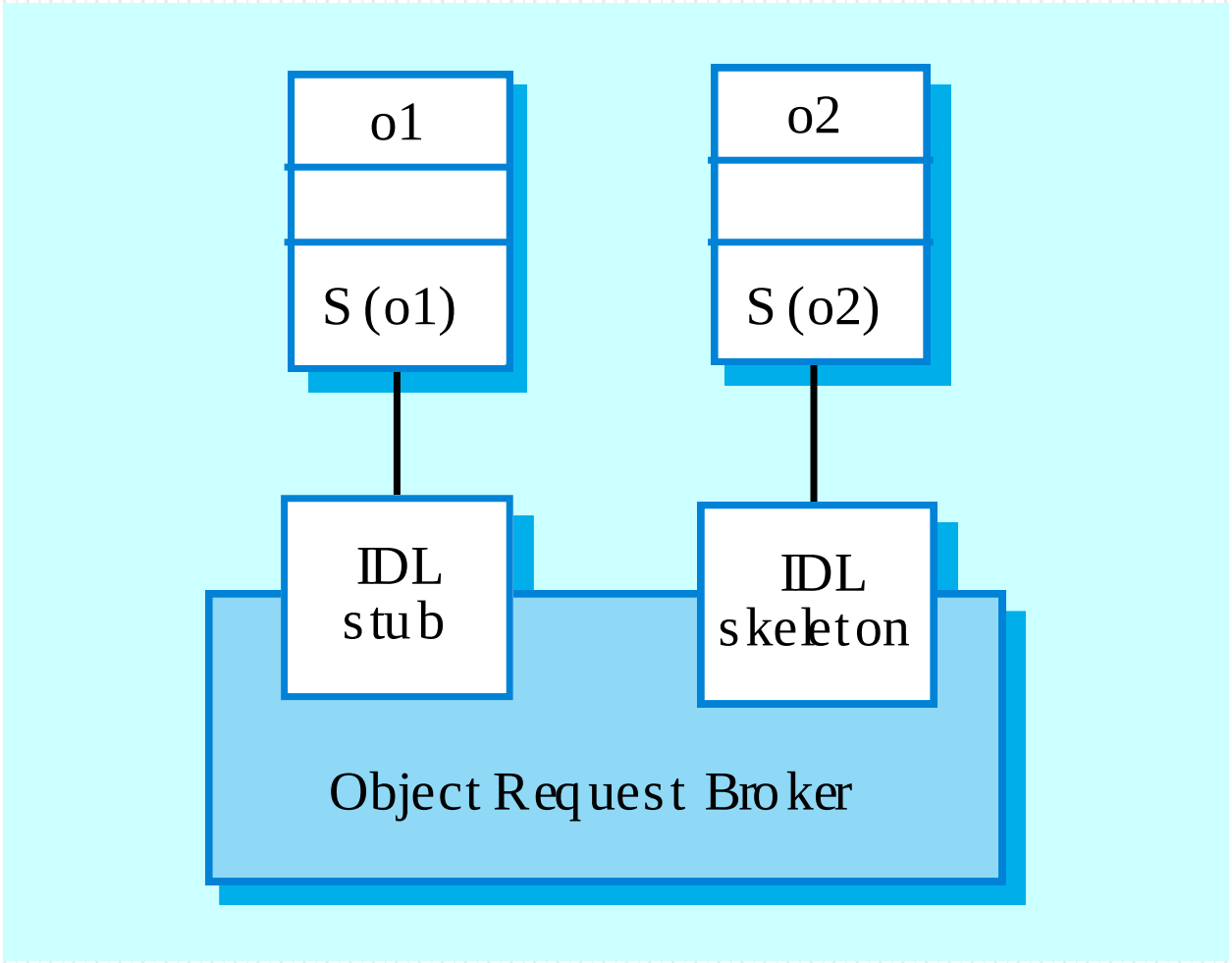
CORBA objects

- CORBA objects are comparable, in principle, to objects in C++ and Java having encapsulation of attributes and services.
- They MUST have a separate interface definition that is expressed using a common language (IDL) similar to C++.
- CORBA object interfaces are defined using a standard, language-independent interface definition language.
- There is a mapping from this IDL to programming languages (C++, Java, etc.).
- Therefore, objects written in different languages can communicate with each other.
- CORBA objects have a unique identifier called an Interoperable Object Reference (IOR), which is used when one object request services from another object.

Object request broker (ORB)

- The ORB handles object communications. It knows of all objects in the system and their interfaces.
- The location of the objects and their implementation are not of any interest to ORB.
- Using an ORB, the calling object binds an IDL stub that defines the interface of the called object.
- Calling this stub results in calls to the ORB which then calls the required object through a published IDL skeleton that links the interface to the service implementation.

ORB-based object communications



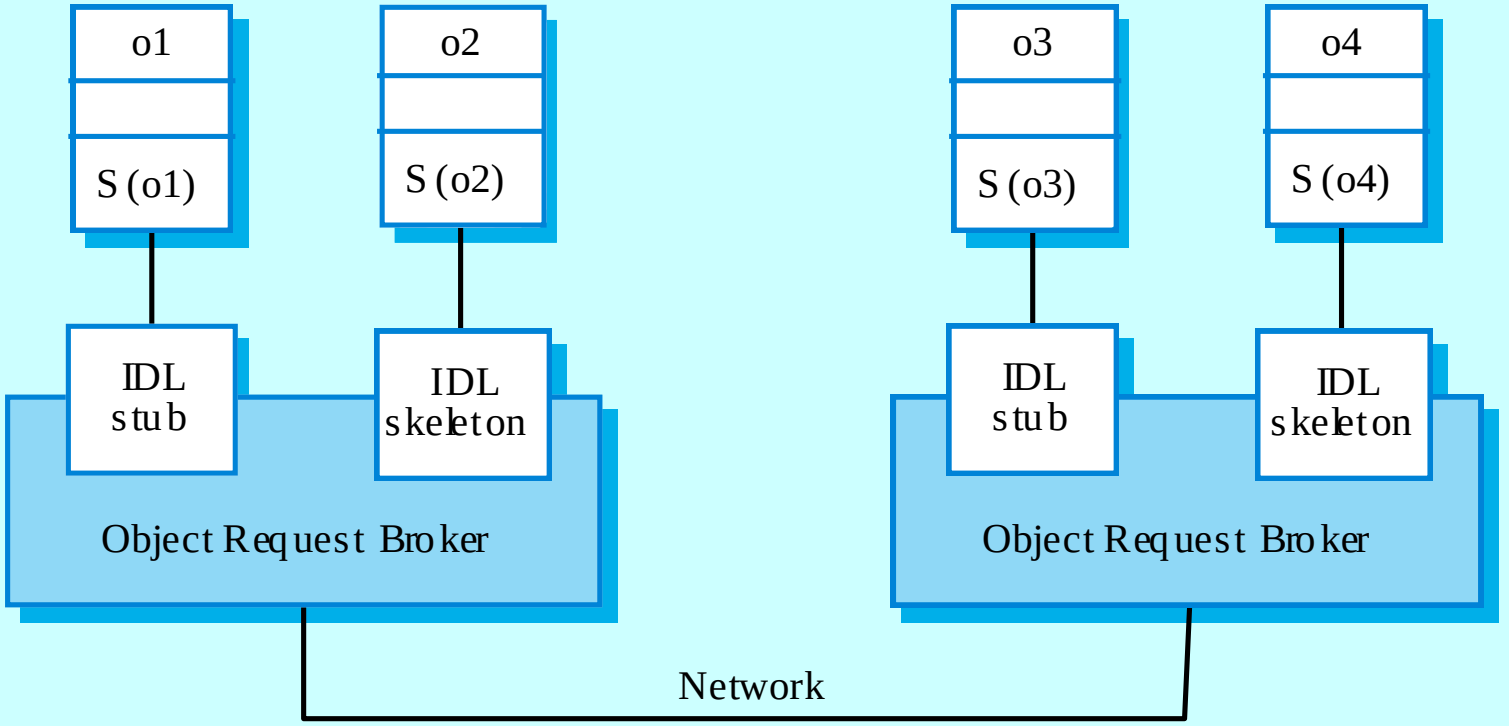
ORB based object communications

- When a service is called through the interface, the IDL skeleton translates this into a call to the service in whatever implementation language has been used.
- When the method or procedure has been executed, the IDL skeleton translates the results into IDL so that it can be accessed by the calling object.
- Where an object both provides services to other objects and use services that are provided elsewhere, it needs both an IDL stub and IDL skeleton.
- IDL stub is required for every object that is used.

Inter-ORB communications

- ORBs are not usually separate programs but are a set of objects in a library that are linked with an application when it is developed.
- Therefore, in a distributed system, each computer that is running distributed objects will have its own object request broker.
- ORBs handle communications between objects executing on the sane machine.
- Inter-ORB communications are used for distributed object calls.

Inter-ORB communications



CORBA services

- Naming and trading services
  - These allow objects to discover and refer to other objects on the network.
- Notification services
  - These allow objects to notify other objects that an event has occurred.
- Transaction services
  - These support atomic transactions and rollback on failure.

Advantages of distributed object architecture

- It allows the system designer to delay decisions on where and how services should be provided.
- It is a very open system architecture that allows new resources to be added to it as required.
- The system is flexible and scalable.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required.

Disadvantages of distributed object architecture

- Complex to design as compared to client server systems.

Uses of distributed object architecture

- As a logical model that allows you to structure and organise the system. In this case, you think about how to provide application functionality solely in terms of services and combinations of services.
- As a flexible approach to the implementation of client-server systems. The logical model of the system is a client-server model but both clients and servers are realised as distributed objects communicating through a common communication framework.

Thank You!!!

Any Questions???