Distributed Systems
Concepts and Architectures

Objectives

- To explain the advantages and disadvantages of different distributed systems architectures
- To discuss client-server and distributed object architectures
- To introduce service-oriented architectures as new model of distributed computing
- To introduce communication models

Distributed System Architectures
Topics Covered

- Distributed system characteristics and disadvantages
- Client-server architectures
- Distributed object architectures

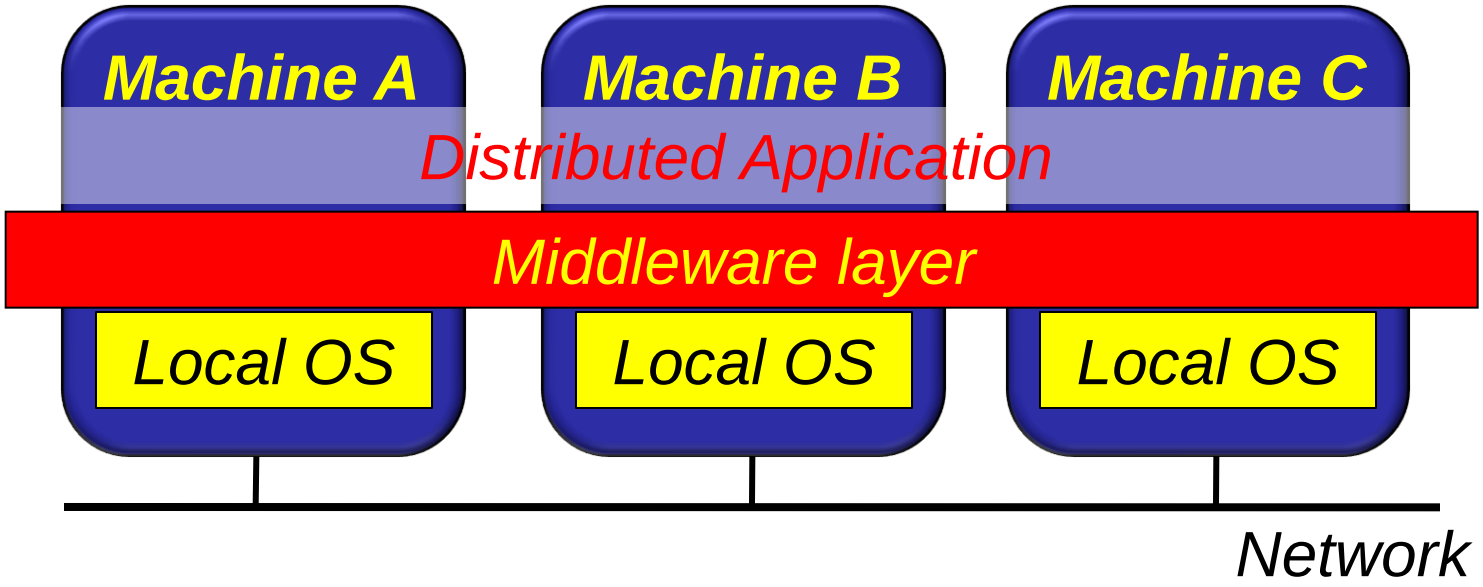
Definition of a Distributed System

A distributed system is:

- *a collection of independent computers that appears to its users as a single coherent system*

- This definition deals with two aspects
  - Hardware: Machines are autonomous
  - Software: Users think that they are dealing with a single system

Definition of a Distributed System



- A distributed system is organized by means of a middleware.
- The middleware layer extends over multiple machines.

Transparency is the basis of a Distributed System

Transparency in a Distributed System

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located (Naming plays IMP. role)
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be ***moved*** to another location while in use
Replication	Hide that a resource may be copied at ***several locations***
Concurrency	Hide that a resource may be ***shared*** by several competitive users (consistent state of resource is IMP)
Failure	Hide the failure and recovery of a resource (Ex. requested time out in web page accessing)
Persistence	Hide whether a (software) resource is in memory or on disk

Different forms of transparency in a distributed system.

Distributed system characteristics

- Resource sharing
  - Sharing of hardware and software resources. (Printer/HPC systems/files)
- Openness
  - Offer services (normally interfaces, described in IDL) according to the standard rules, describing syntax and semantics. Use of equipment and software from different vendors.
- Concurrency
  - Concurrent processing to enhance performance.

Distributed system characteristics

- Scalability
  - Increased throughput by adding new resources.
    - Add user
    - Geographical
    - With multiple admin domain
- 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 vulnerable 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.
  - Communication between C&S can be implemented using connectionless (IP,UDP) protocol if the network is reliable, otherwise it is connection oriented (TCP, DCCP, Phone Call: user must dial the telephone, get an answer before transmitting data, ATM, Frame Relay)
  - In connection less protocol it is difficult to predict if the delay is due to a failure or a communication delay

Where is the Origin

First invented in the 1960s. It was called *Procedural Programming* in those days, and it featured this neat abstraction where:

- A procedure can take requests called "Procedure Calls"
- Given a well-formed request, a procedure will provide a response
- A procedure can use other procedures

Where is the Origin

- In 1981 or thereabouts, **Bruce Jay Nelson** got the bright idea that one should be able to call procedures on other systems

  This was called a remote procedure call, a remarkably simple name.

Clients and Servers

- General interaction between a client and a server. This is known as request-reply behavior.

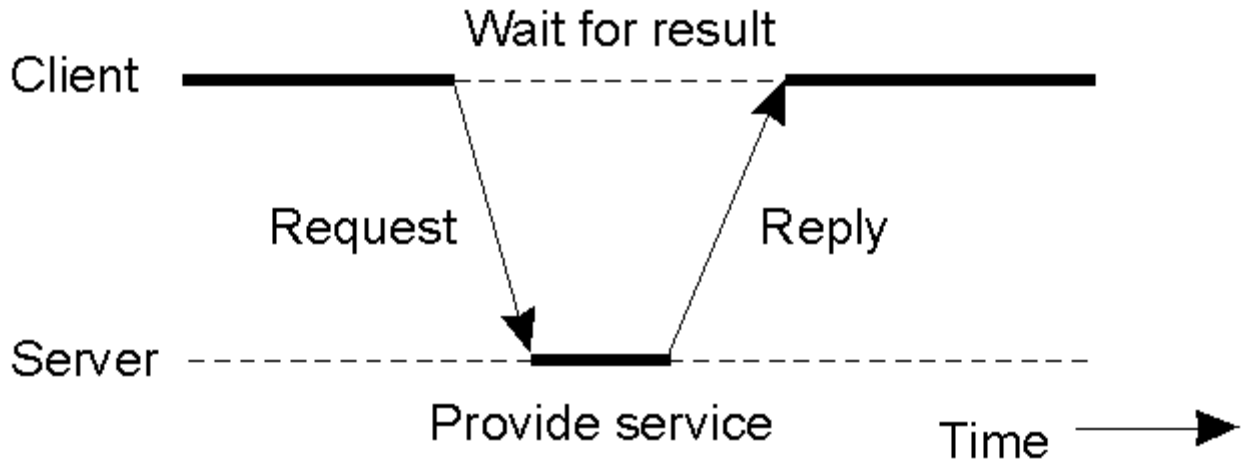


Fig. Curtsey: Distributed Systems: Principles and Paradigms by Tannenbaum

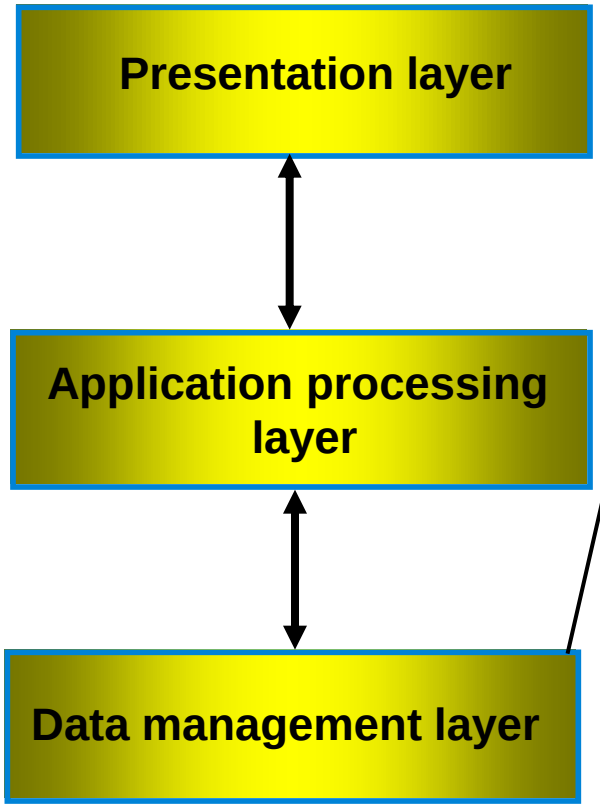
Distributed systems architectures

- **Distributed object architectures**
  - No distinction between clients and servers.
  - Any object on the system may provide and use services from other objects.

Layered application architecture

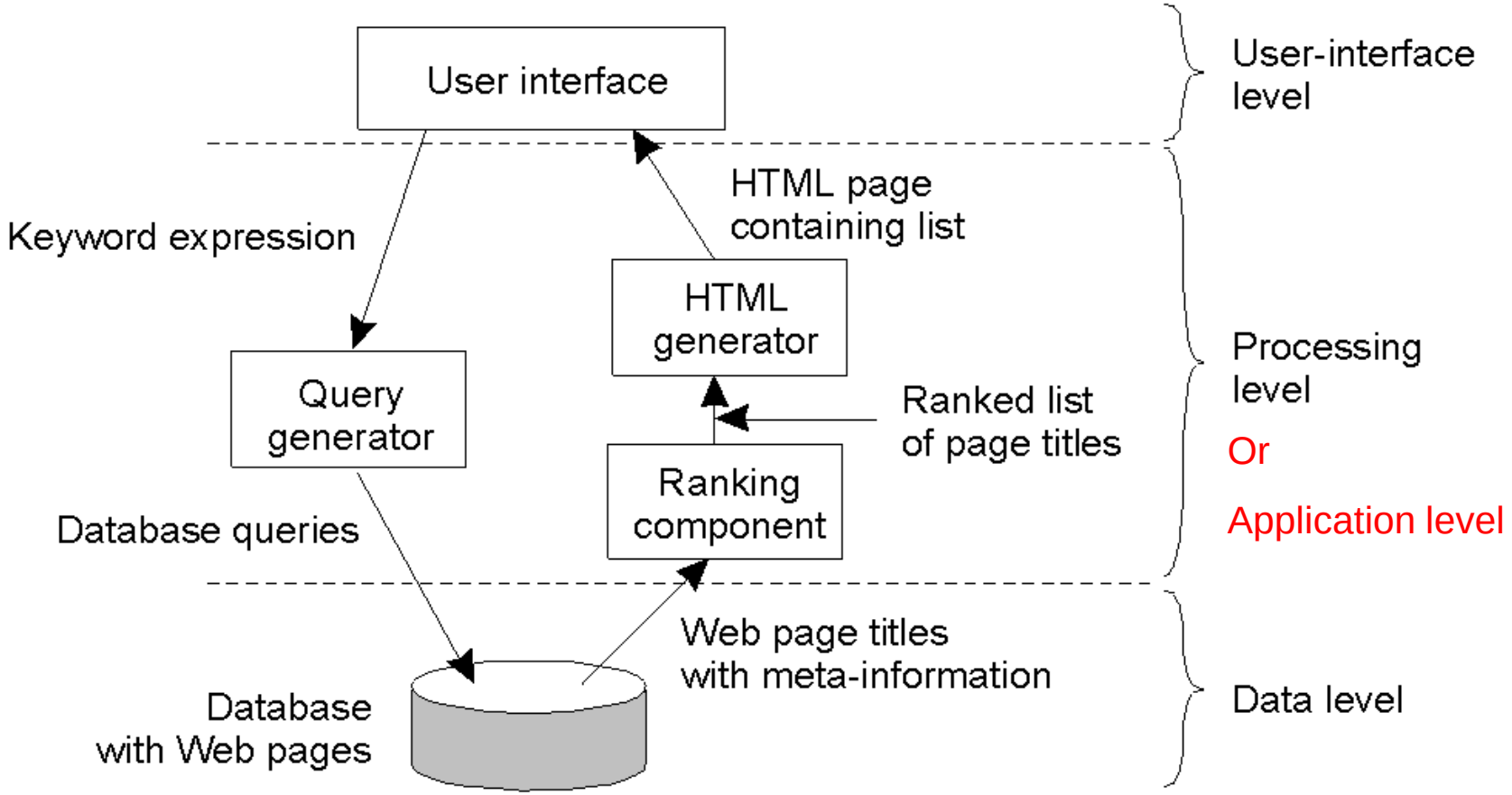
- Presentation layer
  - Concerned with presenting the results of a computation to system users and with collecting user inputs.
- Application processing layer
  - Concerned with providing application specific functionality e.g., in a banking system, banking functions such as open account, close account,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

Application layers



At Data management layer if the database is relational, separation is very clear but at OORDB, this layer has to play bigger role

Processing Level



The general organization of an Internet search engine into three different layers

Fig. Curtsey: Distributed Systems: Principles and Paradigms by Tannenbaum

Alternative client-server organizations

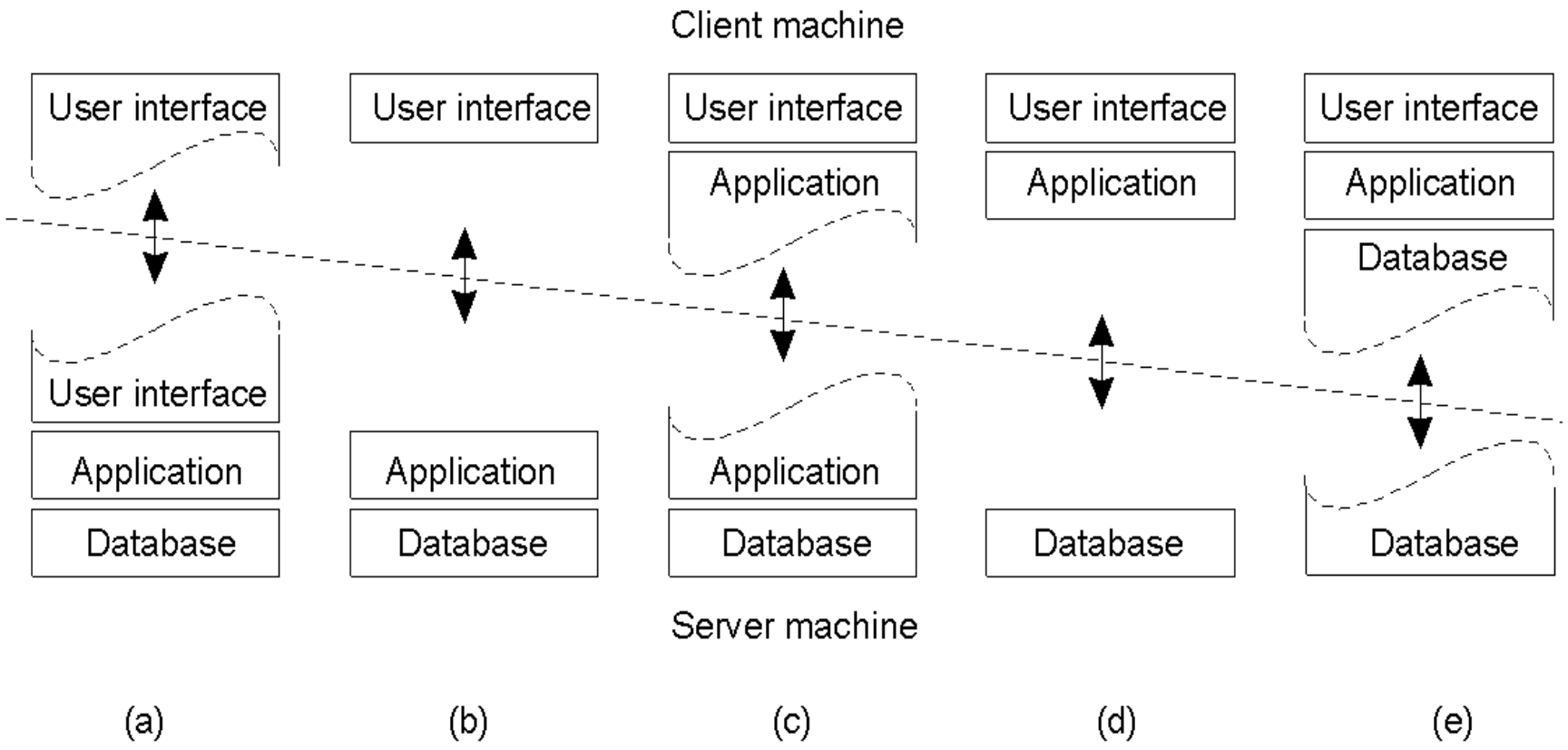
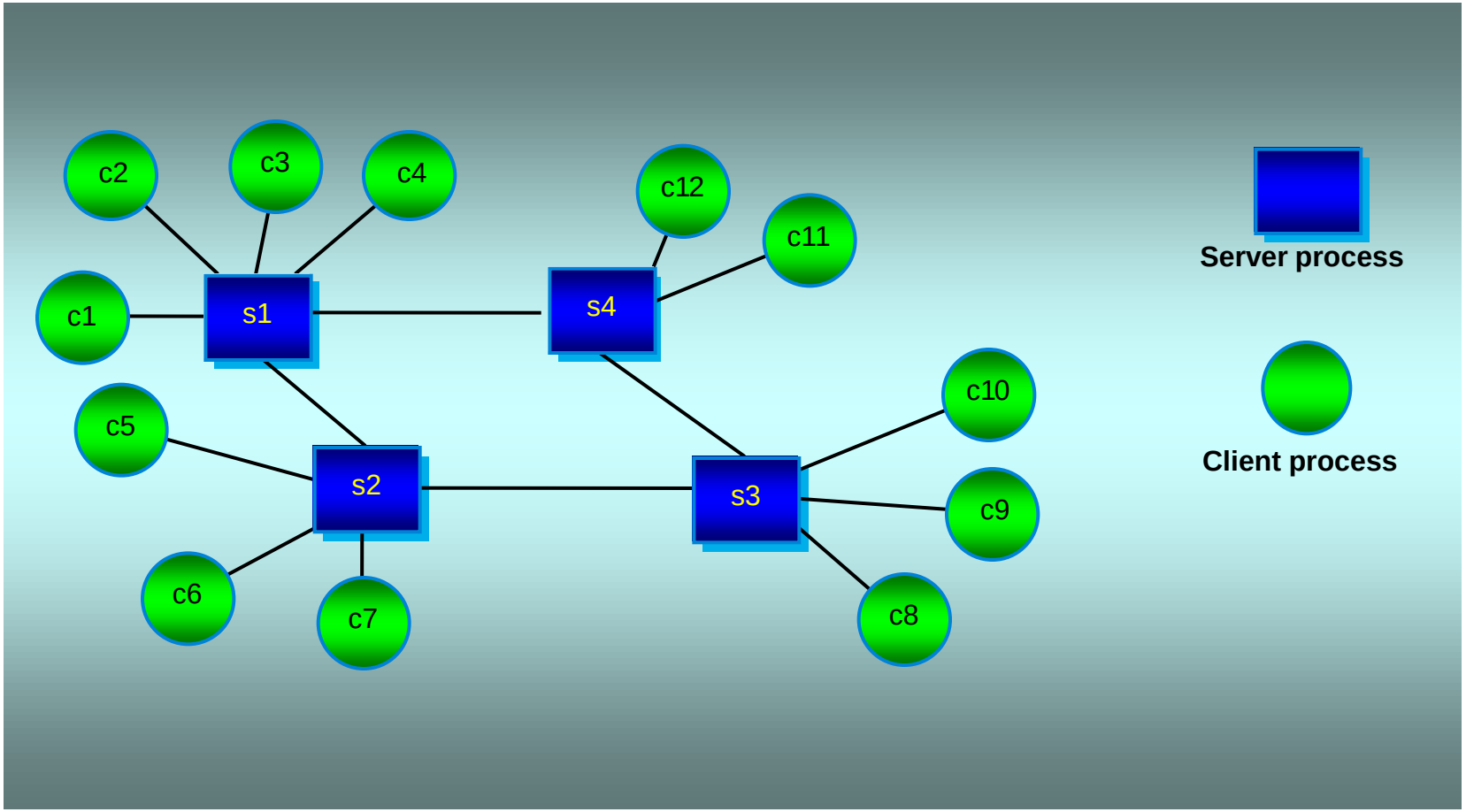


Fig. Curtsey: Distributed Systems: Principles and Paradigms by Tannenbaum

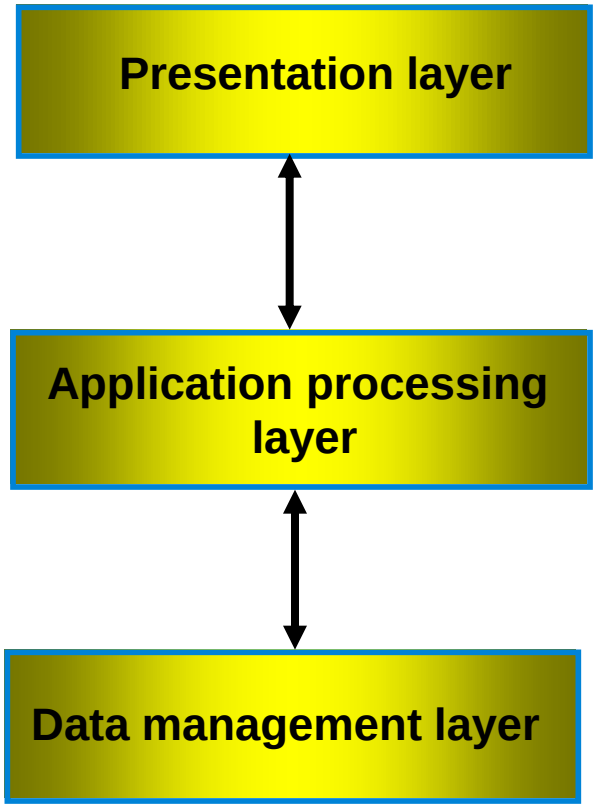
Client-server architectures

- 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
- The mapping of processors to processes is not necessarily 1 : 1.

A client-server system



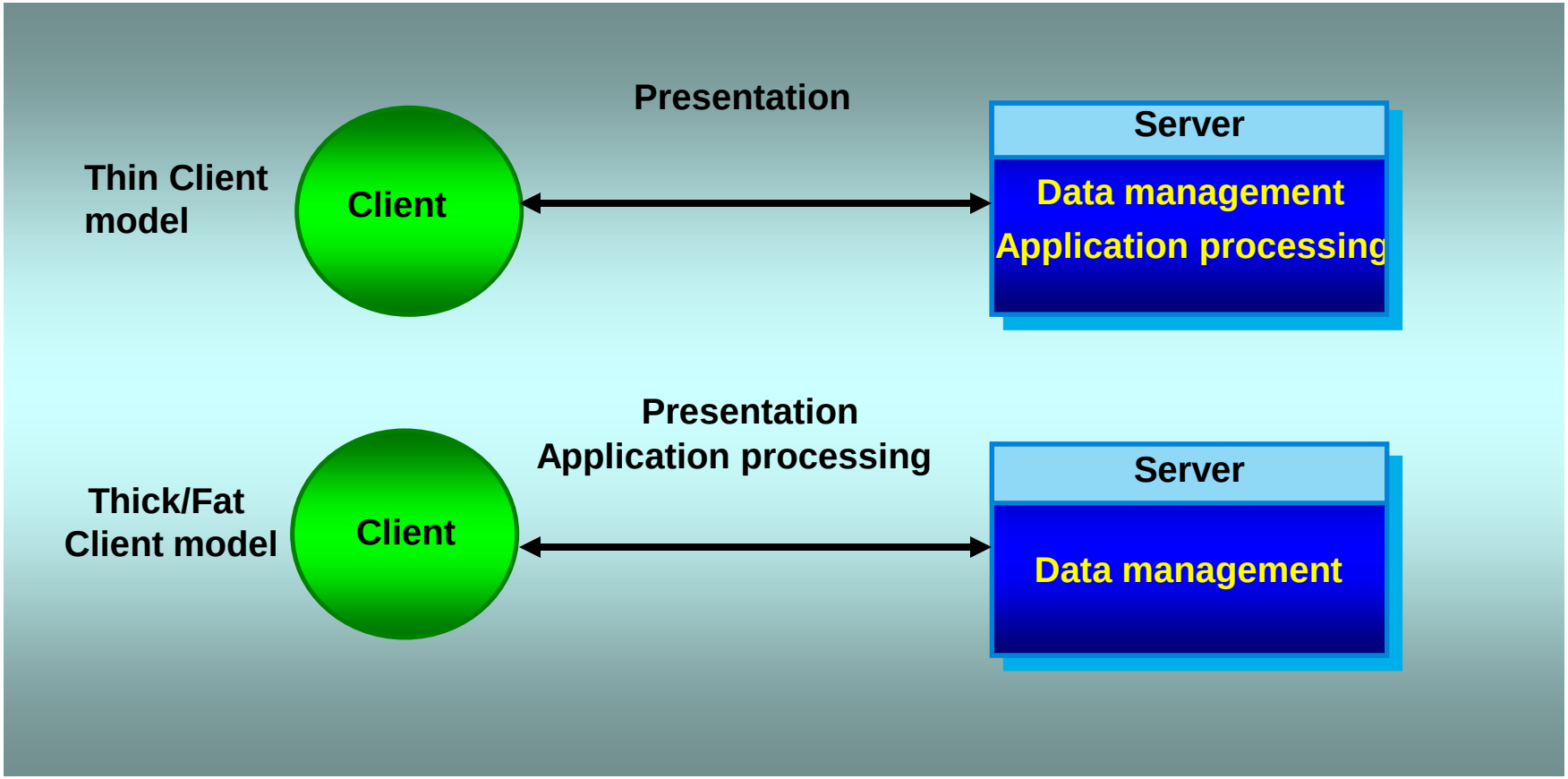
Application layers



Thin and Thick/Fat 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.
- Thick/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 Thick/Fat clients



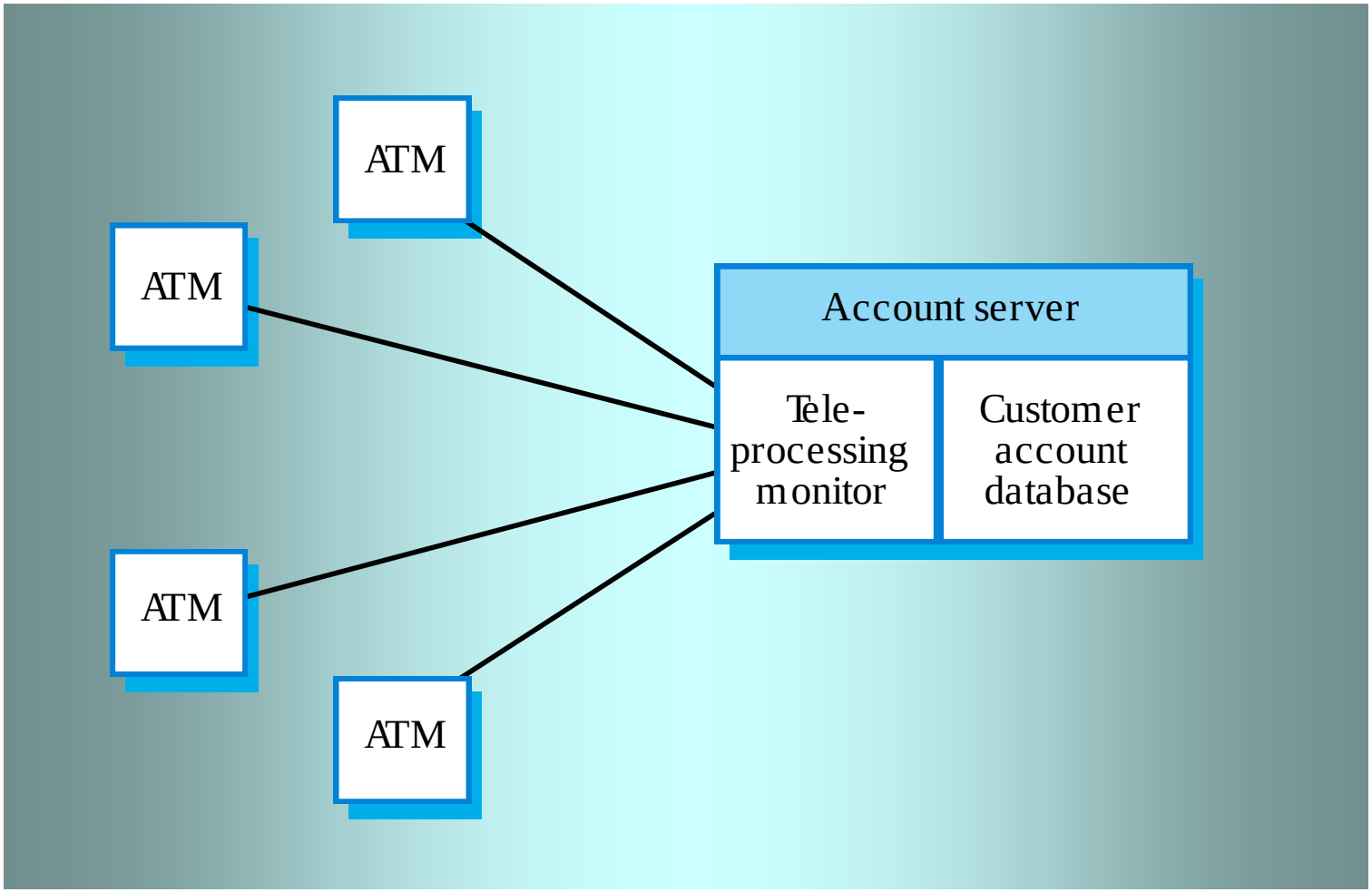
Thin client model

- Used when legacy systems are migrated to client server architectures.
  - The legacy system acts as a server in its own right with a graphical interface implemented on a client.
- A major disadvantage is that it places a heavy processing load on both the server and the network.

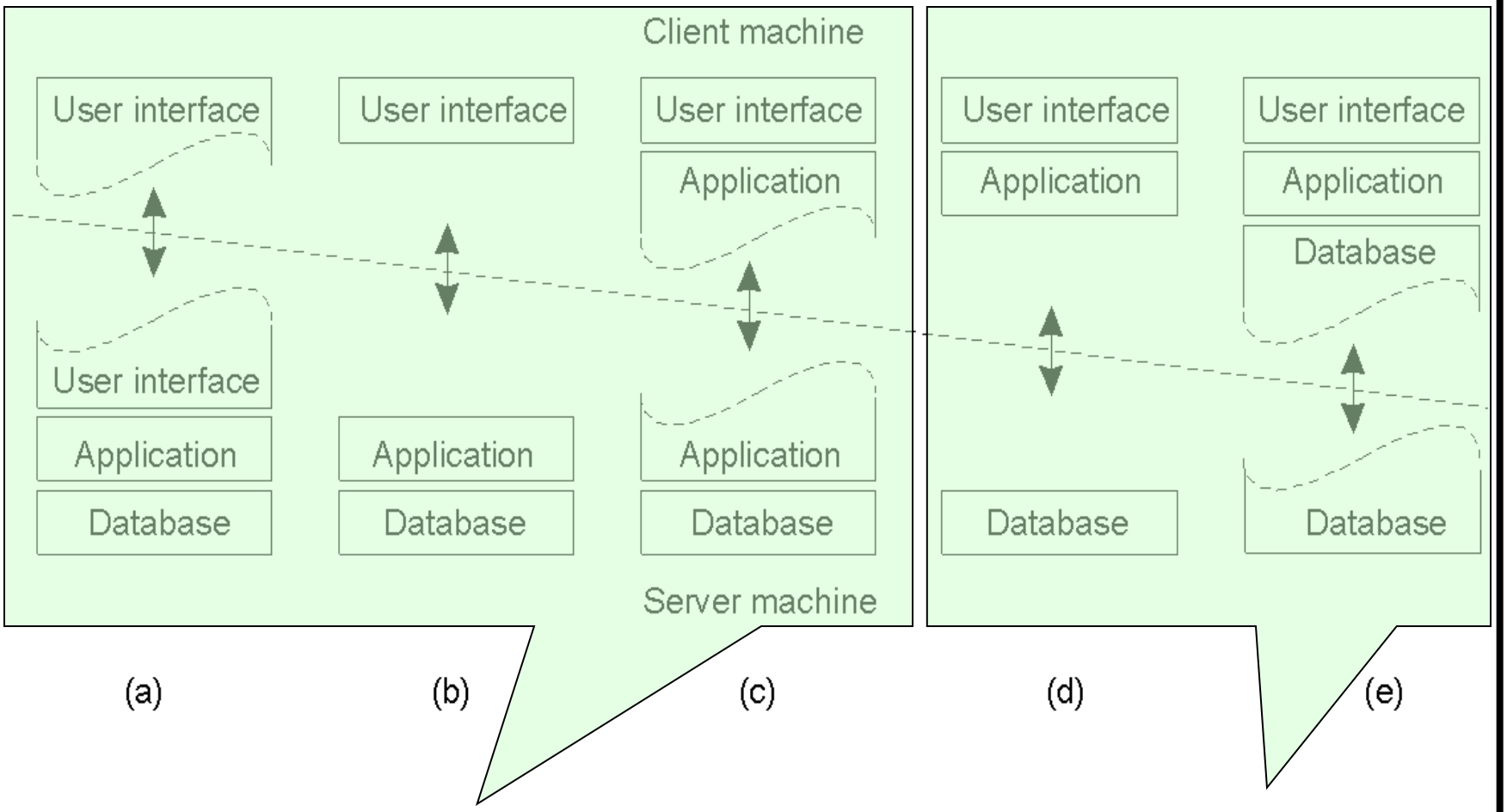
Thick/Fat client model

- More processing is delegated to the client as the application processing is locally executed.
- Most suitable for new C/S systems where the capabilities of the client system are known in advance.
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.

A client-server ATM system



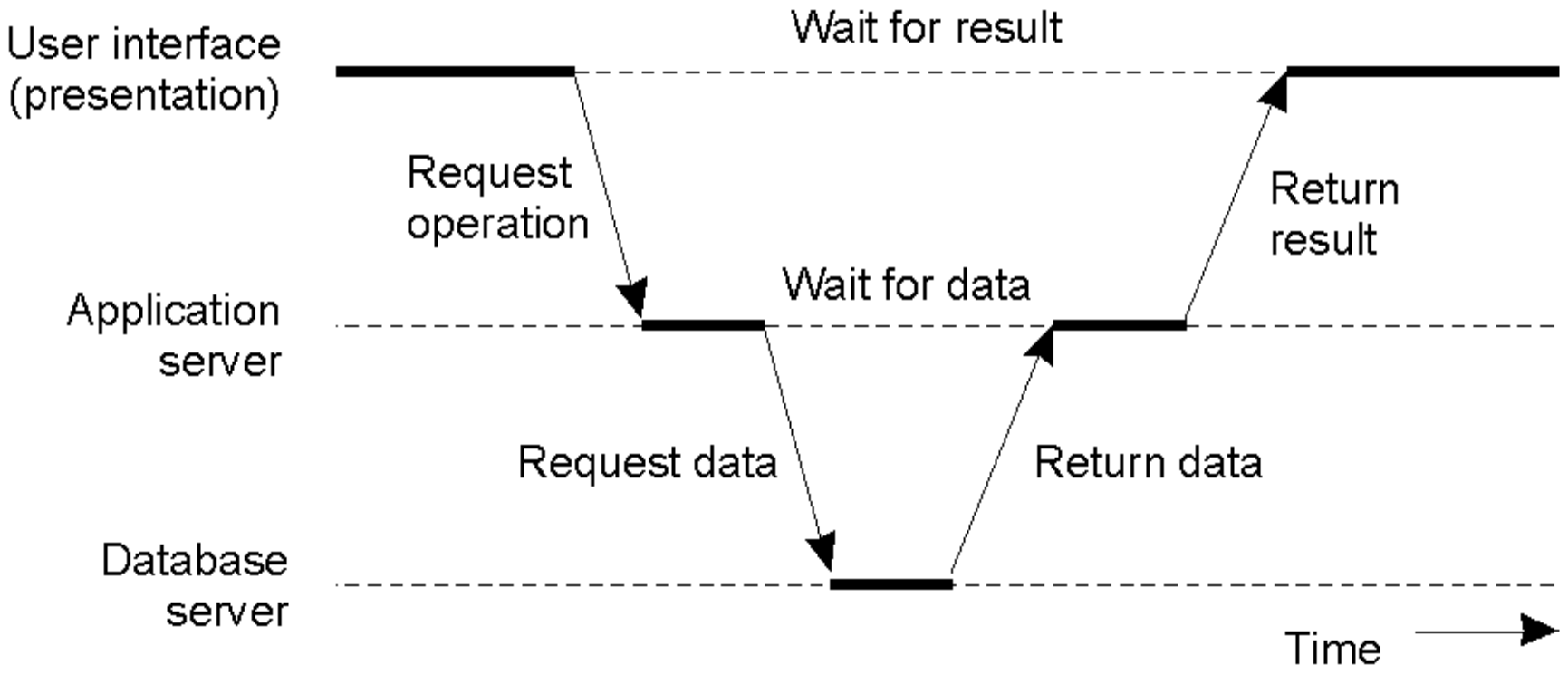
Alternative client-server organizations



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 thick-client approach.
- A more scalable architecture - as demands increase, extra servers can be added.

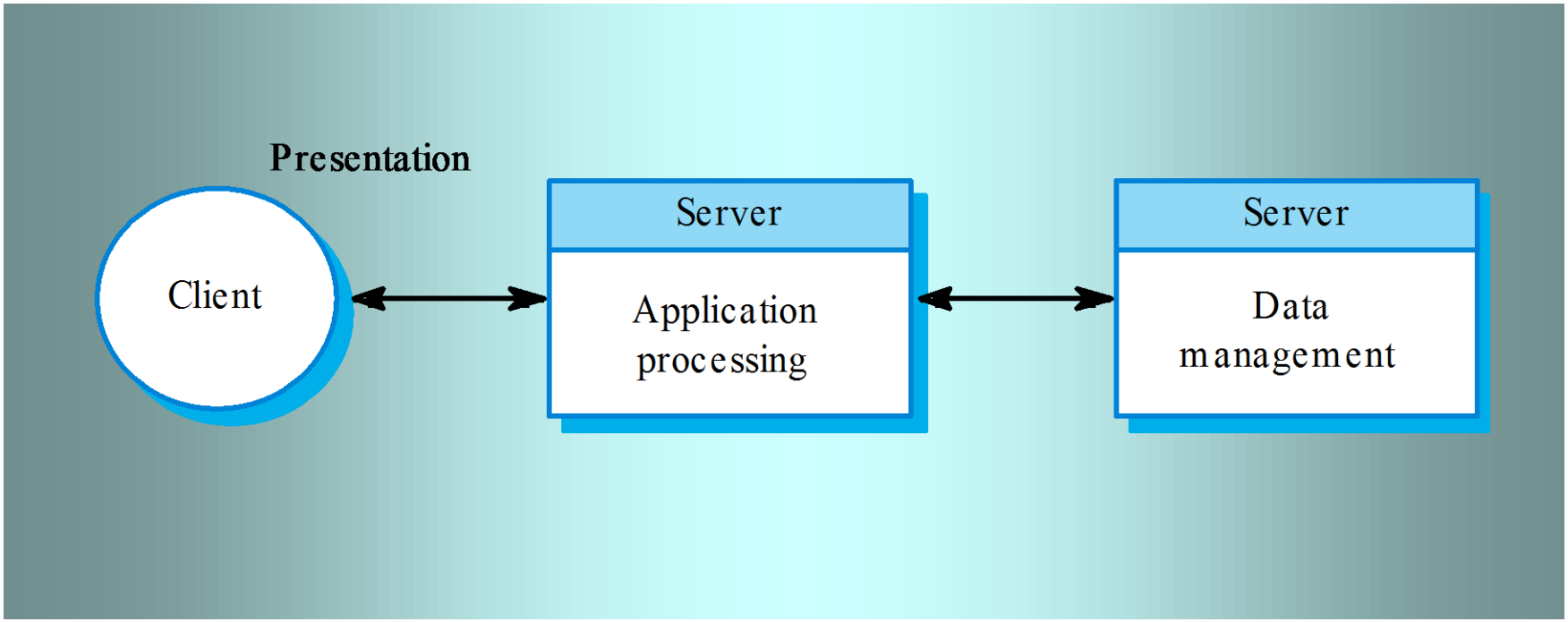
3-tiered Architectures



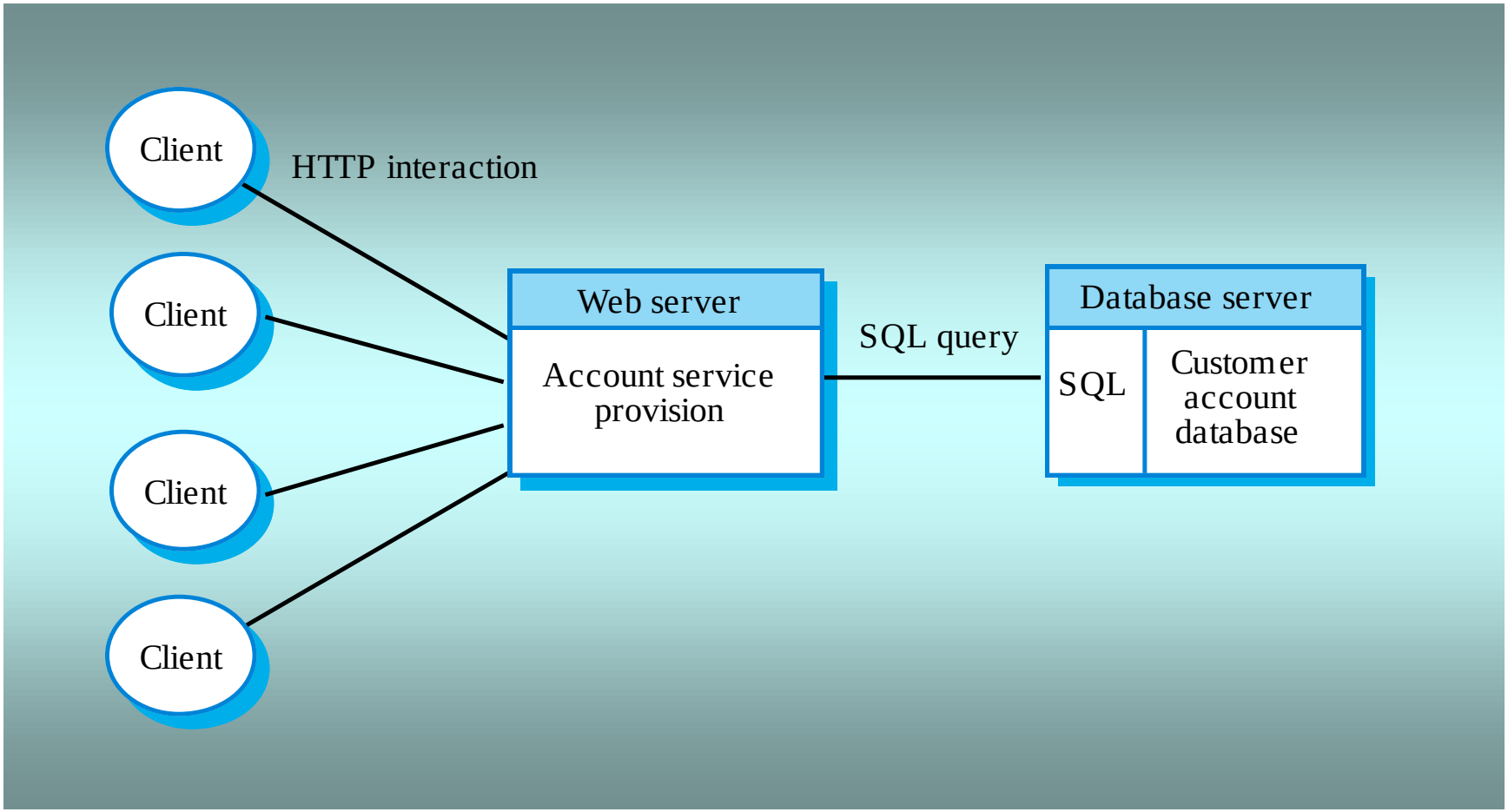
An example of an application server, acting as a client for the DB server

Fig. Curtsey: Distributed Systems: Principles and Paradigms by Tannenbaum

A 3-tier C/S architecture



An internet banking system



Where to use which C/S architectures?

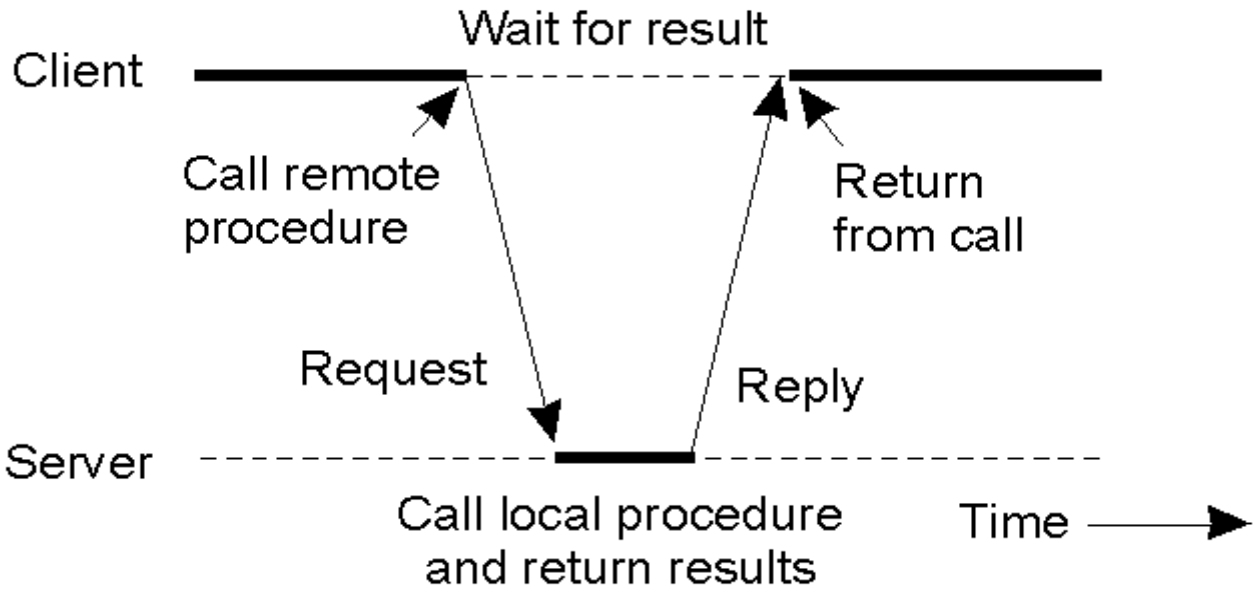
Architecture	Applications
Two-tier C/S architecture with **thin clients**	❖ Legacy system applications where separating application processing and data management is impractical. ❖ Computationally-intensive applications **such as compilers** with little or no data management. ❖ Data-intensive applications (**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.

Service-oriented architectures

- Based around the notion of externally provided services **(web services)**.
- A web service is a standard approach to making a reusable component available and accessible across the web
  - *A tax filing service could provide support for users to fill in their tax forms and submit these to the Income tax office.*

Client and Server Stubs

- Principle of **RPC** between a client and server program was proposed by Birrel and Nelson in 1984 which says: *"When a process on machine A call procedure B, calling process is suspended, execution on called procedure starts on B"*.

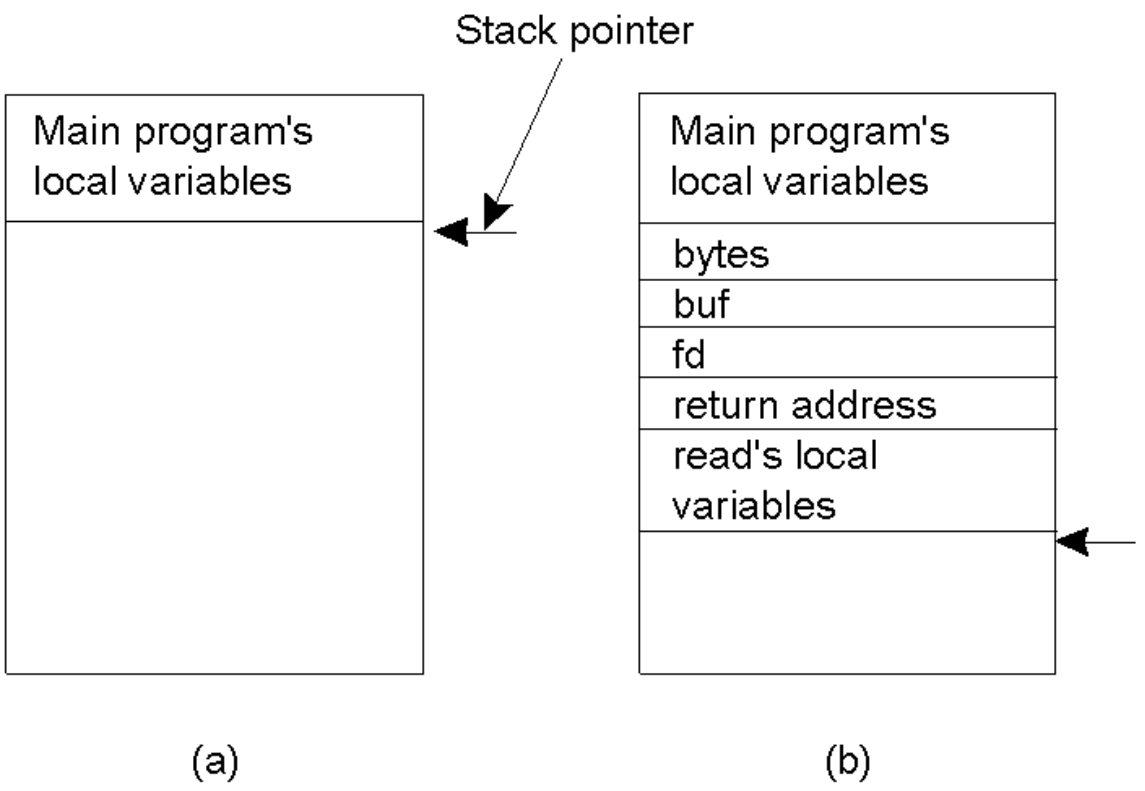


Andrew D. Birrell, Bruce Jay Nelson, Implementing remote procedure calls, Journal ACM Transactions on Computer Systems (TOCS) Volume 2 Issue 1, February 1984

Conventional Procedure Call

a) Parameter passing in a local procedure call: the stack before the call to read(fd, buf, bytes)

b) The stack while the called procedure is active

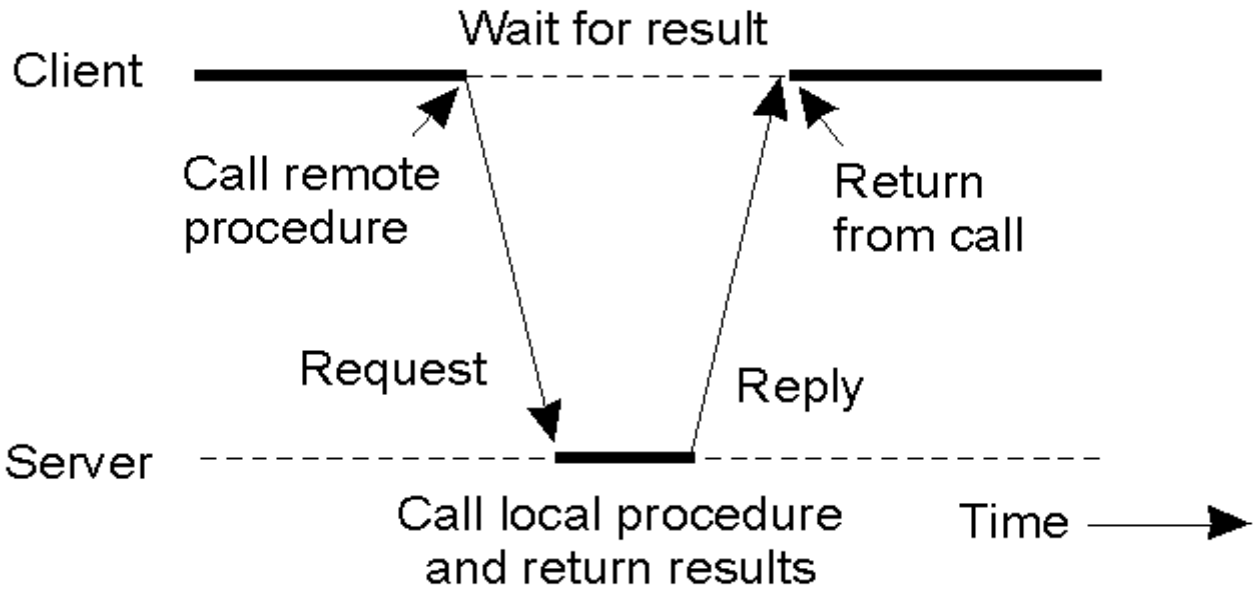


1. *Return value to a register*
2. *Removes the return address*
3. *Control is transferred to the caller*
4. *Caller removes parameter from the stack*

Fig. Curtsey: Distributed Systems: Principles and Paradigms by Tannenbaum

Client and Server Stubs

- Principle of **RPC** between a client and server program was proposed by Birrel and Nelson in 1984 which says: *"When a process on machine A call procedure B, calling process is suspended, execution on called procedure starts on B"*.



Andrew D. Birrell, Bruce Jay Nelson, Implementing remote procedure calls, Journal ACM Transactions on Computer Systems (TOCS) Volume 2 Issue 1, February 1984

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Passing Value Parameters & Marshaling

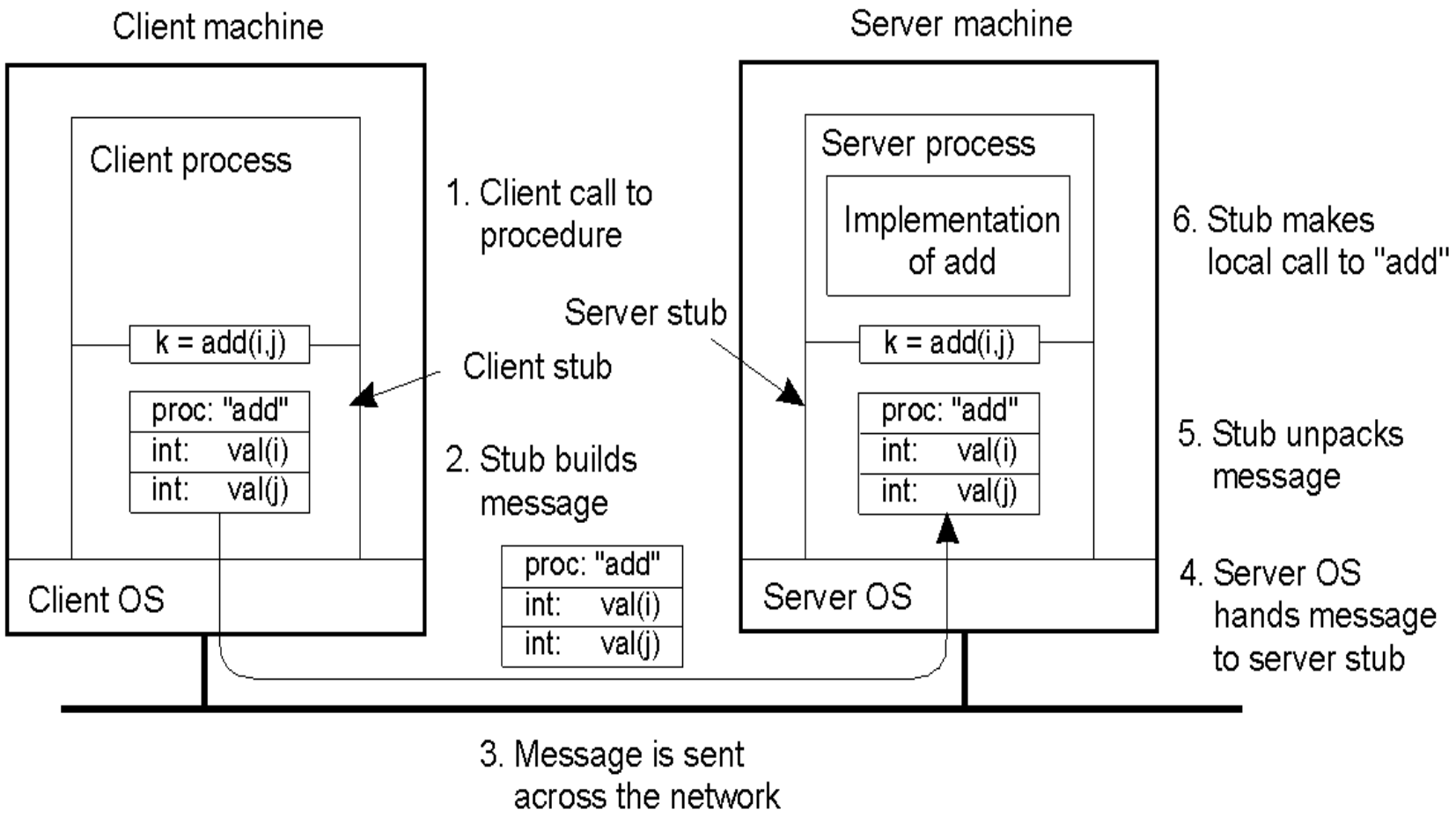


Fig. Curtsey: Distributed Systems: Principles and Paradigms by Tannenbaum

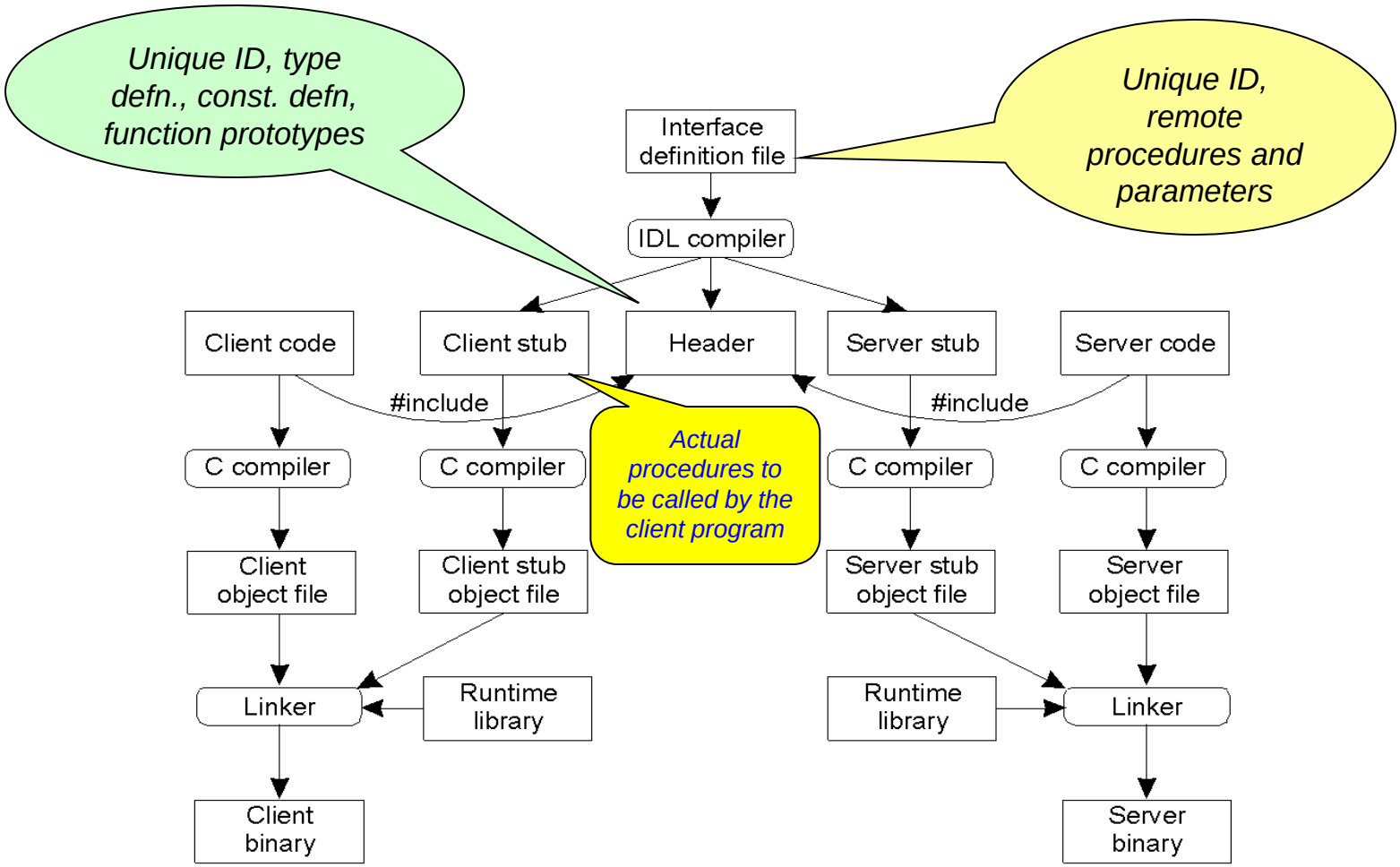
Interface definition language

- Once RPC is known, client and server stubs are needed to be implemented
- Stubs for the same protocol with different procedures differ only in their implementations
- Interface is a collection of procedures implemented by server & called by the client
- Interface is generally available in same programming language (not necessarily always)
- Interfaces are specified using **Interface Definition Language**
- Interfaces are then compiled into stub and skeleton

Interface definition language

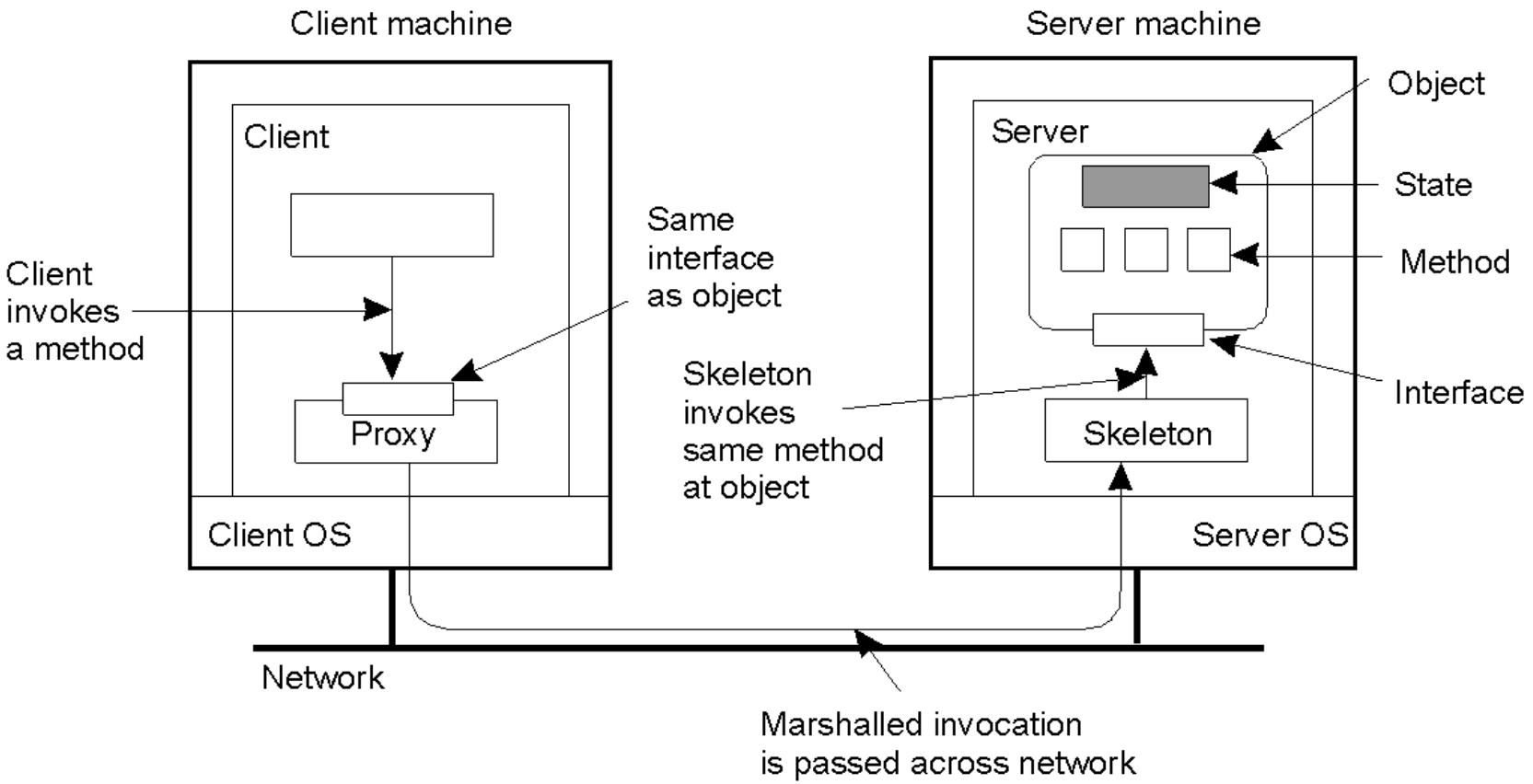
- Interface definition works as a glue that hold every thing
- It permits procedure declaration resembling prototype declaration in ANSI C
- Crucial element of every IDL is globally unique identifier for specified interface (normally 128 bit number represented as ASCII string in hexadecimal number)
- IDL file is edited for filling name of the remote procedure and parameter

Writing a Client and a Server



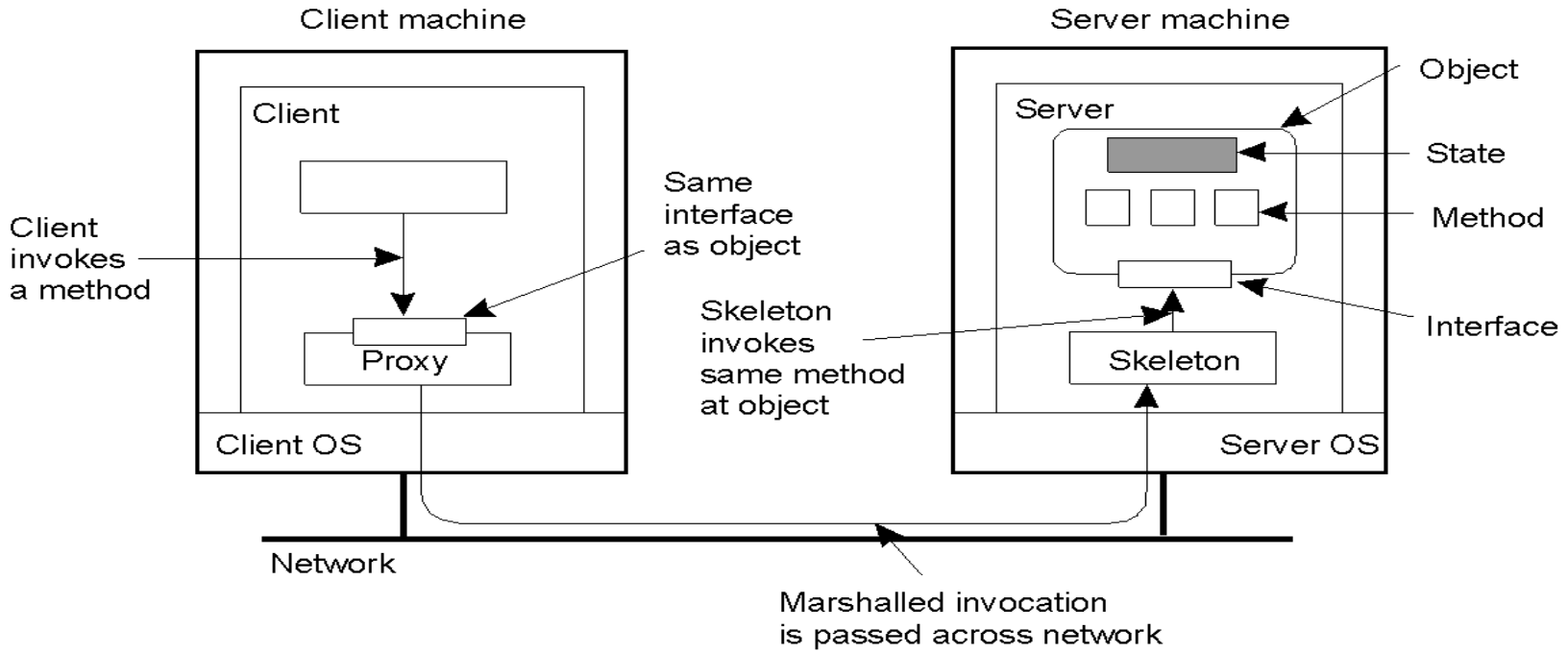
The steps in writing a client and a server in C

Distributed Objects in RPC way

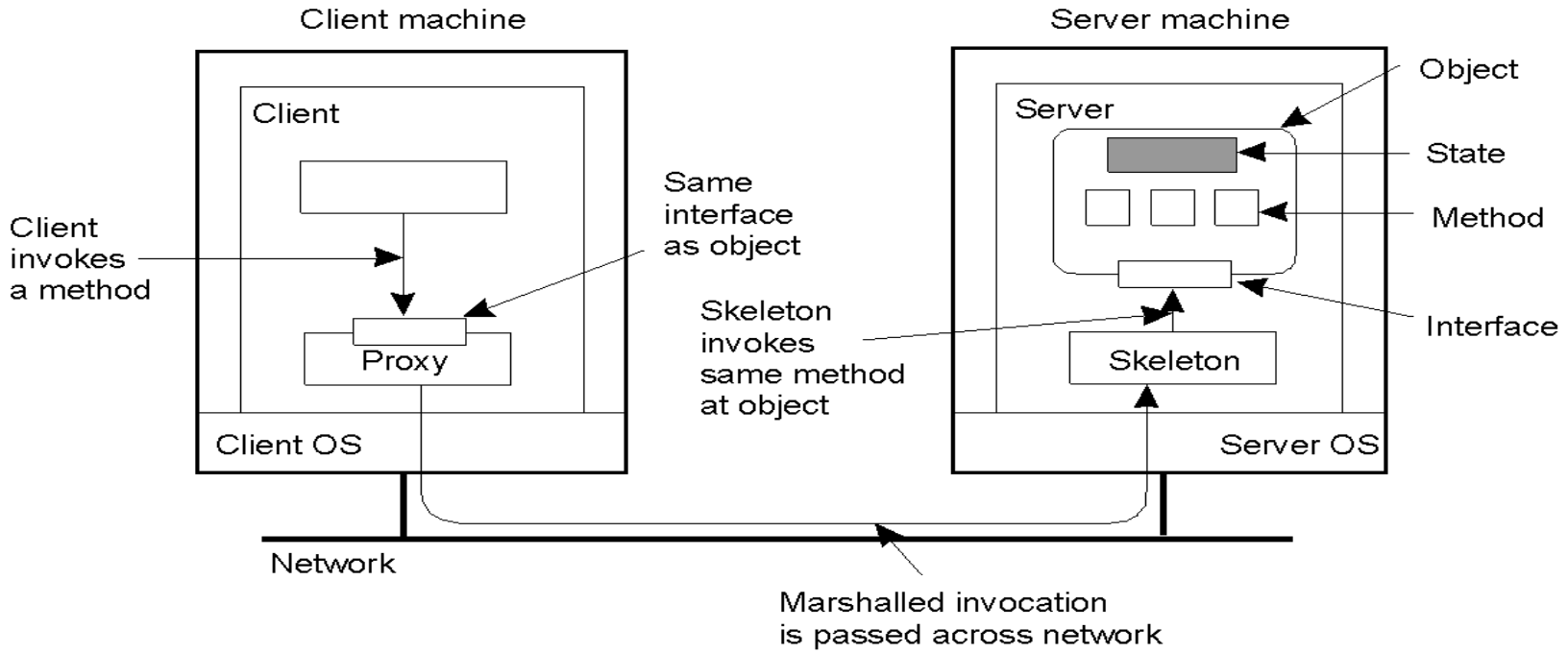


Separation between interface and the objects implementing these interfaces is crucial in distributed systems

Fig. Curtsey: Distributed Systems: Principles and Paradigms by Tannenbaum

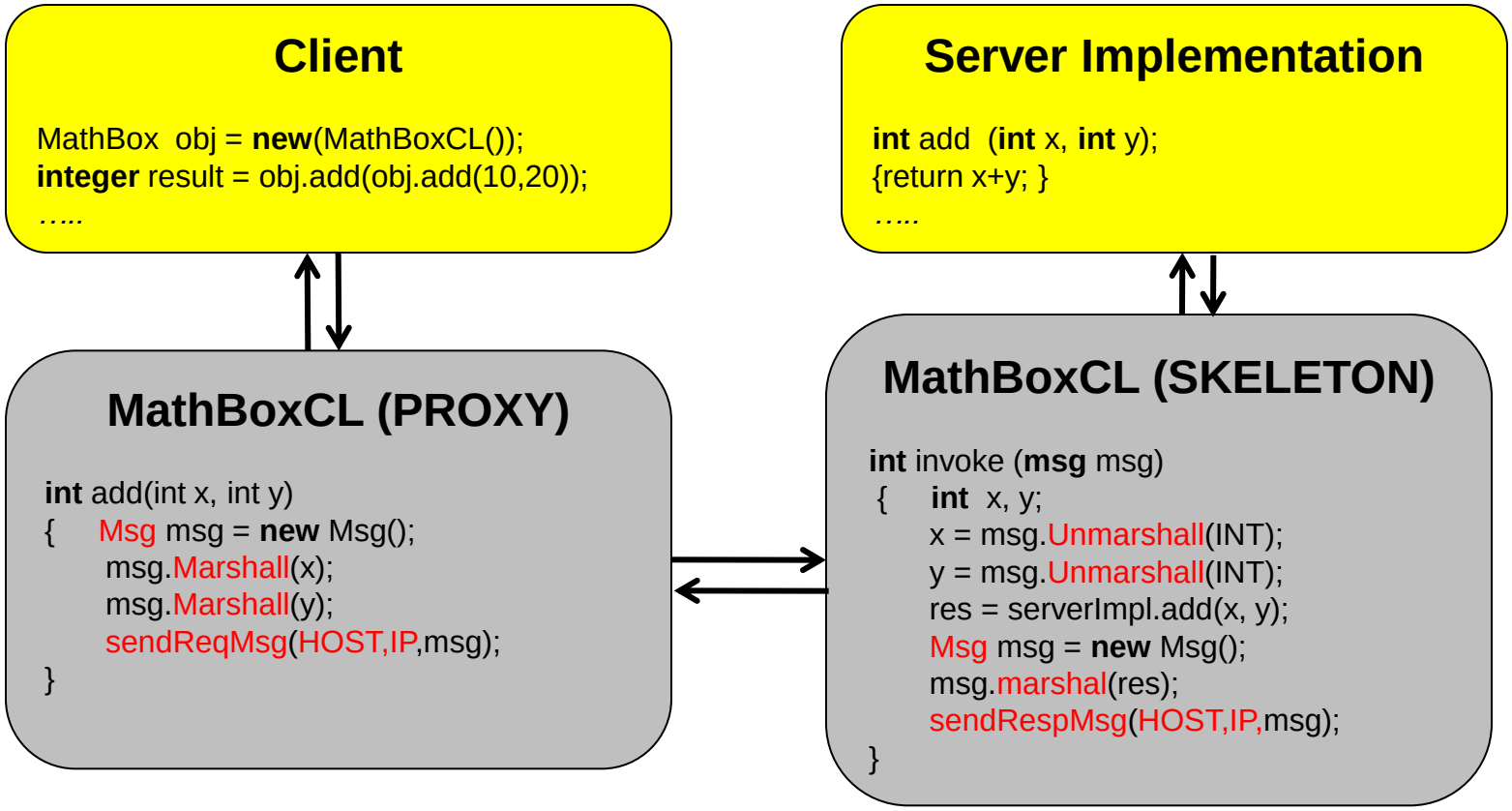


- Object may implement multiple interfaces
- When a client binds to a distributed object, an implementation of the object's interface called proxy is loaded in the client address space
- Proxy does marshalling of method invocation into message



- Actual objects resides at a server, where it offers the same interface
- Skeleton unmarshals the message into a method invocation at the object's interface at the server
- Such objects are referred to as the remote objects

Communication



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 scaleable.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required.

Common Object Request Broker Architecture

- CORBA is an international standard for an Object Request Broker - middleware to manage communications between distributed objects.

Middleware for distributed computing is required at two 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.

Berkeley Sockets

- Socket primitives for TCP/IP.

Primitive	**Meaning**
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

The Message-Passing Interface (MPI)

- Some of the most intuitive message-passing primitives of MPI.

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there are none
MPI_irecv	Check if there is an incoming message, but do not block

Programming Issues

Naming Convention for Classes, Variables & Methods

- Use full words – avoid abrvtns

Pascal Casing

- Capitalize the first character of each word
- **SomeClassName**

Camel Casing

- Capitalize the first character of each word except the first word
- **someVariableName**

Item	Java	Smalltalk	C#
Class	PascalCase	PascalCase	PascalCase
Method	camelCase	camelCase	PascalCase
Field	camelCase	camelCase	CamelCase
Parameter	camelCase	camelCase	camelCase
Local Variable	camelCase	camelCase	camelCase

Names

"Finding good names is the hardest part of OO Programming"

"Names should fully and accurately describe the entity the variable represents"

What role does the variable play in the program?

Data Structure Role, function

InputRec EmployeeData
BitFlag PrinterReady

Some Examples of Names, Good and Bad

TrainVelocity	Velt, V, X, Train
CurrentDate	CD, Current, C, X
LinesPerPage	LPP, Lines, L, x