



# PA160: Net-Centric Computing II.

## Distributed Systems

**Luděk Matyska**

**Slides by: Tomáš Rebok**

Spring 2025





# Lecture overview

## Distributed Systems

Key characteristics

Challenges and Issues

Distributed System Architectures

Inter-process Communication

## Middleware

Remote Procedure Calls (RPC)

Remote Method Invocation (RMI)

Common Object Request Broker Architecture (CORBA)

## Service Oriented Architecture (SAO)

## Web Services



# Lecture overview

## Distributed Systems

Key characteristics

Challenges and Issues

Distributed System Architectures

Inter-process Communication

## Middleware

Remote Procedure Calls (RPC)

Remote Method Invocation (RMI)

Common Object Request Broker Architecture (CORBA)

## Service Oriented Architecture (SAO)

## Web Services

## Distributed Systems – Definition

### Distributed System by Coulouris, Dollimore, and Kindberg

A system in which hardware and software components located at networked computers communicate and coordinate their actions only by message passing.

### Distributed System by Tanenbaum and Steen

A collection of independent computers that appears to its users as a single coherent system.

- the independent/autonomous machines are interconnected by communication networks and equipped with software systems designed to produce an integrated and consistent computing environment

**Core objective of a distributed system: resource sharing**

## Distributed Systems – Key characteristics

- **Autonomy** – there are several autonomous computational entities, each of which has its own local memory
- **Heterogeneity** – the entities may differ in many ways
  - computer HW (different data types' representation), network interconnection, operating systems (different APIs), programming languages (different data structures), implementations by different developers, etc.
- **Concurrency** – concurrent (distributed) program execution and resource access
- **No global clock** – programs (distributed components) coordinate their actions by exchanging messages
  - message communication can be affected by delays, can suffer from variety of failures, and is vulnerable to security attacks
- **Independent failures** – each component of the system can fail independently, leaving the others still running (and possibly not informed about the failure)
  - How to know/differ the states when a network has failed or became unusually slow?
  - How to know immediately if a remote server crashed?

# Distributed Systems – Challenges and Issues

What do we want from a Distributed System (DS)?

- *Resource Sharing*
- *Openness*
- *Concurrency*
- *Scalability*
- *Fault Tolerance*
- *Security*
- *Transparency*

# Distributed Systems – Challenges and Issues

## Resource Sharing

- main motivating factor for constructing DSs
- it should be easy for the users (and applications) to access remote resources, and to share them in a *controlled and efficient way*
  - each resource must be managed by a software that provides interfaces which enable the resource to be manipulated by clients
  - *resource* = anything you can imagine (e.g., storage facilities, data, files, Web pages, etc.)

# Distributed Systems – Challenges and Issues

## Openness

- whether the system can be extended and re-implemented in various ways and new resource-sharing services can be added and made available for use by a variety of client programs
  - specification and documentation of key software interfaces must be published
    - using an *Interface Definition Language (IDL)*
- involves HW extensibility as well
  - i.e., the ability to add hardware from different vendors



# Distributed Systems – Challenges and Issues

## Concurrency

- every resource in a DS must be designed to be safe in a concurrent environment
  - applies not only to servers, but to objects in applications as well
- ensured by standard techniques, like *semaphores*

# Distributed Systems – Challenges and Issues

## Scalability

- a DS is scalable if the cost of adding a user (or resource) is a constant amount in terms of resources that must be added
  - and is able to utilize the extra hardware/software *efficiently*
  - and remains manageable

<i>Date</i>	<i>Computers</i>	<i>Web servers</i>
1979, Dec.	188	0
1989, July	130,000	0
1999, July	56,218,000	5,560,866
2003, Jan	171,638,297	35,424,956

# Distributed Systems – Challenges and Issues

## Fault Tolerance

- a characteristic where a distributed system provides an appropriately handling of errors that occurred in the system
  - the failures can be *detected* (sometimes hard or even impossible), *masked* (made hidden or less severe), or *tolerated*
- achieved by deploying two approaches: *hardware redundancy* and *software recovery*

## Security

- involves *confidentiality*, *integrity*, *authentication*, and *availability*

# Distributed Systems – Challenges and Issues

## Transparency

- certain aspects of the DS should be made invisible to the user / application programmer
  - i.e., the system is perceived as a whole rather than a collection of independent components
- several *forms of transparency*:
  - **Access transparency** – enables local and remote resources to be accessed using identical operations
  - **Location transparency** – enables resources to be accessed without knowledge of their location
  - **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 / application programmers

# Distributed Systems – Challenges and Issues

## Transparency II.

### ■ *forms of transparency cont'd.:*

- **Failure transparency** – enables the concealment of faults, allowing users and application programs to complete their tasks despite of a failure of HW/SW components
- **Mobility/migration 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 changes to the system structure or application algorithms



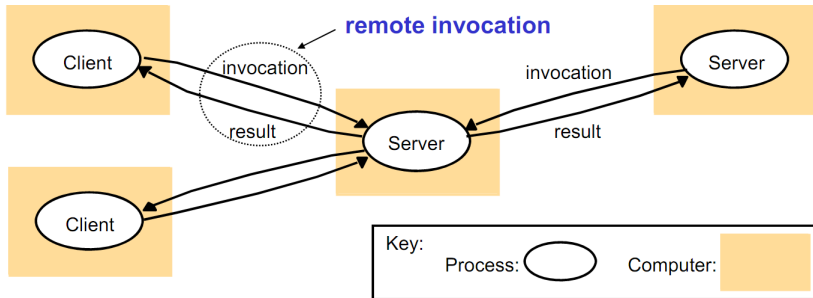
## Distributed Systems – Architecture Models

- defines the way in which the components of systems interact with one another, and
- defines the way in which the components are mapped onto an underlying network of computers
- the overall goal is to ensure that the structure will meet present and possibly future demands
  - the major concerns are to make system *reliable, manageable, adaptable, and cost-effective*
- principal architecture models:
  - **client-server model** – most important and most widely used
    - a service may be further provided by *multiple servers*
    - the servers may in turn be clients for another servers
    - *proxy servers* (caches) may be employed to increase availability and performance
  - **peer processes** – all the processes play similar roles
    - based either on *structured* (Chord, CAN, etc.), *unstructured*, or *hybrid* architectures



# Distributed Systems – Architecture Models

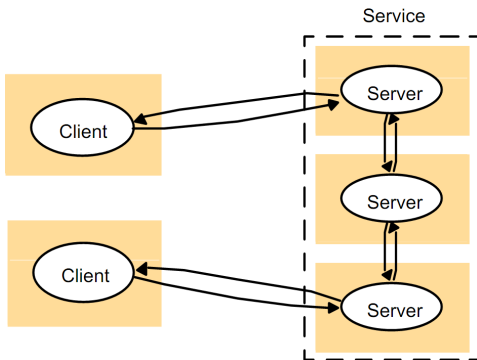
## Client-Server model





# Distributed Systems – Architecture Models

## Client-Server model – A Service provided by Multiple Servers

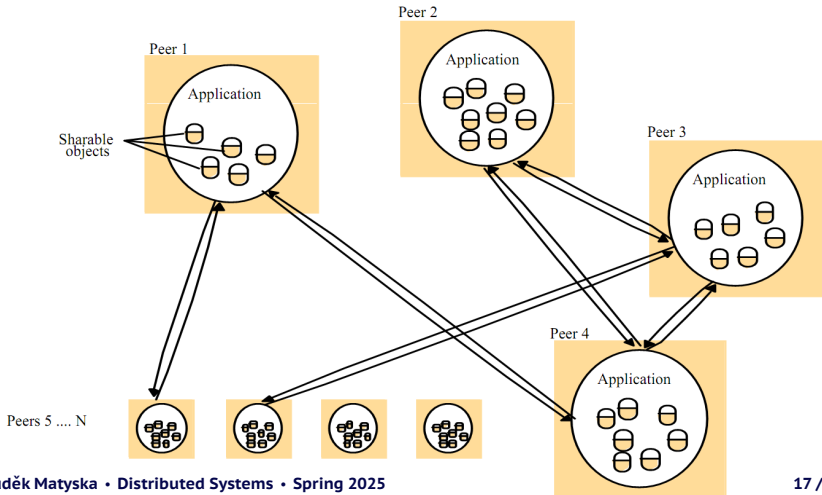






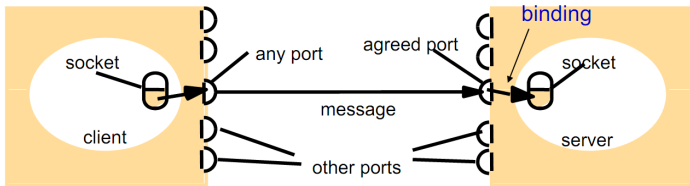
# Distributed Systems – Architecture Models

## Peer processes



# Distributed Systems – Inter-process Communication (IPC)

- the processes (components) need to communicate
- the communication may be:
  - *synchronous* – both send and receive are blocking operations
  - *asynchronous* – send is non-blocking and receive can have blocking (more common) and non-blocking variants
- the simplest forms of communication: **UDP** and **TCP sockets**





# Distributed Systems – Inter-process Communication

## UDP and TCP sockets

### UDP/TCP sockets

- provide unreliable/reliable communication services
- + the complete control over the communication lies in the hands of applications
- too primitive to be used in developing a distributed system software
  - higher-level facilities (marshalling/unmarshalling data, error detection, error recovery, etc.) must be built from scratch by developers on top of the existing socket primitive facilities
  - force read/write mechanism instead of a *procedure call*
- another problem arises when the software needs to be used in a platform different from where it was developed
  - the target platform may provide different socket implementation

⇒ *these issues are eliminated by the use of a **Middleware***

# Lecture overview

## Distributed Systems

Key characteristics

Challenges and Issues

Distributed System Architectures

Inter-process Communication

## Middleware

Remote Procedure Calls (RPC)

Remote Method Invocation (RMI)

Common Object Request Broker Architecture (CORBA)

## Service Oriented Architecture (SAO)

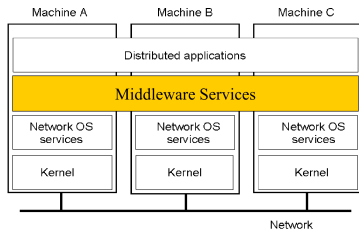
## Web Services

# Middleware

- a software layer that provides a programming abstraction as well as masks the heterogeneity of the underlying networks, hardware, operating systems, and programming languages
  - ⇒ provides *transparency services*
  - represented by processes/objects that interact with each other to implement communication and resource sharing support
  - provides building blocks for the construction of SW components that can work with one another

## ■ middleware examples:

- Sun RPC (ONC RPC)
- DCE RPC
- MS COM/DCOM
- Java RMI
- CORBA
- etc.



## Middleware – Basic Services

- *Directory services* – services required to locate application services and resources, and route messages
  - $\approx$  service discovery
- *Data encoding services* – uniform data representation services for dealing with incompatibility problems on remote systems
  - e.g., Sun XDR, ISO's ASN.1, CORBA's CDR, XML, etc.
  - data marshalling/unmarshalling
- *Security services* – provide inter-application client-server security mechanisms
- *Time services* – provide a universal format for representing time on different platforms (possibly located in various time zones) in order to keep synchronisation among application processes
- *Transaction services* – provide transaction semantics to support commit, rollback, and recovery mechanisms



# Middleware

## Basic Services – A Need for Data Encoding Services

**Data encoding services** are required, because remote machines may have:

- different byte ordering
- different sizes of integers and other types
- different floating point representations
- different character sets
- alignment requirements

```
main() {  
    unsigned int n;  
    char *a = (char *)&n;  
  
    n = 0x11223344;  
    printf("%02x, %02x, %02x, %02x\n",  
           a[0], a[1], a[2], a[3]);  
}
```

Output on a Pentium:  
44, 33, 22, 11

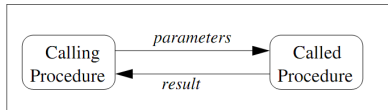
Output on a G4:  
11, 22, 33, 44

## Remote Procedure Calls (RPC)

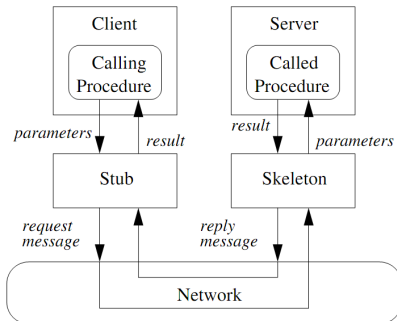
- very simple idea similar to a well-known procedure call mechanism
  - a client sends a request and blocks until a remote server sends a response
- the goal is to allow distributed programs to be written in the same style as conventional programs for centralised computer systems
  - while being *transparent* – the programmer need not be aware that the called procedure is executing on a local or a remote computer
- the idea:
  - the remote procedure is represented as a **stub on the client side**
    - behaves like a local procedure, but rather than placing the parameters into registers, it packs them into a message, issues a send primitive, and blocks itself waiting for a reply
  - the server passes the arrived message to a **server stub** (known as **skeleton** as well)
    - the skeleton unpacks the parameters and calls the procedure in a conventional manner
    - the results are returned to the skeleton, which packs them into a message directed to the client stub



## Remote Procedure Calls (RPC)



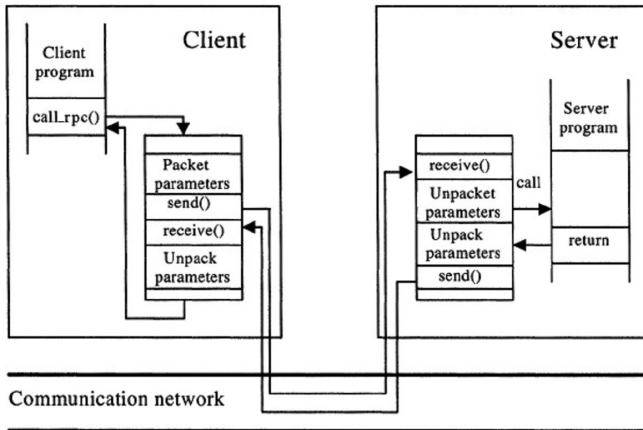
**Local Procedure Call**



**Remote Procedure Call**

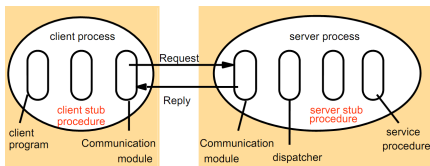
# Remote Procedure Calls (RPC)

## The remote procedure call in detail



# Remote Procedure Calls (RPC)

## Components



- client program, client stub
- *communication modules*
- server stub, service procedure
- *dispatcher* – selects one of the server stub procedures according to the procedure identifier in the request message

*Sun RPC*: the procedures are identified by:

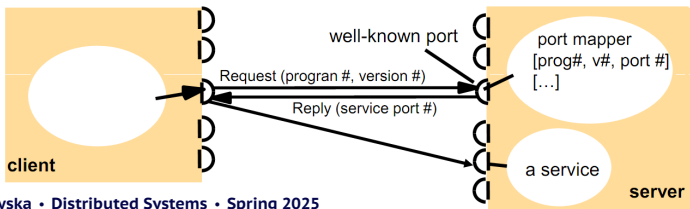
- *program number* – can be obtained from a central authority to allow every program to have its own unique number
- *procedure number* – the identifier of the particular procedure within the program
- *version number* – changes when a procedure signature changes

# Remote Procedure Calls (RPC)

## Location Services – Portmapper

Clients need to know the port number of a service provided by the server  $\Rightarrow$  **Portmapper**

- a server registers its *program#*, *version#*, and *port#* to the local portmapper
- a client finds out the *port#* by sending a request
  - the portmapper listens on a *well-known port* (111)
  - the particular procedure required is identified in the subsequent procedure call



# Remote Procedure Calls (RPC)

## Parameters passing

How to pass parameters to remote procedures?

- **pass by value** – easy: just copy data to the network message
- **pass by reference** – makes no sense without shared memory

**Pass by reference:** the steps

1. copy referenced items (marshalled) to a message buffer
2. ship them over, unmarshal data at server
3. pass local pointer to server stub function
4. send new values back

■ *to support complex structures:*

- copy the structure into pointerless representation
- transmit
- reconstruct the structure with local pointers on the server



## Remote Procedure Calls (RPC)

### Parameters passing – eXternal Data Representation (XDR)

*Sun RPC*: to avoid compatibility problems, the **eXternal Data Representation (XDR)** is used

- XDR primitive functions examples:
  - `xdr_int()`, `xdr_char()`, `xdr_u_short()`, `xdr_bool()`,  
`xdr_long()`, `xdr_u_int()`, `xdr_wrapstring()`,  
`xdr_short()`, `xdr_enum()`, `xdr_void()`
- XDR aggregation functions:
  - `xdr_array()`, `xdr_string()`, `xdr_union()`, `xdr_vector()`,  
`xdr_opaque()`
- *only a single input parameter is allowed in a procedure call*
  - $\Rightarrow$  procedures requiring multiple parameters must include them as components of a single structure

# Remote Procedure Calls (RPC)

## When Things Go Wrong I.

- local procedure calls do not fail
  - if they core dump, entire process dies
- there are more opportunities for errors with RPC
  - server could generate an error
  - problems in network (lost/delayed requests/replies)
  - server crash
  - client might crash while server is still executing code for it
- transparency breaks here
  - applications should be prepared to deal with RPC failures

Semantics of local procedure calls: **exactly once**

- difficult to achieve with RPC

# Remote Procedure Calls (RPC)

## When Things Go Wrong II.

Four remote calls semantics available in RPC:

- **at-least-once semantic**

- client keeps trying sending the message until a reply has been received

- failure is assumed after  $n$  re-sends

- guarantees that the call has been made “at least once”, but possibly multiple times

- ideal for *idempotent operations*

- **at-most-once semantic**

- client gives up immediately and reports back a failure

- guarantees that the call has been made “at most once”, but possibly none at all

- **exactly-once semantic**

- the most desirable, but the most difficult to implement

- **maybe semantic**

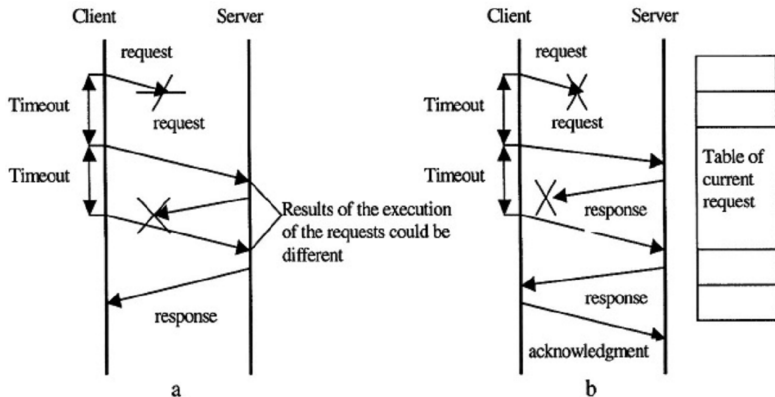
- no message delivery guarantees are provided at all

- (easy to implement)



# Remote Procedure Calls (RPC)

## When Things Go Wrong III.



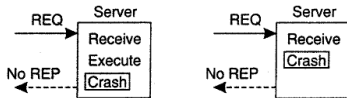
**Figure:** Message-passing semantics. (a) at-least-once; (b) exactly-once.

# Remote Procedure Calls (RPC)

## When Things Go Wrong IV. – Complications

- it is necessary to understand the application
  - idempotent functions* – in the case of a failure, the message may be retransmitted and re-run without a harm
  - non-idempotent functions* – has side-effects  $\Rightarrow$  the retransmission has to be controlled by the server
    - the duplicity request (retransmission) has to be detected
    - once detected, the server procedure is NOT re-run; just the results are resent (if available in a server cache)

- in the case of a *server crash*, the order of execution vs. crash matters



- in the case of a *client crash*, the procedure keeps running on the server
  - consumes resources (e.g., CPU time), possesses resources (e.g., locked files), etc.
  - may be overcome by employing *soft-state principles*
    - keep-alive messages

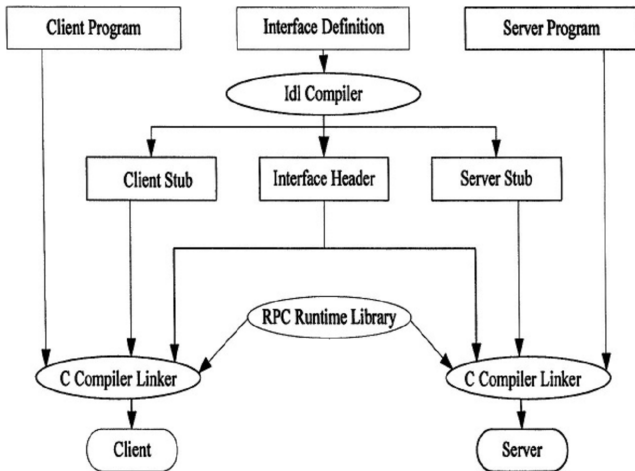
# Remote Procedure Calls (RPC)

## Code Generation I.

- RPC drawbacks:
  - complex API, not easy to debug
  - the use of XDR is difficult
- *but, it's often used in a similar way*
- ⇒ the server/client code can be automatically generated
  - assumes well-defined interfaces (IDL)
  - the application programmer has to supply the following:
    - *interface definition file* – defines the interfaces (data structures, procedure names, and parameters) of the remote procedures that are offered by the server
    - *client program* – defines the user interfaces, the calls to the remote procedures of the server, and the client side processing functions
    - *server program* – implements the calls offered by the server
  - compilers:
    - rpcgen for C/C++, jrpcgen for Java

# Remote Procedure Calls (RPC)

## Code Generation II.



## Remote Method Invocation (RMI)

- as the popularity of object technology increased, techniques were developed to allow calls to remote objects instead of remote procedures only

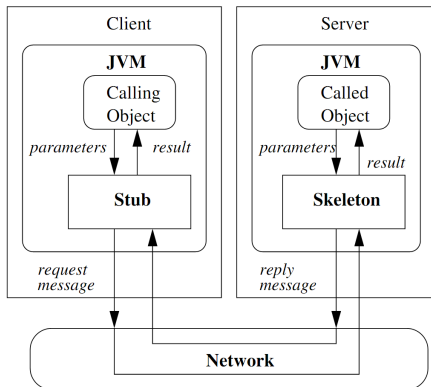
### ⇒ Remote Method Invocation (RMI)

- essentially the same as the RPC, except that it operates on objects instead of applications/procedures
  - the RMI model represents a *distributed object application*
    - it allows an object inside a JVM (a client) to invoke a method on an object running on a remote JVM (a server) and have the results returned to the client
  - the server application creates an object and makes it accessible remotely (i.e., registers it)
  - the client application receives a reference to the object on the server and invokes methods on it
    - the reference is obtained through looking up in the registry
  - **important:** *a method invocation on a remote object has the same syntax as a method invocation on a local object*

# Remote Method Invocation (RMI)

## Architecture I.

The interface, through which the client and server interact, is (similarly to RPC) provided by **stubs** and **skeletons**:

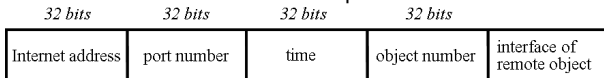


# Remote Method Invocation (RMI)

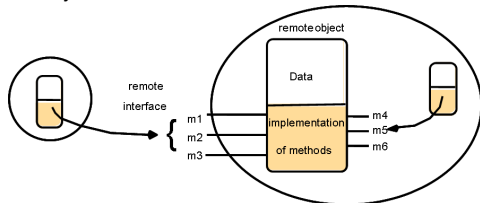
## Architecture II.

Two fundamental concepts as the heart of distributed object model:

- *remote object reference* – an identifier that can be used throughout a distributed system to refer to a particular unique remote object
  - its construction must ensure its uniqueness



- *remote interface* – specifies, which methods of the particular object can be invoked remotely



# Remote Method Invocation (RMI)

## Architecture III.

- the remote objects can be accessed concurrently
  - the encapsulation allows objects to provide methods for protecting themselves against incorrect accesses
    - e.g., synchronization primitives (condition variables, semaphores, etc.)
- RMI transaction semantics similar to the RPC ones
  - *at-least-once*, *at-most-once*, *exactly-once*, and *maybe semantics*
- data encoding services:
  - stubs use *Object Serialization* to marshal the arguments
    - object arguments' values are rendered into a stream of bytes that can be transmitted over a network
    - ⇒ the arguments must be primitive types or objects that implement `Serializable` interface
- parameters passing:
  - local objects passed *by value*
  - remote objects passed *by reference*



# Common Object Request Broker Architecture (CORBA)

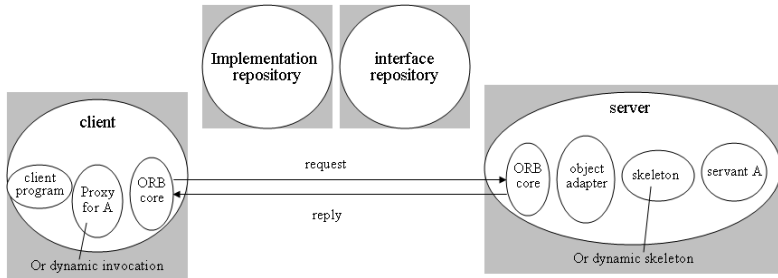
- an industry standard developed by the OMG (*Object Management Group* – a consortium of more than 700 companies) to aid in distributed objects programming
  - OMG was established in 1988
  - initial CORBA specification came out in 1992
    - but significant revisions have taken place from that time
- provides a **platform-independent** and **language-independent** architecture (framework) for writing distributed, object-oriented applications
  - i.e., application programs can communicate without restrictions to:
    - programming languages, hardware platforms, software platforms, networks they communicate over
  - but CORBA is just a *specification* for creating and using distributed objects; it is *not a piece of software or a programming language*
    - several implementations of the CORBA standard exist (e.g., IBM's SOM and DSOM architectures)



## CORBA Components

CORBA is composed of five major components:

- *Object Request Broker (ORB)*
- *Interface Definition Language (IDL)*
- *Dynamic Invocation Interface (DII)*
- *Interface Repositories (IR)*
- *Object Adapters (OA)*



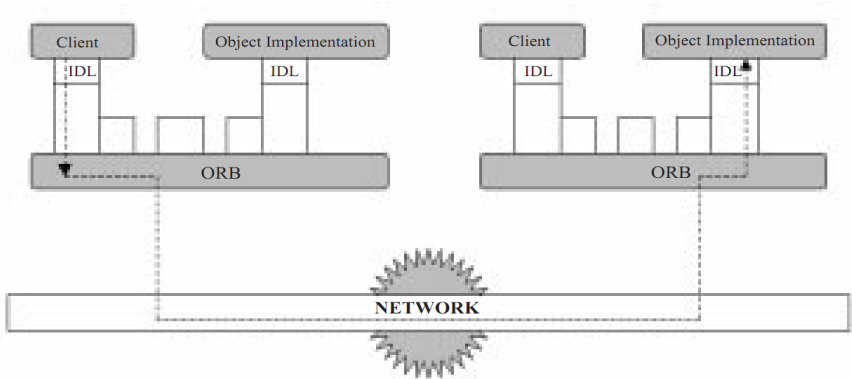
# CORBA Components

## Object Request Broker (ORB)

- the heart of CORBA
  - introduced as a part of OMG's *Object Management Architecture (OMA)*, which the CORBA is based on
- a distributed service that implements all the requests to the remote object(s)
  - it **locates the remote object** on the network, **communicates the request** to the object, waits for the results and (when available) **communicates those results** back to the client
- implements *location transparency*
  - exactly the same request mechanism is used regardless of where the object is located
    - might be in the same process with the client or across the planet
- implements *programming language independence*
  - the client issuing a request can be written in a different programming language from the implementation of the CORBA object
- both the client and the object implementation are isolated from the ORB by an IDL interface
- *Internet Inter-ORB Protocol (IIOP)* – the standard communication protocol between ORBs

# CORBA Components

## Object Request Broker (ORB) II.





# CORBA Components

## Interface Definition Language (IDL)

- as with RMI, CORBA objects have to be specified with interfaces
  - interface  $\approx$  a contract between the client (code using a object) and the server (code implementing the object)
    - indicates a set of operations the object supports and how they should be invoked (but NOT how they are implemented)
- defines modules, interfaces, types, attributes, exceptions, and method signatures
  - uses same lexical rules as C++
    - with additional keywords to support distribution (e.g. interface, any, attribute, in, out, inout, readonly, raises)
- defines language bindings for many different programming languages (e.g., C/C++, Java, etc.)
  - via language mappings, the IDL translates to different constructs in the different implementation languages
  - it allows an object implementor to choose the appropriate programming language for the object, and
  - it allows the developer of the client to choose the appropriate and possibly different programming language for the client



## CORBA Components

### Interface Definition Language (IDL) example:

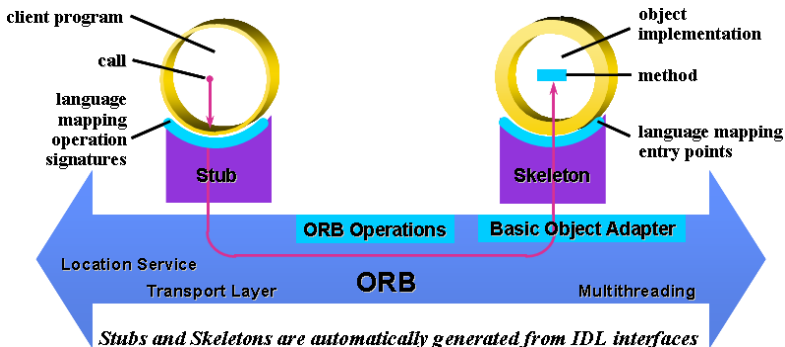
```
module StockObjects {
    struct Quote {
        string symbol;
        long at_time;
        double price;
        long volume;
    };
    exception Unknown{};
    interface Stock {
        // Returns the current stock quote.
        Quote get_quote() raises(Unknown);
        // Sets the current stock quote.
        void set_quote(in Quote stock_quote);
        // // Provides the stock description, e.g. company name.
        readonly attribute string description;
    };
    interface StockFactory {
        Stock create_stock(in string symbol, in string description);
    };
};
```



## CORBA Components

### Interface Definition Language (IDL) III. – Stubs and Skeletons

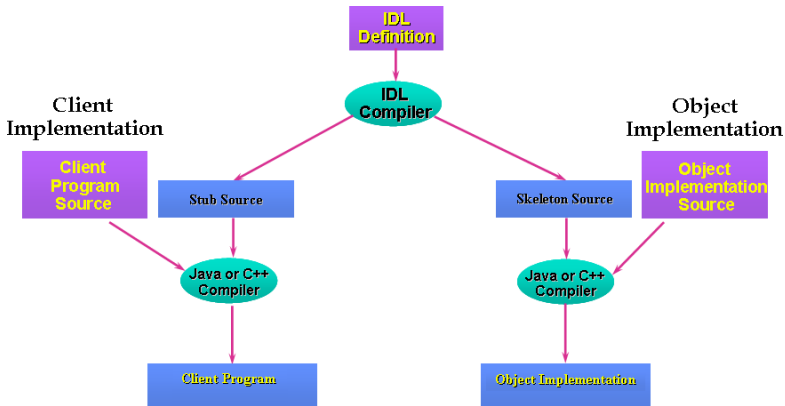
IDL compiler automatically compiles the IDL into **client stubs** and **object skeletons**:





# CORBA Components

## Interface Definition Language (IDL) IV. – Development Process Using IDL





# CORBA Components

## DII & DSI

### Dynamic Invocation Interface (DII)

- CORBA supports both the *dynamic* and the *static invocation interfaces*
  - *static invocation interfaces* are determined at compile time
  - *dynamic interfaces* allow client applications to use server objects without knowing the type of those objects at compile time
- DII – an API which allows dynamic construction of CORBA object invocations

### Dynamic Skeleton Interface (DSI)

- DSI is the server side's analogue to the client side's DII
  - allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing



# CORBA Components

## Interface Repository (IR)

- a runtime component used to dynamically obtain information on IDL types (e.g. object interfaces)
  - using the IR, a client should be able to locate an object that is unknown at compile time, find information about its interface, and build a request to be forwarded through the ORB
  - this kind of information is necessary when a client wants to use the DII to construct requests dynamically

# CORBA Components

## Object Adapters (OAs)

- the interface between the ORB and the server process
  - OAs listen for client connections/requests and map the inbound requests to the desired target object instance
- provide an API that object implementations use for:
  - generation and interpretation of object references
  - method invocation
  - security of interactions
  - object and implementation activation and deactivation
  - mapping object references to the corresponding object implementations
  - registration of implementations
- two basic kinds of OAs:
  - *basic object adapter (BOA)* – leaves many features unsupported, requiring proprietary extensions
  - *portable object adapter (POA)* – intended to support multiple ORB implementations (of different vendors), allow persistent objects, etc.

# CORBA Object & Object Reference

**CORBA Objects** are fully encapsulated

- accessed through well-defined interfaces only
- interfaces & implementations are totally separate
  - for one interface, multiple implementations possible
  - one implementation may be supporting multiple interfaces



**CORBA Object Reference** is the distributed computing equivalent of a pointer

- CORBA defines the *Interoperable Object Reference (IOR)*
- an IOR contains a fixed object key, containing:
  - the object's fully qualified interface name (repository ID)
  - user-defined data for the instance identifier
- can also contain transient information:
  - the host and port of its server, metadata about the server's ORB (for potential optimizations), etc.
- ⇒ the IOR uniquely identifies one object instance



# CORBA Services

## CORBA Services (COS)

- the OMG has defined a *set of Common Object Services* to support the integration and interoperation of distributed objects
  - = frequently used components needed for building robust applications
  - typically supplied by vendors
  - OMG defines interfaces to services to ensure interoperability



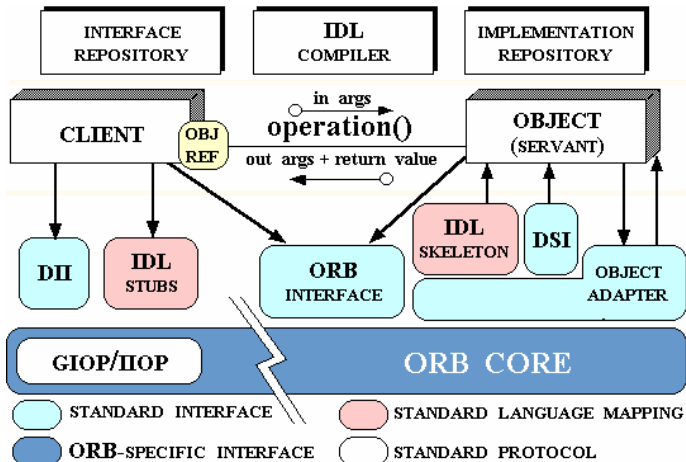
# CORBA Services

## Popular Services Example

<i>Service</i>	<i>Description</i>
<b>Object life cycle</b>	Defines how CORBA objects are created, removed, moved, and copied
<b>Naming</b>	Defines how CORBA objects can have friendly symbolic names
<b>Events</b>	Decouples the communication between distributed objects
<b>Relationships</b>	Provides arbitrary typed n-ary relationships between CORBA objects
<b>Externalization</b>	Coordinates the transformation of CORBA objects to and from external media
<b>Transactions</b>	Coordinates atomic access to CORBA objects
<b>Concurrency Control</b>	Provides a locking service for CORBA objects in order to ensure serializable access
<b>Property</b>	Supports the association of name-value pairs with CORBA objects
<b>Trader</b>	Supports the finding of CORBA objects based on properties describing the service offered by the object
<b>Query</b>	Supports queries on objects



## CORBA Architecture Summary





# Lecture overview

## Distributed Systems

Key characteristics

Challenges and Issues

Distributed System Architectures

Inter-process Communication

## Middleware

Remote Procedure Calls (RPC)

Remote Method Invocation (RMI)

Common Object Request Broker Architecture (CORBA)

## Service Oriented Architecture (SAO)

## Web Services



# Lecture overview

## Distributed Systems

- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

## Middleware

- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

## Service Oriented Architecture (SAO)

## Web Services