



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Introduction to Middleware

TNGK Ranganath

What is Middleware?

- Today, most business processes have become digitalized, and most organizations have very diverse digital needs.
- To fulfill these needs, businesses have to use many different hardware and software products.
- Most of these hardware and software were designed separately. They were not necessarily built to work together.
- At the same time, organizations need these different hardware and software to work together in order to make digital processes more efficient.
- Problem is, how do you make them work harmoniously when they weren't built to work together?
- This is where middleware comes in.
- Middleware refers to any behind the scenes software that allows these two levels to communicate and interact with each other.
- For instance, middleware will sit between Windows 10 and an office productivity suite.
- Aside from the operating system and applications, middleware also helps separate process, applications and software components to exchange information either within the same device, or between multiple devices.

What is Middleware?

- You can compare middleware to a translator helping people who speak different languages understand each other.
- In this case, middleware facilitates interoperability between applications running on different frameworks. Middleware does this by providing a standard-based means of data exchange.
- This way, the two applications can connect without having to communicate directly.
- Some people refer to middleware as plumbing since it connects and passes data between two fundamentally different applications.
- Middleware has also been referred to as “software glue”, since it helps “glue” together different software so they can work together.
- The term middleware is a bit vague since it does not refer to a specific type of software. Instead, it refers to any software that sits between and links two separate applications.

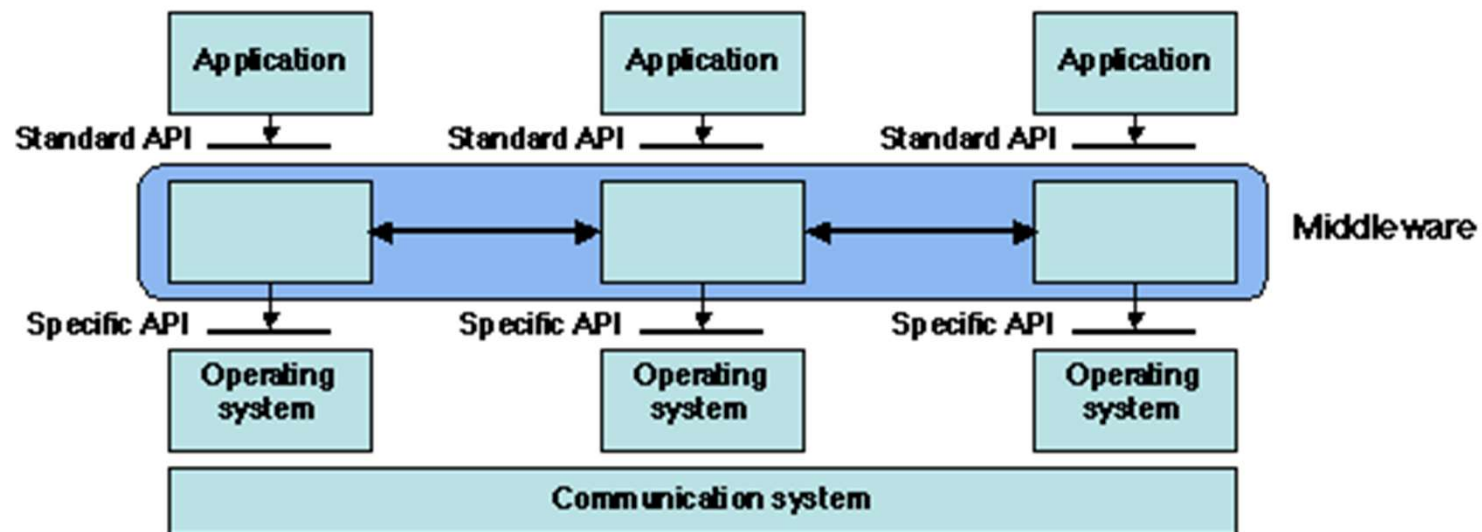
What is Middleware?

- Middleware includes software like [content management systems](#), [application servers](#), [web servers](#), and other similar tools that support the development and delivery of applications.
- Middleware started becoming popular in the 80s as a solution for enabling newer applications to work on older systems.
- To enable communication between different applications, middleware utilizes different communication frameworks such as [Representational State Transfer \(REST\)](#), web services, [JavaScript Object Notation \(JSON\)](#), [Simple Object Access Protocol \(SOAP\)](#), and so on.
- Modern integration infrastructure such as [enterprise service bus \(ESB\)](#) and [API management software](#) also depend on middleware concepts.

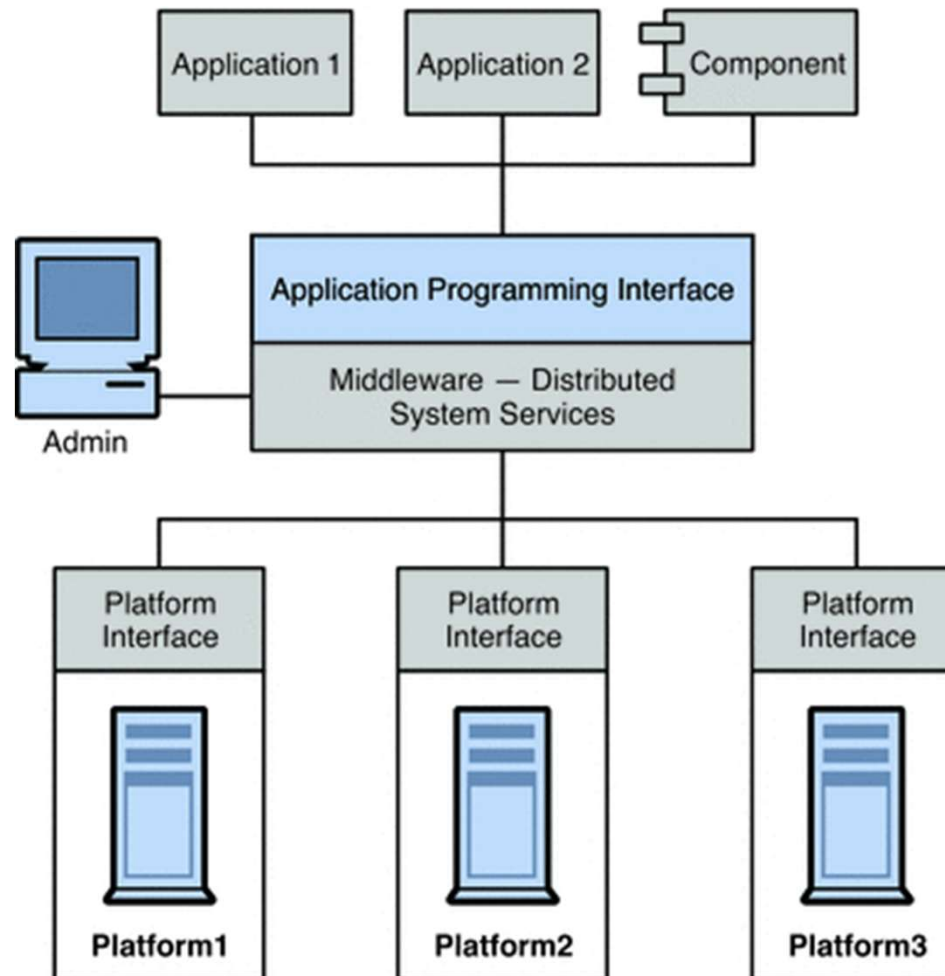
What is Middleware ?

What is it ?

- “Middleware is the software that connects software components or enterprise applications in a distributed system”.
- Examples: Enterprise Application Integration software, telecommunications software, transaction monitors, and messaging-and-queueing software.



What is Middleware?



How does Middleware Work?

- Today, businesses rely a lot on network applications which involve enterprise and database systems.
- These network applications need to perform many different functions, such as updating orders, messaging customers, facilitating payments, allowing customers to track shipments, and so on.
- All these functions require real-time transfer of data between different devices.
- In addition, different devices with varying processing power, bandwidth capacities, and screen and visual display capabilities need to access the network.
- Middleware steps in to provide a unified means for all these systems to communicate and interact with each other.
- To do this, a lot of middleware is cross-language, which means that it is capable of understanding and processing several different operating languages, such as Ruby on Rails, Java, C++, PHP, and so on.

How does Middleware Work?

- Apart from allowing communication between fundamentally different systems, middleware also performs several other functions.
- Hiding the distributed nature of an application. On the surface, applications appear to be one unified package.
- Below the surface, however, they are comprised of several interconnected elements running in distributed locations.
- Middleware helps these different elements work in harmony to provide a unified experience for the user, despite the distributed nature of the application.
- Hiding the heterogeneity of the enterprise. The enterprise is usually made up of different hardware, different operating systems and different communication protocols.
- Middleware allows these different systems to work together while masking their differences.
- Providing application developers with uniform and standard high-level enterprises that they can use to build applications that can be run on different hardware and operating systems and work with each other.
- Avoiding duplication and enabling compatibility between applications by providing a common framework for performing various general purpose functions.

Middleware helps make application development easier

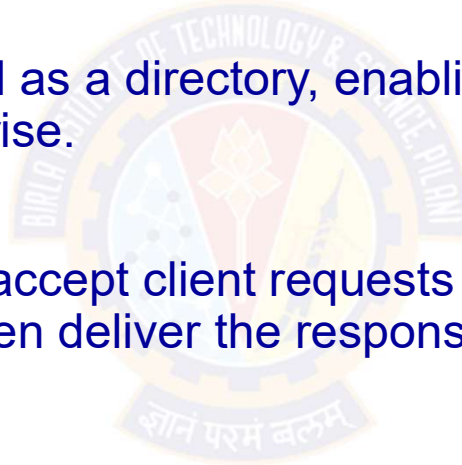
- In order to support application development, middleware uses the following components:
- **Database software:** Most multi-tier systems will require a database.
 - Middleware acts as the link between the client and the server.
 - It accepts client requests, passes them on to the database server and then passes back the response to the client.
- **Application server:** This is the part of the application that holds the business logic of the application.
- **Portal:** This is an interaction tool that is used to provide a selected audience with access to business applications, relevant information, instant messaging, discussion forums, and other company resources.
- **Service Oriented Architecture (SOA):** This is a framework that is used to easily design, develop and deploy applications.
 - Many types of middleware use SOA with prebuilt services that can be utilized by multiple systems.
- **Web server:** The role of the web server is to process and deliver client requests.
 - Web servers provide one of the best and most flexible options for the integration of different systems.

USES OF MIDDLEWARE

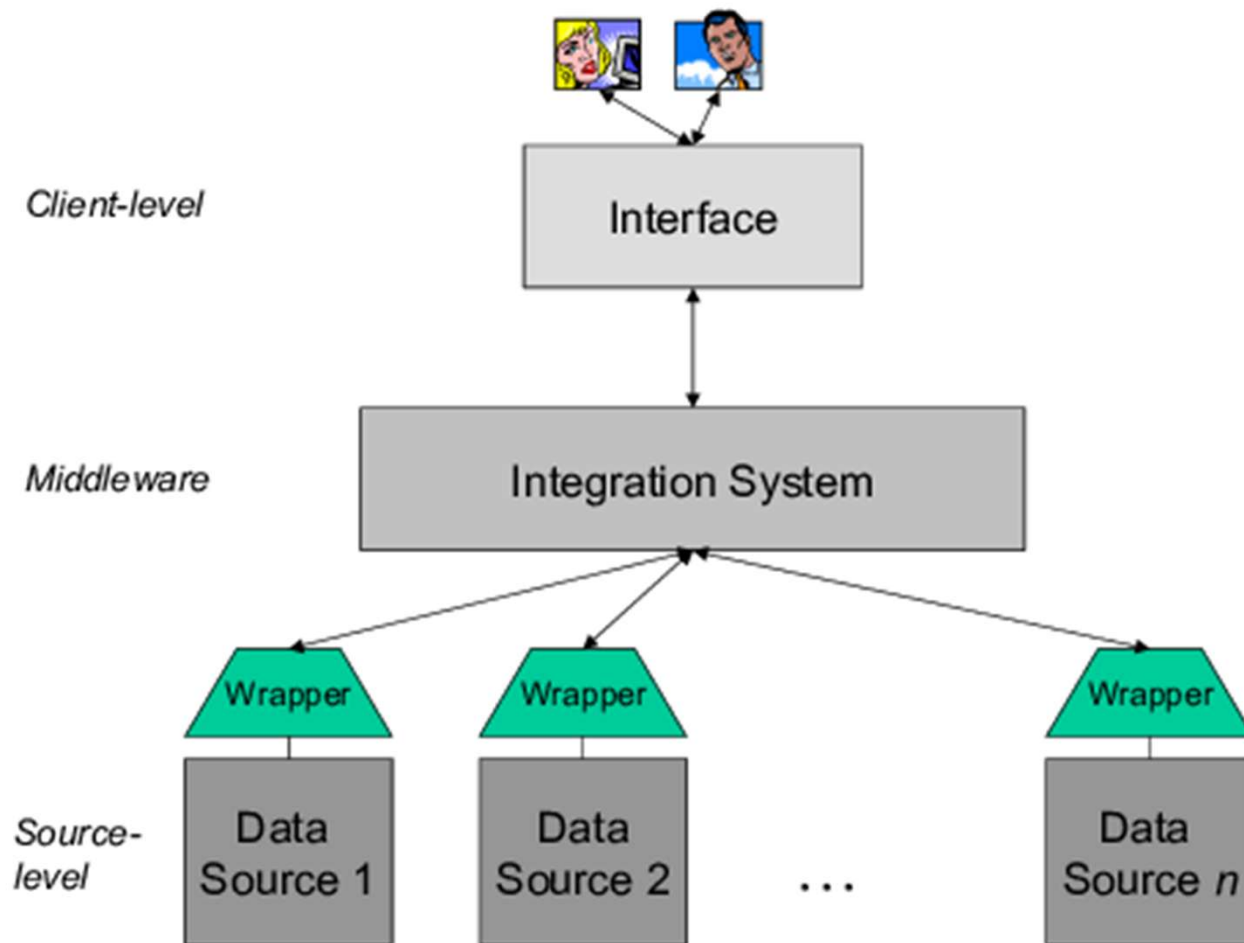
- We have already seen that middleware acts as a link between different software.
- But what exactly does it do apart from acting as a conduit between the different software?
- Some of the uses of middleware include:
- **Transaction management:** Middleware can be used to manage and control individual transactions and ensure that any problems do not corrupt the system or database.
- **Application server:** Middleware can be used to host an API, allowing other applications to access and use the main application's business logic and processes.
- **Security:** Middleware can be used to authenticate client programs and confirm that the program and the user behind the program are actually who they claim to be.

USES OF MIDDLEWARE

- **Message queues:** Middleware can be used to pass messages between different systems or software.
- The messages can then trigger a transaction or other action.
- **Directory:** Middleware can be used as a directory, enabling client programs to locate other services within a distributed enterprise.
- **Web server:** Middleware can also accept client requests from web browsers and channel them to the main server/database and then deliver the responses to the browsers.



What is Middleware?



Applications of Middleware

From enterprise to platform middleware

- There are specifically two applications of middleware: Enterprise and Platform.
- **Enterprise middleware:**
 - Enterprise middleware connects software components or enterprise applications.
 - It is the layer of software between the operating system and the applications on either side of a computer network, usually supporting complex, distributed business software applications.
- **Platform middleware:**
 - Platform middleware connects different application architectures.
 - Some technology firms operate using multiple application structures.
 - In the event in which firms merge or when there are third-party application acquisitions, a company may find that they are using multiple structures.
 - Middleware supports these structures, and provides methods in three arenas of interconnection: development environments, production and test.
 - In all of these arenas, middleware transfers the data from application to application, as well as between databases and files.

Types of Middleware

- **Message Oriented Middleware (MOM)**
 - This is software infrastructure that allows messages to be sent and received over distributed applications
 - This prevents the message from getting misplaced while awaiting to get to the server or client. An example of message oriented middleware is email systems.
- **Remote Procedure Call (RPC) Middleware**
 - This is a client-server interaction that makes it possible for the functionality of an application to be distributed across multiple platforms.
 - This type of middleware is most used to execute synchronous data transfers, where the both the client and the server need to be online at the time of the communication.

Types of Middleware

- **Database Middleware**
 - This type of middleware allows direct access and interaction with a database. Database middleware is the most common and most widely used type of middleware.
 - It is mostly used by developers as a mechanism to request information from a database hosted either locally or remotely.
 - A good example of database middleware is the SQL database software
- **Application Programming Interface (API)**
 - An API is a set of protocols, tools and definitions for building applications, which allow a secondary application or service to communicate with a primary application or service, without having to know how the primary application or service is being implemented.
- **Object Middleware**
 - Also known as an object request broker, the role of object middleware is to control the communication between objects in distributed computing.
 - Object middleware allows one computer to make program calls to another through a computer network.
 - It also allows requests and objects to be sent through an object-oriented system.

Types of Middleware

- **Transaction Processing (TP) Middleware**

- This is a type of middleware whose role is to reinforce the function of electronic transactions.
- Transactional middleware does this by controlling transaction apps, pushing database updates related to the transaction and enforcing the business rules and logic of the transaction.

- **Robotic Middleware**

- This type of middleware is very handy when it comes building extensive software systems for controlling robot systems.
- Robotic middleware helps to manage and control the heterogeneity and complexity of the hardware and software systems that form part of a robot.

- **Integration Middleware**

- This type of middleware provides an integration framework through which operations, executions and runtime services from several apps can be monitored and controlled.
- Integration middleware can also be useful in combining data from several different sources into one unified platform where users can access and manipulate data.

Types of Middleware

- **Application Framework**

- This is a framework that provides the basic structure on which applications for a particular environment can be built.
- The application framework acts as a backbone that supports the applications
- Using an application framework makes the application development process a lot simpler.

- **Device Middleware**

- This is a type of middleware that provides a set of tools which are used to build applications meant to be run in a specific hardware environment.

- **Game Engines**

- This type of middleware provides game designers with access to tools that make the game creation process easier.
- Game engines utilize tools such as game scripting, physics simulations, and graphics rendering.

Types of Middleware

- **Portals**
 - Though they might not actually be a type of middleware, enterprise portal servers are also sometimes referred to middleware because they enable a smooth front-end integration.
 - The main role of portals is to allow interaction between a client device and back end systems.
- **Content-Centric Middleware**
 - This is a type of middleware that makes it possible for developers to extract some piece of content without having to know how the system obtains the content.
 - This type of middleware is commonly used in most content-oriented web-based applications.

What is a Transaction ?

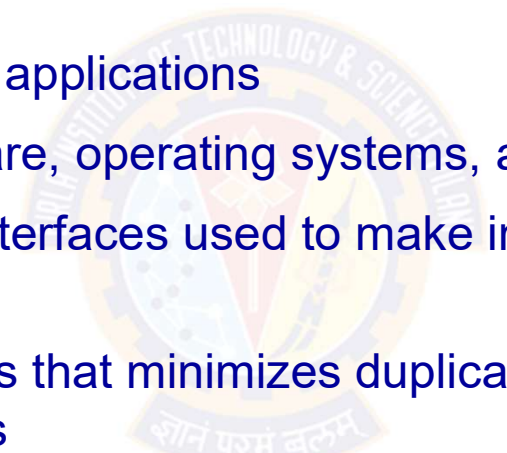
Transaction

- “A transaction symbolizes a unit of work performed (comprising of multiple operations if needed) within a system (distributed or monolith) and treated in a coherent and reliable way independent of other operations.”
- Properties of a Transaction:
 - Atomic
 - Consistent
 - Isolated
 - Durable
- All or Nothing mode of operation
- Ex: Funds transfer (debit remittance account and credit beneficiary account)



Middleware in Distributed computing

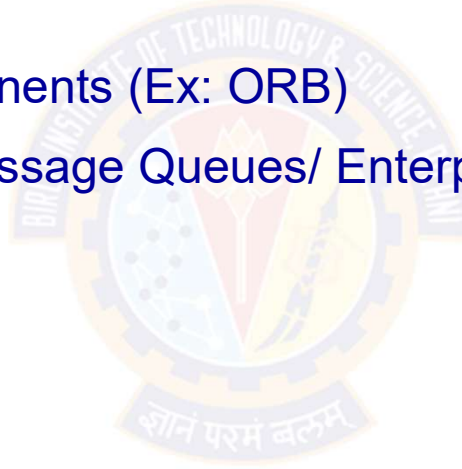
What does it do ?

- Lies between the operating system and the applications on each side of a distributed computer network
 - Hides the intricacies of distributed applications
 - Hides the heterogeneity of hardware, operating systems, and protocols
 - Provides uniform and high-level interfaces used to make interoperable, reusable and portable applications
 - Provides a set of common services that minimizes duplication of efforts and enhances collaboration between applications
- 

Middleware – common forms

Commonly used Architectures of Middleware

- Sockets
- Remote Procedure Calls
- Distributed Object Oriented Components (Ex: ORB)
- Message Oriented Middleware (Message Queues/ Enterprise Message Bus etc.)
- Service Oriented Architectures
- Web services (Arbitrary / RESTful)
- SQL-oriented data access
- Embedded middleware
- Cloud Computing



Importance & Benefits of Middleware

- Implementing middleware so that you can integrate the data across various applications and systems.
- Integration makes the flow of data across the various applications a lot easier and allows your company to focus on other important aspects of your business, since you no longer have to spend time on manual processes.
- Some of the benefits to be gained by implementing middleware include :
- **Improved Agility**
- Today, businesses need to deliver services to customers across various platforms, including on the cloud, on mobile and through traditional application platforms.
- Delivering services across all these platforms can be a challenge.
- Middleware can help provide this agility. It provides a framework that allows changes to be easily made to business processes.
- This way, the business can easily respond and adapt to customer requirements and expectations and deliver new services much faster.

Importance & Benefits of Middleware

- **Increased Efficiency**
 - Middleware technology is very useful when it comes to automating business processes.
 - With middleware, processes such as ordering and product configuration can be automated, leading to time and cost improvements when compared to performing these processes manually.
- **Rapid Innovation**
 - Today's business landscape has become very competitive.
 - For instance, one hotel chain used middleware technology from Oracle to provide users with real-time information about room availability and rates directly on Google Maps.

Importance & Benefits of Middleware

- **Portability and Reusability**
 - An application built on top of certain middleware can be used on multiple platforms, making the application a lot more portable.
- **Cost Effectiveness**
 - Due to the use of common components, developing applications on top of middleware technology means less effort is required to build the application from scratch.
- **Information Management**
 - Middleware technology can make information management much easier by providing a framework on which an information management system can be designed, built and deployed.



Examples of Middleware

- **Red Hat JBoss Enterprise Application Platform**

- This is a powerful and versatile middleware technology developed by Red Hat Software.
- This middleware has a solid architectural foundation, with very low memory requirements and very quick startup times.
- Red Hat JBoss Enterprise Application Platform also offers integration with DevOps tools such as Arquillian, Jenkins and Maven.
- The middleware comes with a “migration center” that makes it very easy to move existing applications to this platform.
- It also provides a cloud-compatible solution for those who want to make their enterprises more agile.
- Red Hat JBoss Enterprise Application Platform also provides very excellent customer support.
- Red Hat JBoss Enterprise Application Platform costs \$8,000 for a 1 year subscription running on 16 CPU cores. You can also get round the clock assistance and support for an extra \$4,000.

Examples of Middleware

- **IBM WebSphere**
- The Websphere Application Server is a high performance middleware platform with a plethora of features. Installing and deploying IBM WebSphere is quite simple.
- The platform can be used for local, cloud-based and hybrid solutions. The platform is built of strong but flexible architecture that can easily scale as required.
- One of the greatest advantages of IBM WebSphere is that it can be integrated with other cloud-based IBM products, such as dashDB SQL database service and [Watson artificial intelligence](#).
- A one year subscription of IBM WebSphere inclusive of support will cost about \$14,000 per limited use socket..

Examples of Middleware

- **Oracle WebLogic**
- Oracle's middleware solution is another very popular and very powerful option for enterprise middleware.
- Deploying applications on Oracle WebLogic is easy and hassle-free, which means lower operational costs.
- The platform comes with an integration to Oracle's high performance cross-platform communication architecture, which is christened Enterprise Grid Messaging.
- WebLogic also offers hassle-free integration with other Oracle databases and applications.
- The price of an Oracle WebLogic license will vary based on the number of processors or number of users.

Choosing a Middleware platform

- Some things to keep in mind when evaluating an enterprise middleware solution. These include:
 - Updating the business logic should be easy and fast.
 - The middleware solution should offer seamless back end interaction.
 - It should be able to offer seamless integration between different platforms running on different architecture.
 - Security should be a key consideration. Since middleware will work through the web, there should be a security mechanism to ensure that any secured data passing through web servers cannot get compromised.
 - In the case of message oriented middleware, there should be a message encryption feature to prevent communication from being intercepted. In the case of application servers, there should be a mechanism in place to ensure proper authentication.
 - The middleware solution should have audit functionality such that all activities performed on the platform can be logged and reported.

In this segment

Sockets Overview

- Introduction
- Data Structures
- Library Calls
- General Operation
- Socket Example

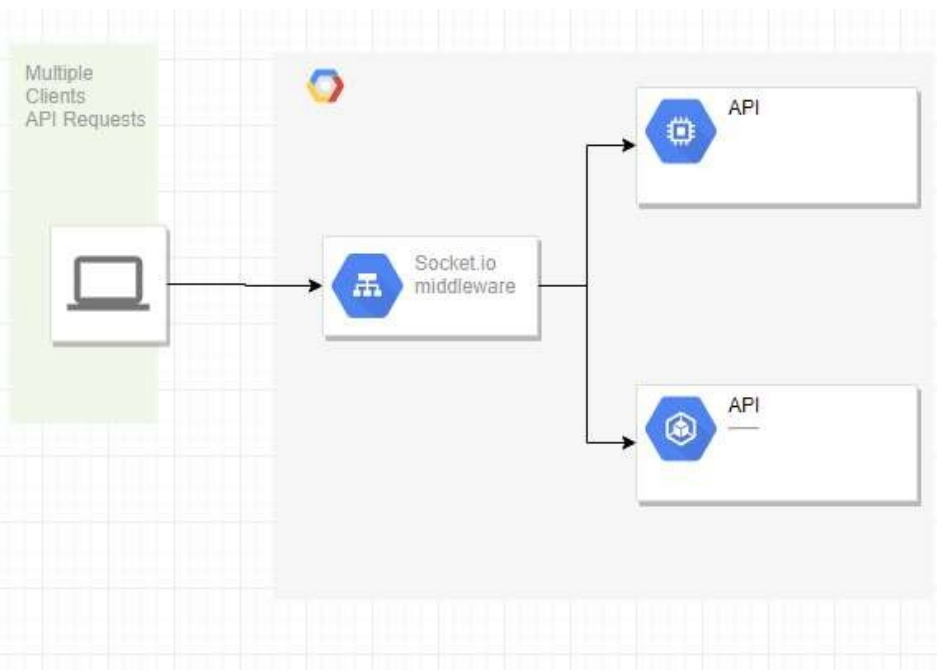


Socket - Introduction

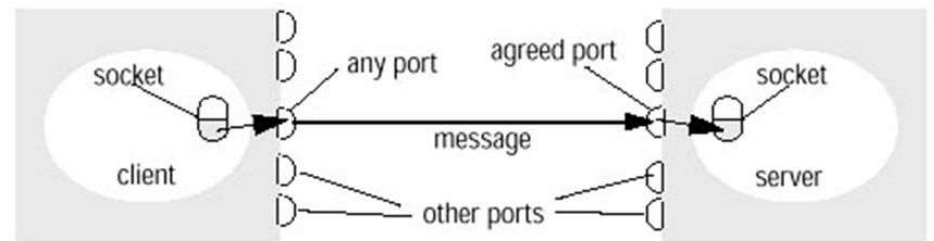
What is it ?

- Socket is an internal endpoint for sending/receiving data within a node on network
- Berkeley/POSIX sockets defined API for Inter Process Communication (IPC) within same host (BSD 4.2 – circa 1983)
- Early form of Middleware (limited to same host systems)
- Windows variant (WinSock) based on BSD Sockets.
- Treated similar to files in BSD/POSIX
- Maintained in File Descriptor table
- Supported protocols
 - TCP/IP – IPv4, IPv6
 - UDP

Socket - Introduction



Sockets and Ports



Source: G. Coulouris et al., *Distributed Systems: Concepts and Design*

Socket – Data Structures

Socket Address

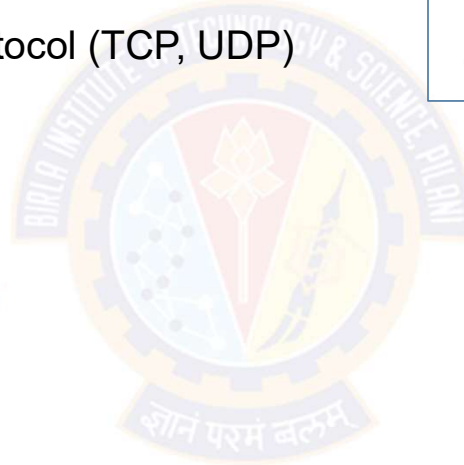
- Defined in *sys/socket.h*
- Address length is 8 bytes by default
- Family – Denotes the family of protocol (TCP, UDP)
- Address contains – host and port
- Socket Address family TCP/IP
 - IPv4 - `sockaddr_in`

```
sa_family_t    sin_family;  
in_port_t      sin_port;  
struct in_addr sin_addr;
```

- IPv6 - `sockaddr_in6`

```
sa_family_t    sin6_family;  
in_port_t      sin6_port;  
uint32_t       sin6_flowinfo;  
struct in6_addr sin6_addr;  
uint32_t       sin6_scope_id;
```

```
struct sockaddr  
{  
    unsigned char sa_len; // length of address  
    sa_family_t sa_family; // the address family  
    char sa_data[14]; // the address  
};
```



Socket – Library Calls

Socket APIs

socket —creates a descriptor for use in network communications

connect —connect to a remote peer (client)

write —send outgoing data across a connection

read —acquire incoming data from a connection

close —terminate communication and deallocate a descriptor

bind —bind a local IP address and protocol port to a socket

listen —set the socket listening on the given address and port for connections from the client and set the number of incoming connections from a client (backlog) that will be allowed in the listen queue at any one time

accept —accept the next incoming connection (server)

recv —receive the next incoming datagram

recvmsg —receive the next incoming datagram (variation of recv)

recvfrom —receive the next incoming datagram and record its source endpoint address

send —send an outgoing datagram

sendmsg —send an outgoing datagram (variation of send)

sendto —send an outgoing datagram, usually to a prerecorded endpoint address

shutdown —terminate a TCP connection in one or both directions

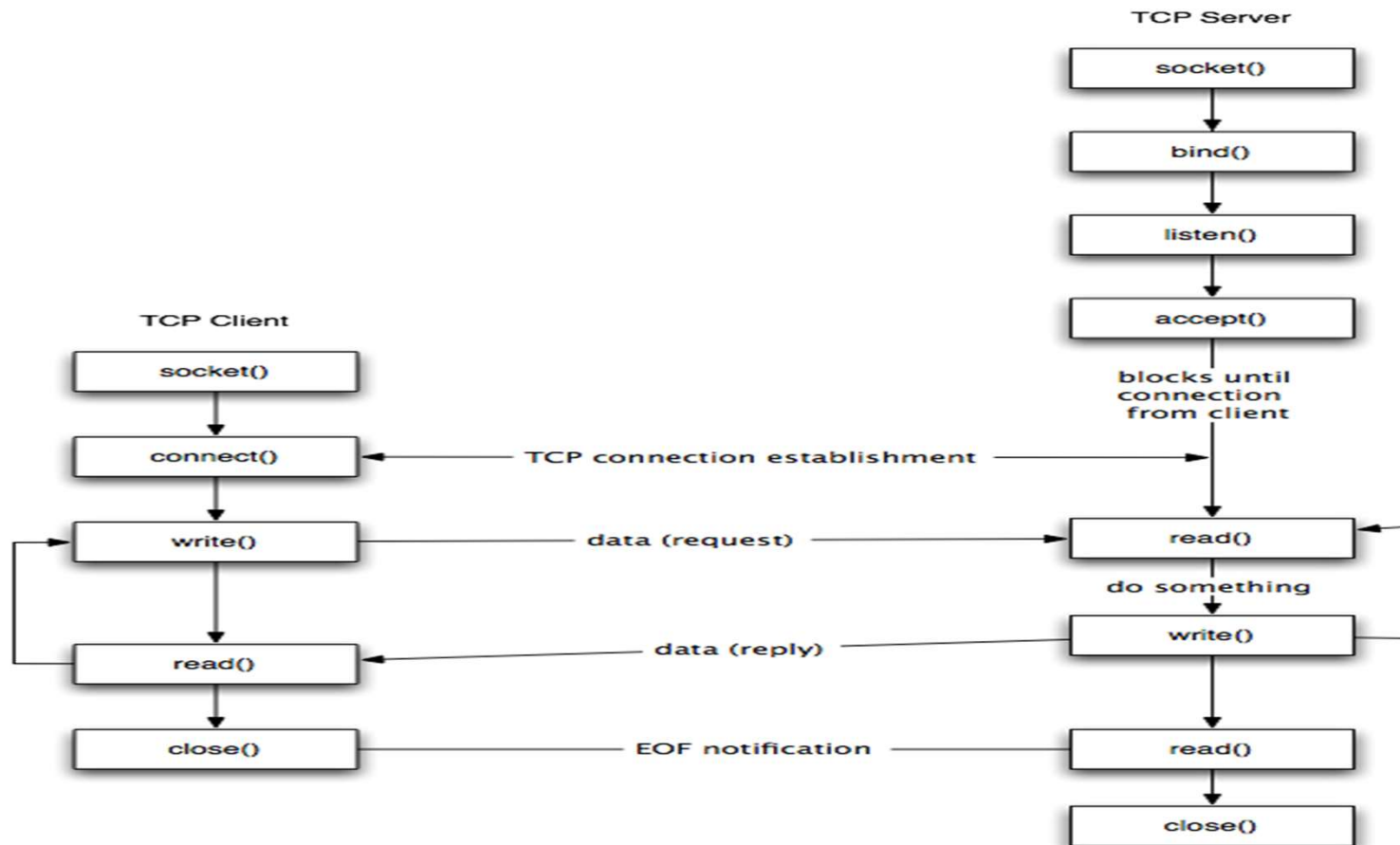
getpeername —after a connection arrives, obtain the remote machine's endpoint address from a socket

getsockopt —obtain the current options for a socket

setsockopt —change the options for a socket

Socket – General Operation

Lifecycle



Socket – Example

Server code

```
int main()
{
    int server_socket_fd, client_conn_fd;
    int client_addr_size;
    struct sockaddr_in server_addr, client_addr;
    int port_number;
    char message_from_client[256];
    char message_from_server_to_client[256];
    int client_message_length, server_message_length;
    // Set the port number
    port_number = 1132;
    // Create the socket for the server to listen on
    server_socket_fd = socket( AF_INET,
    SOCK_STREAM,
    0
    );
    if (server_socket_fd < 0)
        PrintError("ERROR opening socket");
    // clear out the server address to make sure no problems with binding
    // if the address is clear then it won't think the address
    has
    // already been used by another socket
    bzero( (char *) &server_addr, sizeof(server_addr));
```

```
    // set the sockaddr_in struct appropriately
    // note that this assumes IPv4
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(port_number);
    // Bind the socket descriptor to the address
    if (bind( server_socket_fd,
    (struct sockaddr *) &server_addr,
    sizeof(server_addr))
    < 0)
        PrintError("binding to socket failed");
    // Start the server listening on the socket
    // limit the number of connections in the listen queue to 3
    // (this is the backlog)
    listen(server_socket_fd, 3);
    while (1)
    {
        // clear out the client address to make sure no problems with accepting on this address
        // if the address is clear then it won't think the address has
        // already been used by another socket
        bzero( (char *) &client_addr, sizeof(client_addr));
```

Socket – Example

Server code

// accept a connection from a client

// on the open socket

```
client_addr_size = sizeof(client_addr);
client_conn_fd = accept( server_socket_fd,
(struct sockaddr *) &client_addr,
(socklen_t *) &client_addr_size
);
```

```
if (client_conn_fd < 0)
```

```
PrintError("the accept failed");
```

// Clear out the character array to store the client message

// to make sure there's no garbage in it

```
bzero(message_from_client, 256);
```

// read the message from the client

```
client_message_length = read( client_conn_fd,
message_from_client,
255
```

```
);
```

```
if (client_message_length < 0)
```

```
PrintError(" unable to read from socket" );
```

```
cout << " message is" << message_from_client <<
endl;
```

// Now write a message back to the client

// Doing a little mixed mode C++ strings and char buffers

// just to show you how

```
string mystring;
```

```
mystring = " Server received from client, then echoed back to
client:" ;
```

```
mystring += message_from_client;
```

```
bzero(message_from_server_to_client, 256);
```

```
mystring.copy(message_from_server_to_client, mystring.length() );
```

```
server_message_length = write( client_conn_fd,
```

```
(char *) message_from_server_to_client,
```

```
strlen(message_from_server_to_client)
```

```
);
```

```
if (server_message_length < 0)
```

```
PrintError(" unable to write to socket" );
```

```
sleep(5); // let the client close first, avoids socket address reuse
issues
```

```
close(client_conn_fd);
```

```
} // end while (1)
```

```
close(server_socket_fd); // this isn't reached because we used
```

```
while (1)
```

```
// but with a different while loop test condition
```

```
// this would be important
```

```
return 0;
```

```
}
```

Socket – Example

Client code

```
int main()
{
    int socket_fd;
    struct sockaddr_in serv_addr;
    int port_number;
    char * IP_address;
    char client_message[256];
    int message_result;
    // Set the IP address (IPv4)
    IP_address = new char [sizeof(" 127.0.0.1" )];
    strcpy(IP_address, " 127.0.0.1" ); // could
    instead have copied " localhost"
    // Set the port number
    port_number = 1132;
    // Create the socket
    socket_fd = socket( AF_INET,
    SOCK_STREAM,
    0
    );
    if (socket_fd < 0)
    PrintError(" ERROR opening socket" );
    // clear out the server address to make sure
no problems with binding
    bzero((char *) &serv_addr, sizeof(serv_addr));
```

```
    // set the sockaddr in struct appropriately
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(port_number);
    // Use the inet_pton function to convert the IP
address to binary
    if (inet_pton( AF_INET,
    IP_address,
    &serv_addr.sin_addr
    )
    < 0
    )
    PrintError(" Unable to convert IP address to binary to
    put in serv_addr" );
    // Connect to the server
    if (connect( socket_fd,
    (struct sockaddr *)
    &serv_addr,
    sizeof(serv_addr)
    )
    <0
    )
    PrintError(" unable to connect to server" );
    cout << " Enter message to send to server: " ;
    bzero(client_message,256);
```

Socket – Example

Client code

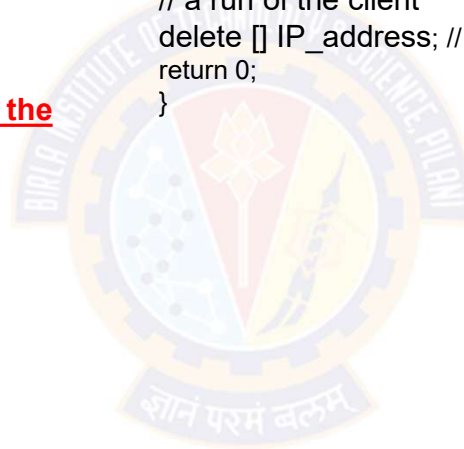
```
string mystring;
getline(cin, mystring); // read the line from standard
input
cout << endl;
strcpy(client_message, mystring.c_str());

// Write the message to the socket to send to the
server
message_result = write( socket_fd,
client_message,
strlen(client_message)
);
if (message_result < 0)
PrintError("unable to write to socket");

// Read the return message from the server
bzero(client_message, 256);
message_result = read( socket_fd,
client_message,
255
);
if (message_result < 0)
PrintError("unable to read from socket");
cout << client_message << endl;
```

```
// close(socket_fd); // commented out because only
close it if you
```

```
// don't want to do more calling the server from
// a run of the client
delete [] IP_address; // return the memory to the heap
return 0;
}
```



In this segment

Early Middleware Technologies

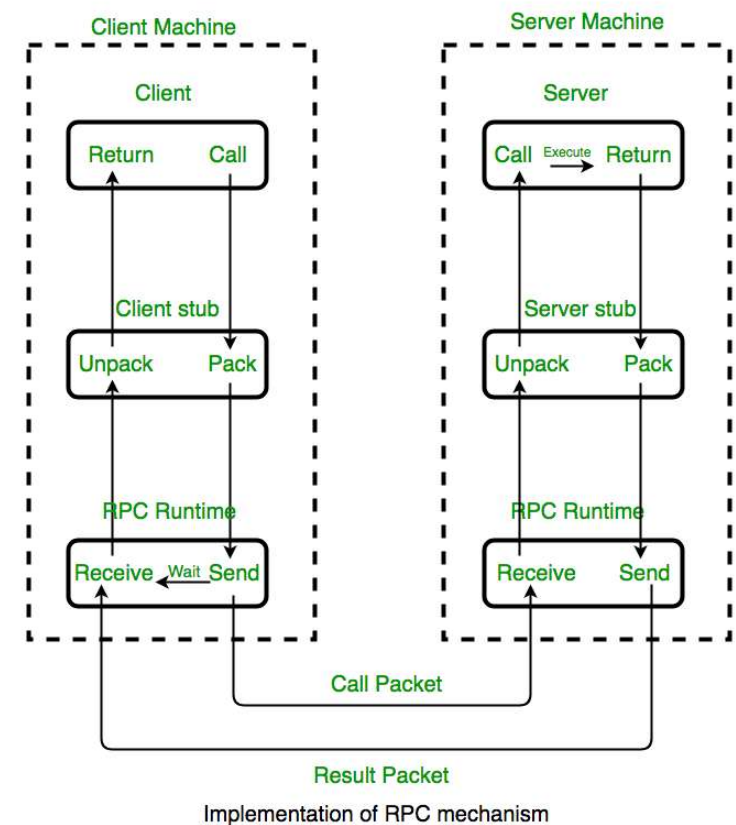
- Remote Procedure Calls
- Distributed Object Oriented Components
- Message Oriented Middleware (MOM)



Remote Procedure Calls (RPC)

Overview

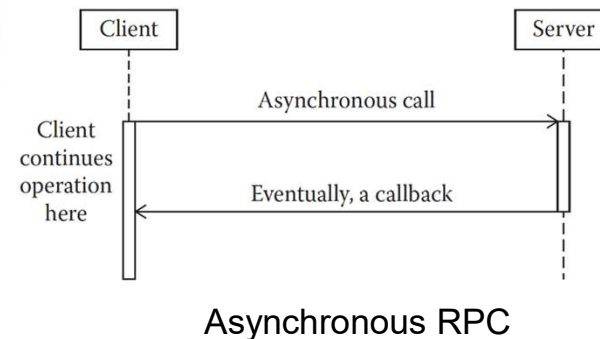
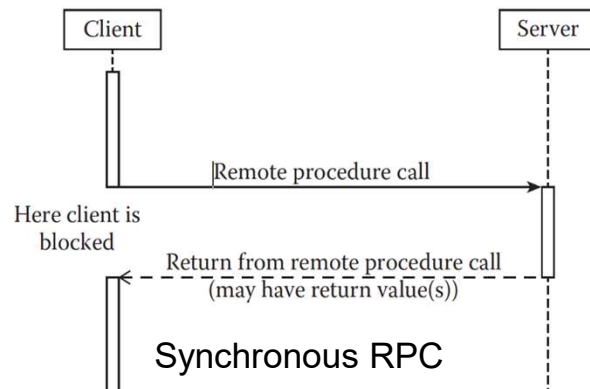
- RPC is a method call across the network, which is treated same as a local procedure call (location transparency)
- Open Network Computing (ONC) RPC implementation (early attempt of middleware)
 - RPC methods defined in “.x” files
 - *rpcgen* translates .x files to stubs on client side and skeleton files on server side
- Stubs and skeletons handle packing/unpacking of data from application to network layer (bytes) – called **Marshaling**
- Applications on either side (client and server) are designed as if they run on the same machine, leaving underlying network / transport specifics to **RPC runtime**



Remote Procedure Calls (RPC)

Synchronous Vs Asynchronous

- Synchronous RPC **blocks** the client until server finishes the operation and returns response (or error)
- Asynchronous RPC allows client to continue its business while server processes RPC
 - Client may send return address (along with request) that server can use to **callback** to send response
 - Server may use a mediator (also called **broker**) to return response to the client when finished



Remote Procedure Calls (RPC)

- On a site A , consider a process p that executes a local call to a procedure P (Figure 1.6 a).
- Design a mechanism that would allow p to perform the same call, with the execution of P taking place on a remote site B (Figure 1.6 b), while preserving the semantics (i.e. the overall effect) of the call.
- We call A and B the client site and the server site, respectively, because RPC follows the client-server, or synchronous request-response, communication paradigm

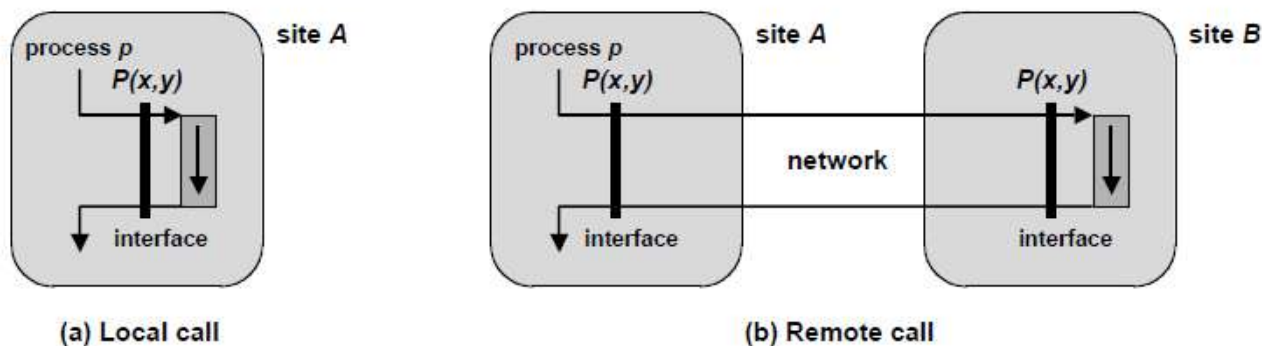


Figure 1.6. Remote procedure call: overview

Remote Procedure Calls - Implementation Principles

- The standard implementation of RPC relies on two pieces of software, the client stub and the server stub (Figure 1.7).
- The client stub acts as a local representative of the server on the client site; the server stub as a local representative of the client on the server site.
- Thus both the calling process (on the client side) and the procedure body (on the server side) keep the same interface as in the centralized case.
- The client and server stubs rely on a communication subsystem to exchange messages.

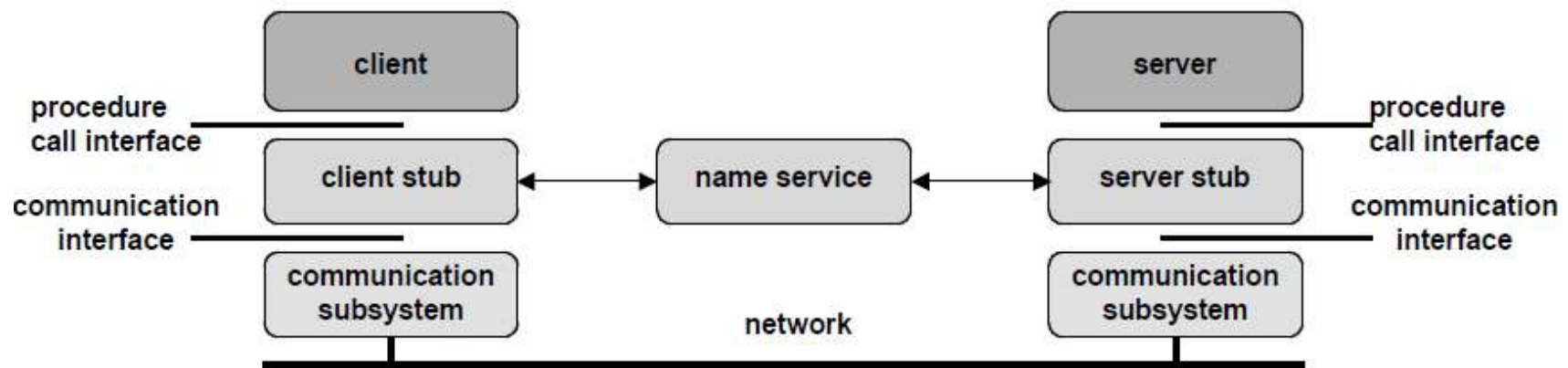
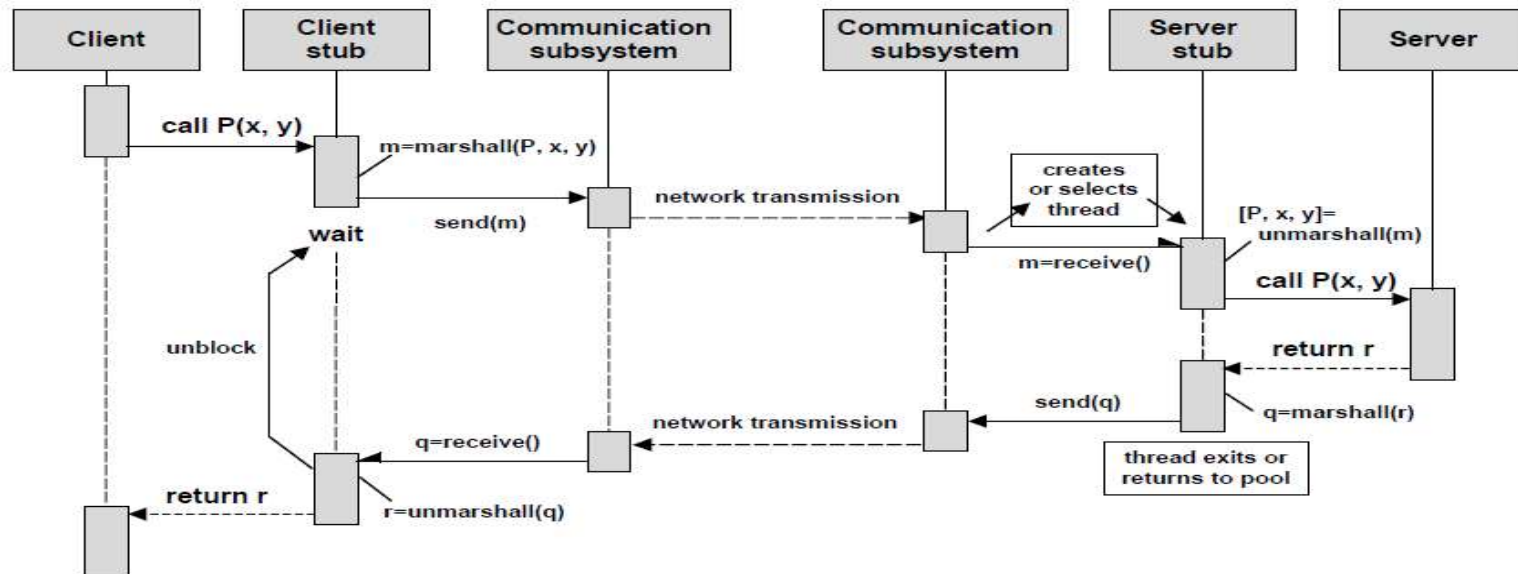


Figure 1.7. Remote procedure call: main components

RPC – Parameter Marshalling and Unmarshalling

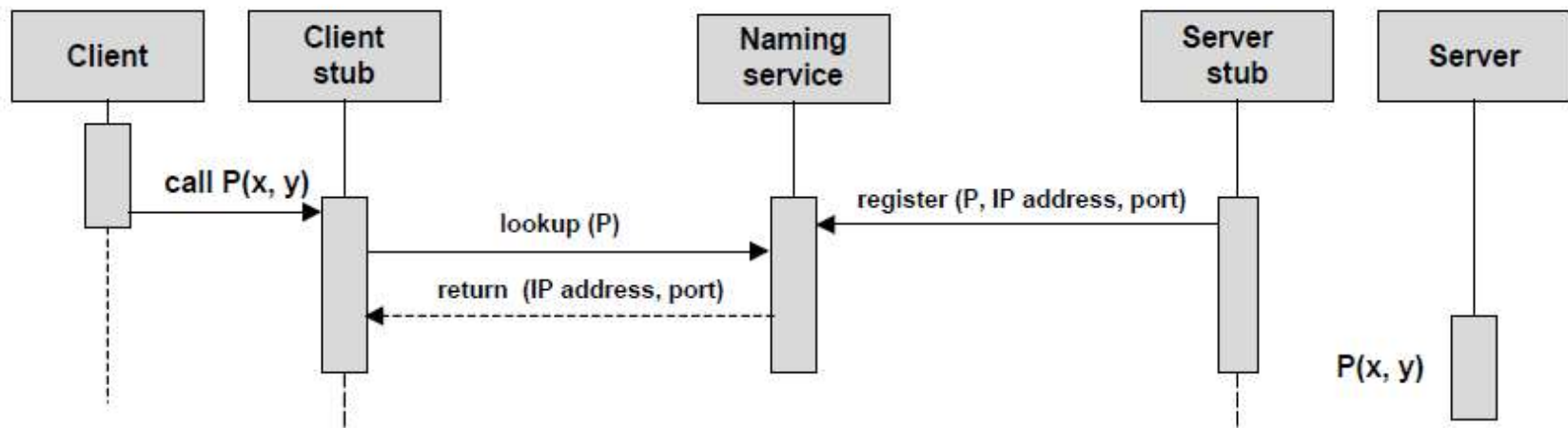
- Parameters and results need to be transmitted over the network. Therefore they need to be put in a serialized form, suitable for transmission.
- In order to ensure portability, this form should be standard and independent of the underlying communication protocols as well as of the local data representation conventions (e.g. byte ordering) on the client and server machines.
- Converting data from a local representation to the standard serialized form is called **marshalling**. The reverse conversion is called **unmarshalling**.



RPC – Client Server Binding

- The client needs to know the server address and port number to which the call should be directed. Therefore, the client must locate the server site prior to the call.
- This is the function of a naming service, which is essentially a registry that associates
- procedure names with server addresses and port numbers.
- A server registers a procedure name together with its IP address and the port number at which the daemon process is waiting for calls on that procedure.
- A client consults the naming service to get the IP address and port number for a given procedure.

Figure 1.10 shows the overall pattern of interaction involving the naming service.



Distributed Object Oriented Components

Overview

- Distributed component is a concurrent object with a well-defined interface, which is a logical unit of distribution and deployment
- Employs object oriented methodologies (where objects encapsulate data inside) to design a system, with components spread over network, communicating using middleware components
- Distributed Object middleware handle both synchronous and asynchronous communication
- Examples: CORBA, DCOM, Java RMI, EJB

Distributed Object Oriented Components

Component or Object Oriented Middleware ?

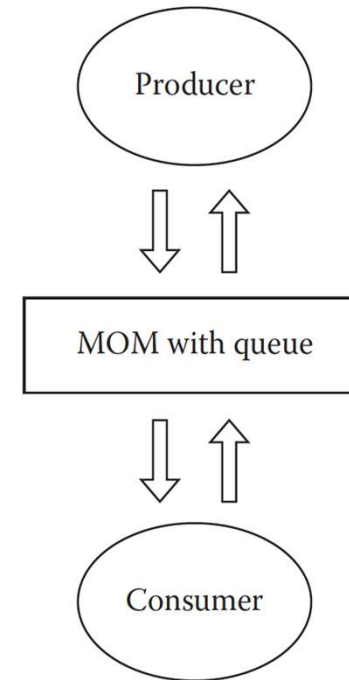
- Component based systems comprises of loosely coupled components (need not be object oriented), most of which could be existing components
 - CORBA
 - DCOM
 - EJB
- Object oriented systems focus on modeling real world situations, albeit using components at times
 - CORBA
 - DCOM



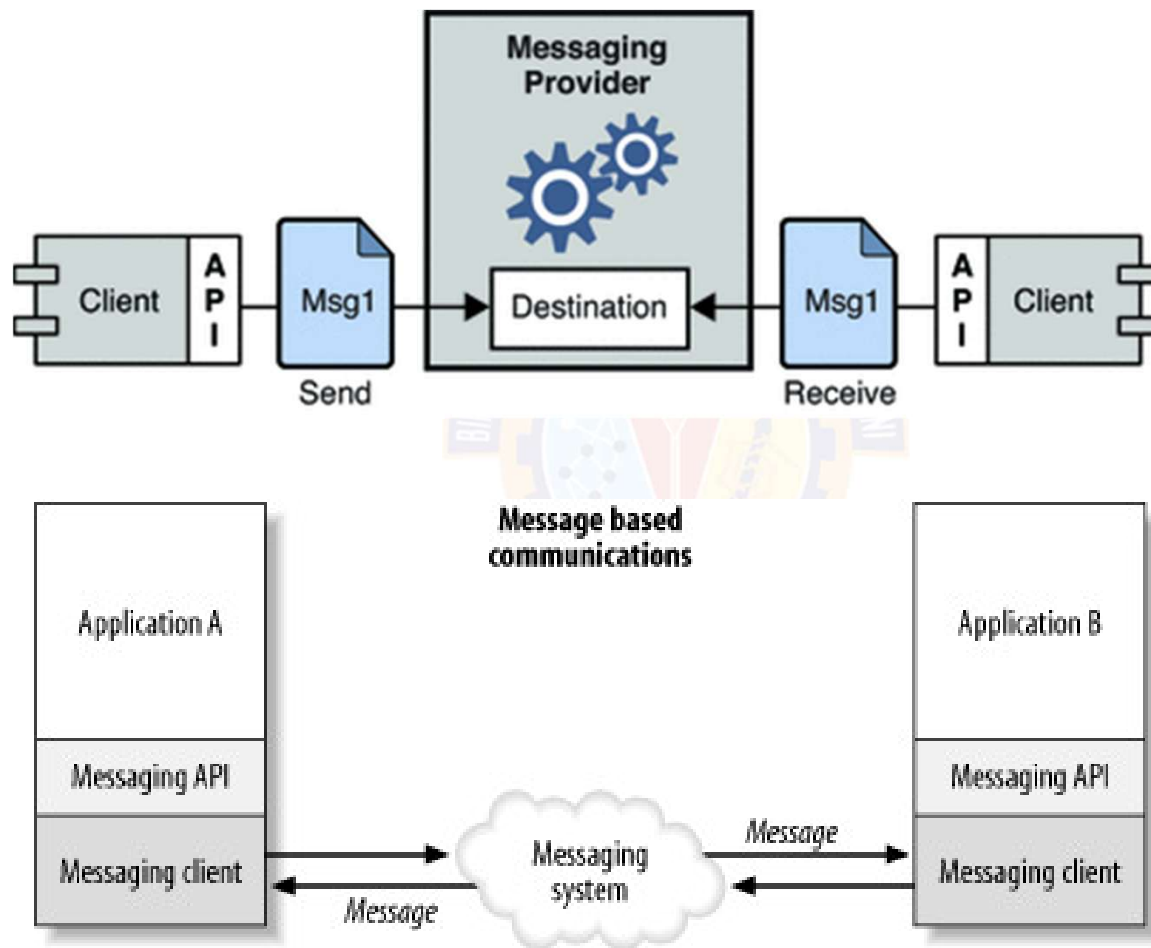
Message Oriented Middleware (MOM)

Overview

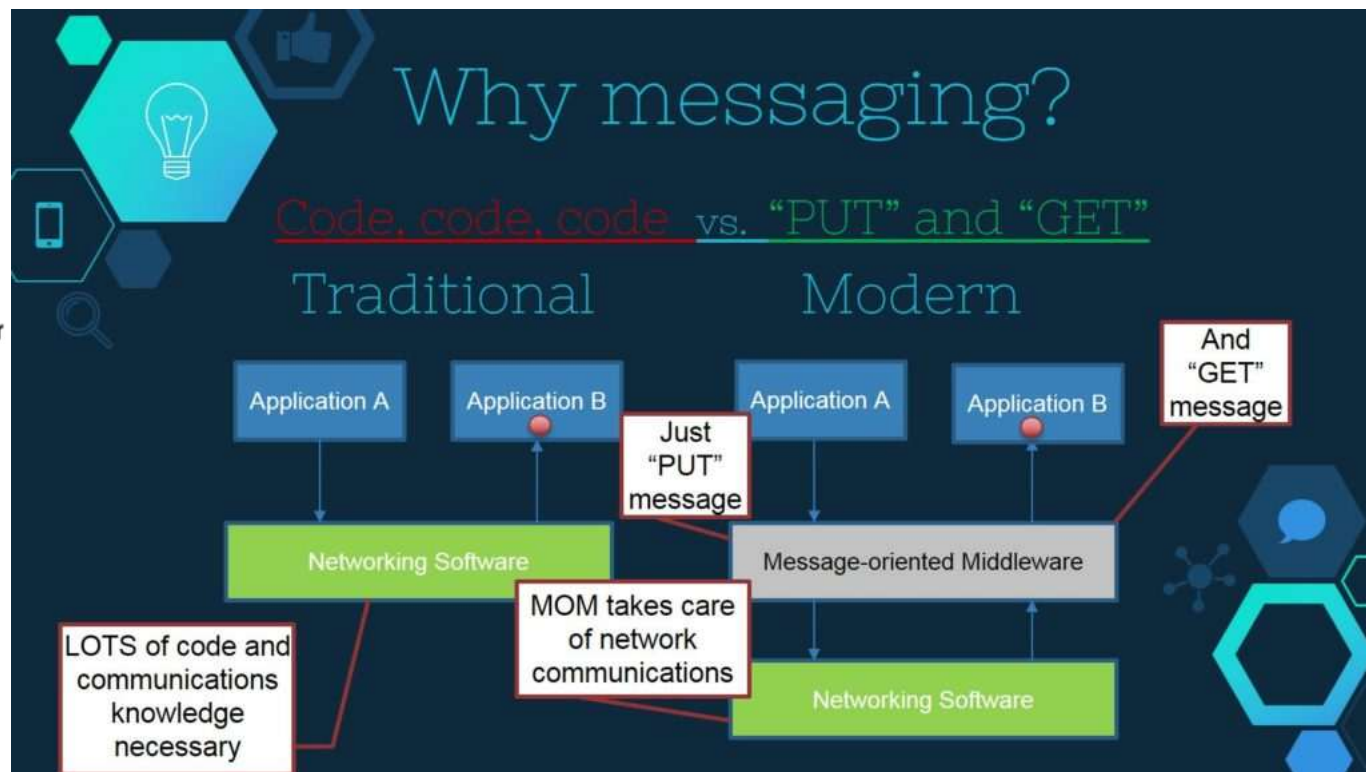
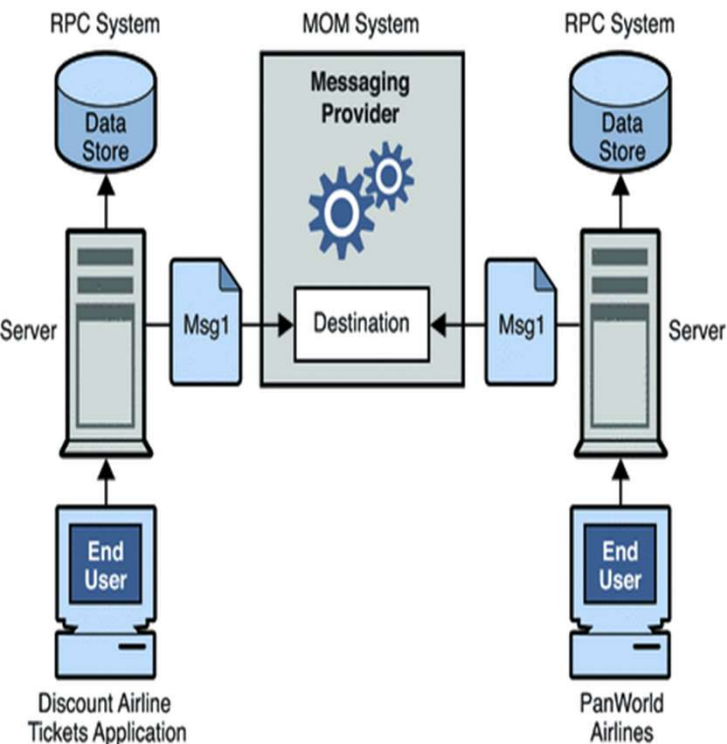
- Acts as a middleman (broker) between client and server for communication
- Brings in loose coupling in the design of client and server, by offloading communication to a third party
- Enables client to work asynchronously with server
- Generally (not necessarily always) uses queueing mechanism to deliver messages, in a publish/subscribe model
- MOM will handle auxiliary operations as well, like Quality of Service, Priority processing, Reliability, Recovery of lost messages etc.



Message Oriented Middleware (MOM)



Message Oriented Middleware (MOM)



In this segment

CORBA Overview

- CORBA Introduction
- CORBA Basics



CORBA

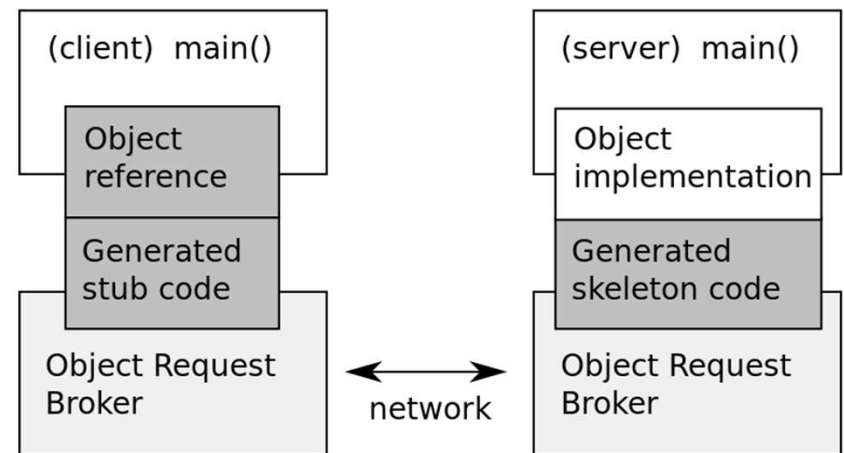
Evolution and History

- Introduced in 1990s by Object Management Group (OMG)
- Popular initially and available even today in some installations
- Introduced the notion of “everything for everybody”, i.e. any language/platform talking to any other language/platform via CORBA
- Gradually declined in 2000s with the explosion of Internet (WWW) and advent of firewalls
- Hacking of ports was a major challenge for CORBA as time progressed

CORBA

Overview

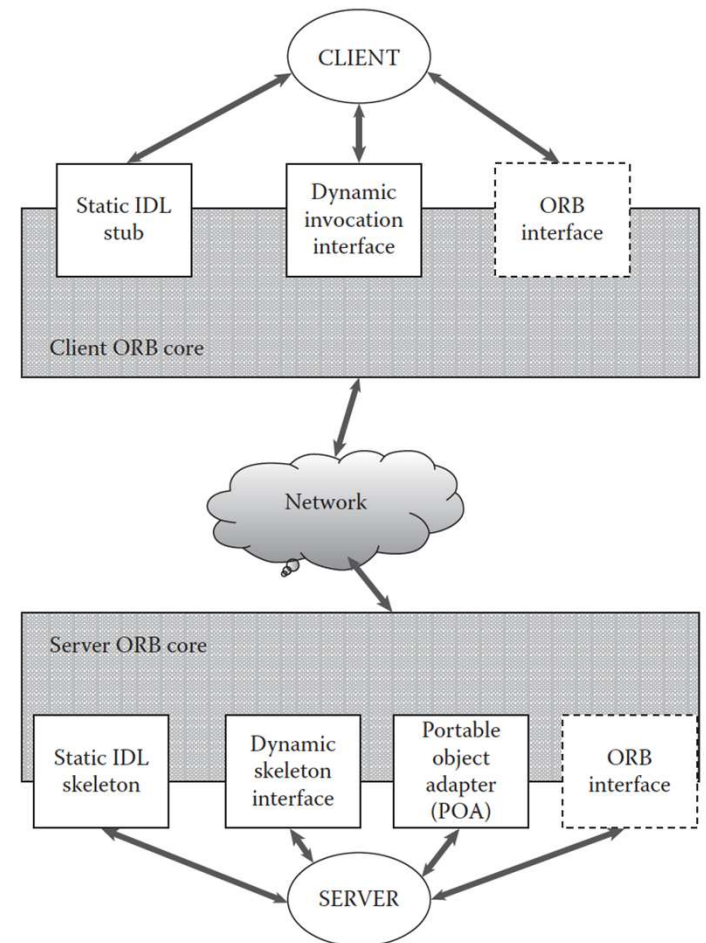
- Primarily client-server oriented
- Can be seen as RPC on remotely located objects (due to distributed OO nature)
- Uses General Inter-Orb Protocol (on TCP/IP) to communicate and transfer data between client and server
- Client side proxy object
 - Represents remote object in local client space
 - Handles data serialization/encoding and communication with network using ORB



CORBA

Architecture

- CORBA supports static and dynamic interfaces using Interface Description language IDL
 - Static interface defined at compile time using IDL compiler
 - Dynamic interface that defines invocation at runtime
- IDL stub is client side implementation that calls CORBA library API for client operation
- IDL skeleton is on server side, that uses CORBA library API for server side operation
- POA (Portable Object Adapter) performs server-side functions, similar to EJB containers, Web servers etc.



In this segment

CORBA IDL – Session 2

- IDL Overview
 - Parameters
 - Modules
 - Exceptions
 - Structs and Arrays
 - Sequences
 - Attributes
- IDL to Java bindings

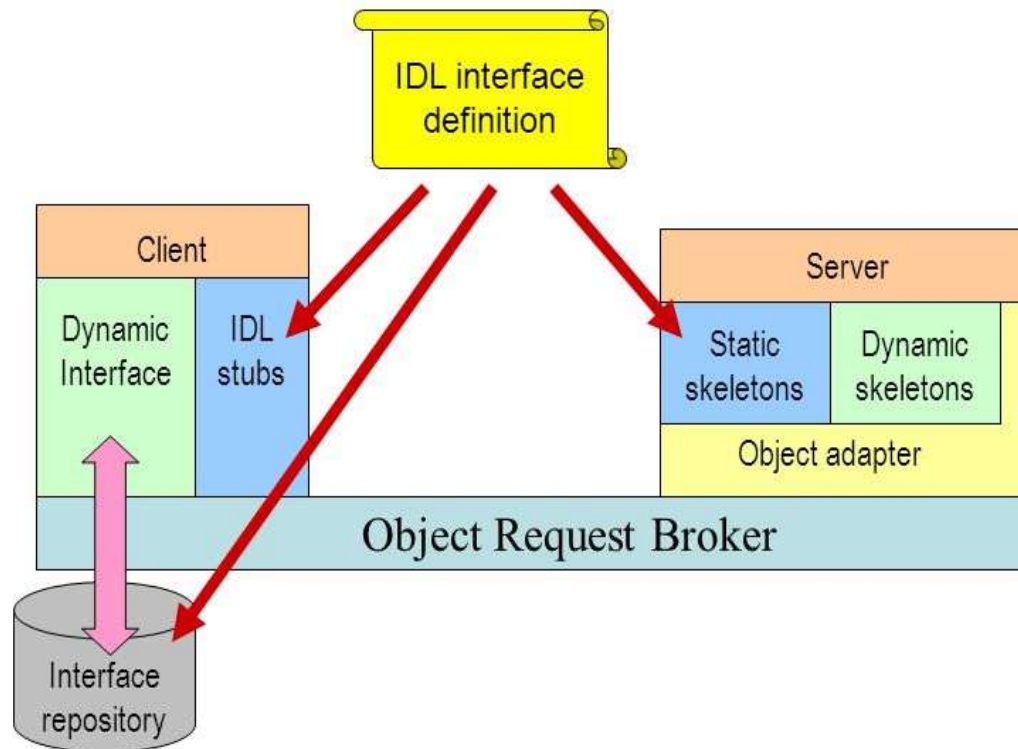


Interface Definition Language - IDL

- An **interface description language** or **interface definition language (IDL)**, is a [specification language](#) used to describe a [software component's application programming interface](#) (API).
- IDLs describe an interface in a [language-independent](#) way, enabling communication between software components that do not share one language, for example, between those written in [C++](#) and those written in [Java](#).
- IDLs are commonly used in [remote procedure call](#) software. In these cases the machines at either end of the *link* may be using different [operating systems](#) and computer languages.
- IDLs offer a bridge between the two different systems.
- Software systems based on IDLs include [Sun's ONC RPC](#), [The Open Group's Distributed Computing Environment](#), [IBM's System Object Model](#), the [Object Management Group's CORBA](#) (which implements OMG IDL, an IDL based on DCE/RPC) and [Data Distribution Service](#), [Mozilla's XPCOM](#), [Microsoft's Microsoft RPC](#) (which evolved into [COM](#) and [DCOM](#)), [Facebook's Thrift](#) and [WSDL](#) for [Web services](#).

CORBA IDL

IDL products



Interface Description Language (IDL)

Overview

- Defines methods in a fashion similar to interfaces in Java
- Implementation is internal to the server (and transparent to client)
- Data types supported:
 - String
 - Integer
 - short – 16 bit (signed / unsigned)
 - long – 32 bit (signed / unsigned)
 - long long – 64 bit (signed / unsigned)
 - Octet – 8 bit (similar to byte)
- No keyword restrictions for naming (as IDL is language neutral), however there may be compiling issues depending on languages involved
- Enums – Order not guaranteed

```
interface echo {  
    string echostring (in string the_echostring);  
};
```

```
interface echo {  
    string echoshort (inout short the_echoshort);  
};
```

```
interface if {  
    while switch (in for the_data);  
}
```

```
enum Money {euro, dollar, pound, yen};
```

Interface Description Language (IDL)

Parameters, Modules and Exceptions

- Parameters
 - 'in' is input – from client to server
 - 'out' is output – from server to client
 - 'inout' – same parameter sent from client is used for output (similar to pass by reference)
- Modules – Groups multiple interfaces together, with hierarchical naming (like Java packages)
- Exceptions
 - System Defined
 - COMM_FAILURE
 - MARSHAL
 - BAD_PARAM
 - OBJECT_NOT_EXIST
 - TRASNIENT
 - UNKNOWN
 - User defined

```
interface echo {  
    string echostring (inout string the_echostring);  
};
```

```
module echomodule {  
    interface echo {  
        string echostring (in string the_echostring);  
    };  
};
```

```
interface echo {  
    exception Bad_Message{ };  
    string echostring (in string the_echostring) raises (Bad_Message);  
};
```

Interface Description Language (IDL)

Structs, Arrays and Sequences as Params

- Structs as params
 - Can be used as in, out and inout
- Arrays as params
 - No guarantee on array indexing
- Sequence as params
 - Sends only the data present in the sequence at the time of method call

```
struct theData {  
    long firstvalue;  
    string secondvalue;  
};
```

```
typedef theData StructType;
```

```
interface sending_stuff {  
    string sendvalue (in StructType mystruct);  
};
```

```
interface sending_stuff {  
    StructType sendvalue (in string mystring);  
};
```

```
typedef short ArrayType[20];  
const short MAX=20;
```

```
interface sending_stuff {  
    string sendvalue (in ArrayType myarray, in short size);  
};
```

```
typedef sequence<long> SequenceType;  
interface sending_stuff {  
    string sendvalue (in SequenceType mysequence);  
};
```

Interface Description Language (IDL)

- An interface description language (or alternatively, interface definition language), or
- IDL for short, is a **specification language** used to describe a software component's interface.
- IDLs describe an interface in a language-independent way
- **Modules:**
- IDL module defines naming scope for a set of IDL definitions as well as Group interface and other IDL type definitions into logical name space.
- The Java Equivalent for a module is : **Package**.

Start for CS2

```
</>
1. // IDL
2. module finance
3.     interface account { . . . };
4. };
```

The code above has an interface account within the scope finance.

To access, we use : account::finance

Interface Description Language (IDL)

- **Interface**

- The basic unit for IDL is the interface whereby we define a type of an object. Within the interface, we define the attributes as well as operations.

```
</>
1. interface account {
2.     // operation and attribute definitions
3. };
```

- **Methods**

- Methods or operations are almost written the same way as Java, we place the return type, then method name followed with arguments. However, parameter passing mode must be specified using **in**, **out**, **inout**.

```
</>
1. module finance{
2.     interface account {
3.         // operations
4.         void makeDeposit (in float amount);
5.         boolean makeWithdrawal (in float amount, out float balance);
6.     }
7. }
```

For passing mode of the parameter :

- in - passed from caller to called object
- out - passed from called object to caller
- Inout - passed in both directions

Interface Description Language (IDL)

• Basic Data Types:

- short (16 bit)
- long (32-bit)
- unsigned short (16-bit)
- unsigned long (32-bit)
- long long (64-bit)
- unsigned long long (64-bit)
- Float
- double
- char (8-bit)
- wchar (16-bit)
- boolean (TRUE or FALSE)
- octet (8-bit – no conversion)
- any (arbitrary IDL type)
- string (can be bounded)

struct Type

- IDL provides a structure type **struct** that contains, as in C and C++, any number of member values of disparate types (even other structs).
- structs are useful in IDL because, unlike CORBA objects, structs are passed by value rather than by reference.
- In other words, when a struct is passed to a remote object, a copy of that struct's values is created and marshaled to the remote object.

```
</>
1.  struct Person{
2.      string name,
3.      short age,
4.      string phone
5.  }
```

Interface Description Language (IDL)

- **Attributes**

- An attribute of an IDL interface is analogous to an instance variable of a Java or C++ class, with the exception that IDL attributes always have public visibility. The general syntax for defining an attribute is this:

```
</>  
1. [readonly] attribute attribute_type attributeName;
```

Inheritance of interfaces

- IDL support the notion of inheritance.
- That is, an interface can inherit the methods and attributes of another:

```
</>  
1. interface Animal{  
2.     ...  
3. };  
4. interface Lion : Animal {  
5.     ...  
6. };
```


Interface Description Language (IDL)

- **typedef**

- Like C and C++, IDL supports a typedef statement to create a user-defined type name
- means that anywhere the SerialNumber type is found, it should be treated as a string.

```
</>  
1. typedef string SerialNumber;
```

Sequence

- An IDL sequence is simply a dynamically sizable array of values.
- These values can be dynamically inserted in or removed from the sequence; the sequence manages its size accordingly.
- All values of a sequence must be of the same type or derived from the same type (with the exception of the any type. For example:

```
</>  
1. sequence temperatureSequence;
```

Interface Description Language (IDL)

- **Array**
- An IDL array corresponds directly to the array constructs in C, C++, and Java. An array stores a known-length series of similar data types. For example:

```
</>  
1. string DayNames[7];
```

- **IDL Comments**
- Comments in IDL follow the same conventions as Java and C++.

```
</>  
1. // this is a comment  
2.  
3. /* this is a different comment */
```



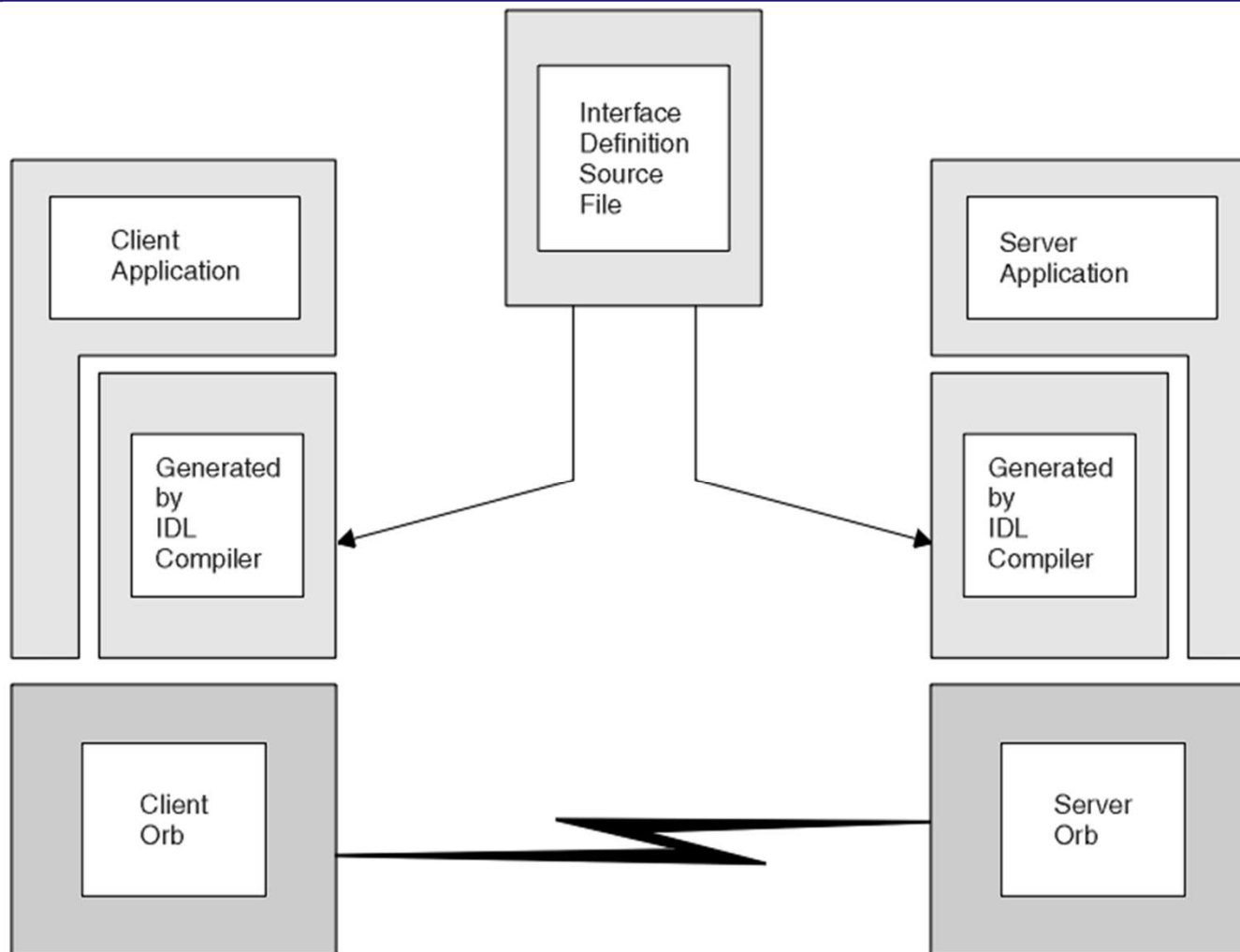
Interface Description Language (IDL)

- **Exception**

- The IDL exception type itself is similar to a struct type; it can contain various data members (but no methods).
- For Example for the code below, the ***browseObject()*** method accepts a single parameter, called object, of type ***any***.
- A client calling this method can use an object of any IDL type to fill this parameter. Internally to ***browseObject()***, the method will attempt to determine what the actual type of object is.
- If it determines that object is a type of object it can interact with, it will do so.
- Otherwise, it will raise the ***UnknownObjectType*** exception, which is returned to the caller.

```
</>
1. interface ObjectBrowser {
2.     exception UnknownObjectType {
3.         string message;
4.     };
5.     void browseObject(in any object) raises (UnknownObjectType);
6. };
```

CORBA IDL



Interface Description Language (IDL)

CORBA IDL to Java bindings

- Module in IDL maps to package in Java
- CORBA IDL types map to most Java types
 - boolean to Boolean
 - octet to Byte
 - short to short
 - long to int
 - long long to long
 - string to string
 - sequences and arrays to arrays
 - float to float
 - double to double
 - enums, structs, unions and exceptions to Java classes
- For user defined types in IDL, Java generates:
 - Helper class – manages read/write to CORBA I/O streams
 - Holder class – Implements out/inout parameters as a wrapper class, to hold their value (by reference) and pass it to Helper class streams



Interface Description Language (IDL)

Attributes

- Attribute gets/sets a particular variable on the servant
- Example:
 - A remote procedure named `the_value` with no parameters passed, that returns a Short value
 - A remote procedure named `the_value` with a Short value passed as a parameter, that does not return any values (has a void value as the return value)
- Not recommended to use as behavior is too dependent on network and language implementations.

```
interface sending_stuff {  
    attribute short the_value;  
};
```

Interface Description Language (IDL)

Example code – in

- IDL definition

```
typedef short ArrayType[20];
const short MAX=20;

interface sending_stuff {
    string sendvalue (in ArrayType myarray, in short size);
};
```

- Java code (servant)

```
public class myreceiver extends sending_stuffPOA
{
    public String sendvalue (short[] myarray, short size)
    {
        int the_real_size;
        the_real_size=size;
        if (the_real_size > MAX.value)
            the_real_size=MAX.value;
        for(int i=0;i<the_real_size;i++)
        {
            System.out.println("myarray["+i+"] is "+myarray[i]);
        }
        String mymsg="got here";
        return mymsg;
    }
}
```

```
public interface MAX
{
    public static final short value = (short)(20);
}
```

Interface Description Language (IDL)

Example code – inout

- IDL definition
- Java code (servant)

```
typedef long ArrayType[20];
const long MAX=20;

interface sending_stuff {
    string sendvalue (inout ArrayType myarray, in long size);
};

public class myreceiver extends sending_stuffPOA
{
    public String sendvalue (ArrayTypeHolder myarray, int size)
    {
        int the_real_size;
        the_real_size=size;
        if (the_real_size > MAX.value)
            the_real_size=MAX.value;
        for(int i=0;i<the_real_size;i++)
        {
            System.out.println("Original value of myarray["+i+"] is "+myarray.value[i]);
            myarray.value[i]=myarray.value[i]*10; // Multiply each myarray value times 10
            System.out.println("Value to pass back to client of myarray["+i+"] is "+myarray.value[i]);
        }
        String mymsg="Hello there from servant to client";
        return mymsg;
    }
}
```

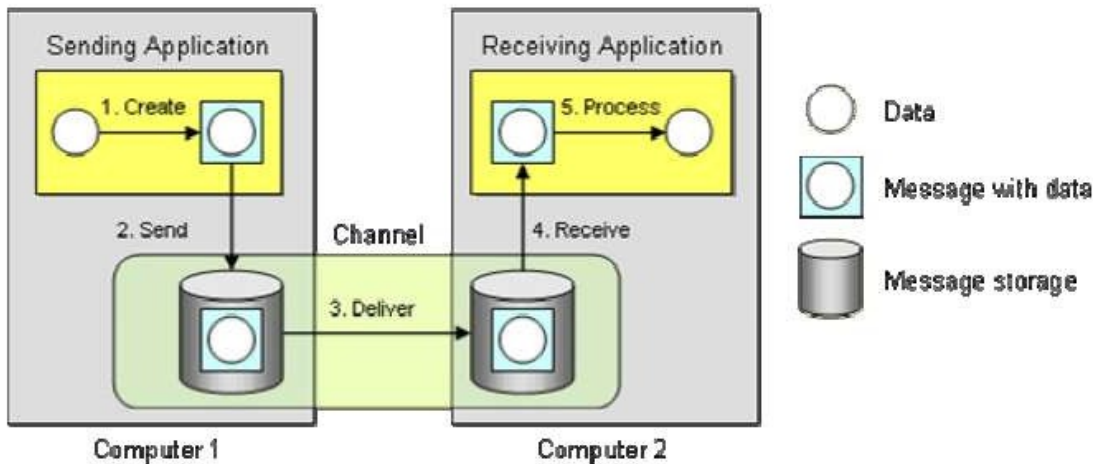

What is Messaging?

- A simple way to understand what messaging does is to consider the telephone system.
- A telephone call is a synchronous form of communication.
- I can only communicate with the other party if the other party is available at the time I place the call.
- Voice mail on the other hand, allows asynchronous communication.
- With voice mail, when the receiver does not answer, the caller can leave a message and later the receiver (at his convenience) can listen to the messages queued in his mailbox.
- Messaging is a technology that enables high-speed, asynchronous, program-to-program communication with reliable delivery.
- **Programs communicate** by sending packets of **data called messages** to each other.
- **Channels, also known as queues**, are logical pathways that connect the programs and convey messages.
- A channel behaves like a collection or array of messages, but one that is magically shared across multiple computers and can be used concurrently by multiple channel.

What is Messaging?

- A **receiver or consumer** is a program that receives a message by reading (and deleting) it from a channel.
- Message is a simple sort of data structure—such as a string, a byte array, a record, or an object.
- A message actually contains two parts, **a header and a body**.
- The header contains meta-information about the message—who sent it, where it's going, etc.
- Header information is used by the messaging system.
- Body contains the data being transmitted.
- In essence, a message is transmitted in five steps:
 - 1. **Create** - The sender creates the message and populates it with data.
 - 2. **Send** - The sender adds the message to a channel.
 - 3. **Deliver** - The messaging system moves the message from the sender's computer to the receiver's computer, making it available to the receiver.
 - 4. **Receive** - The receiver reads the message from the channel.
 - 5. **Process** - The receiver extracts the data from the message.

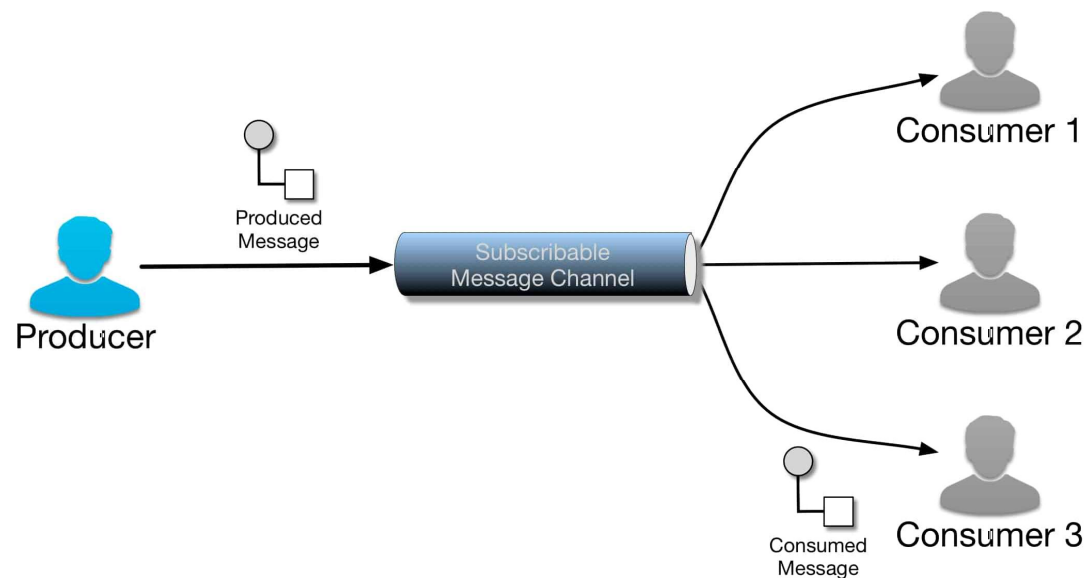
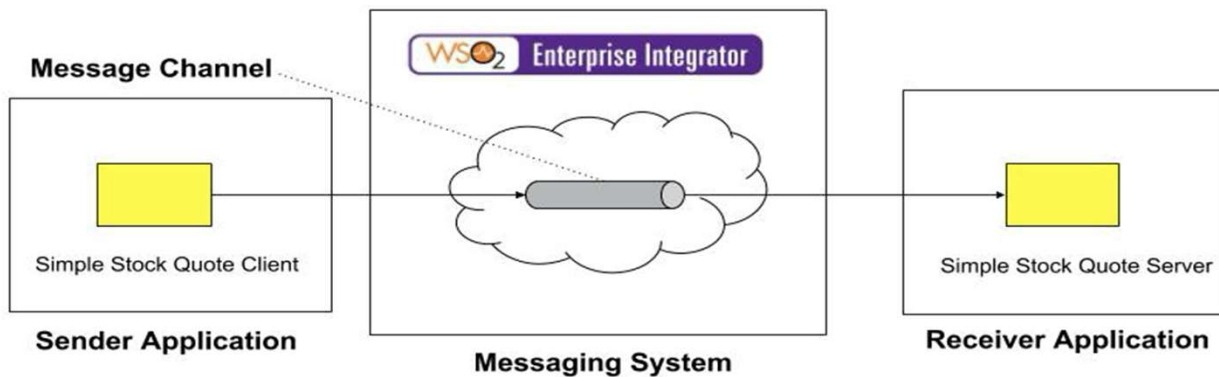
What is Messaging?



This diagram also illustrates two important messaging concepts:

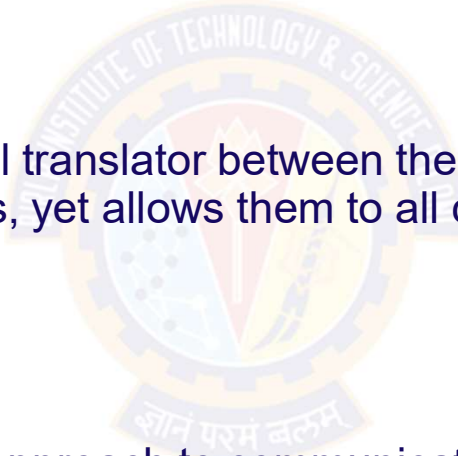
- Send and forget — In step 2, the sending application sends the message to the message channel.
- Once that send is complete, the sender can go on to other work while the messaging system transmits the message in the background.
- The sender can be confident that the receiver will eventually receive the message and does not have to wait until that happens.
- Store and forward — In step 2, when the sending application sends the message to the message channel, the messaging system stores the message on the sender's computer, either in memory or on disk.
- In step 3, the messaging system delivers the message by forwarding it from the sender's computer to the receiver's computer, and then stores the message once again on the receiver's computer.
- This store-and-forward process may be repeated many times, as the message is moved from one computer to another, until it reaches the receiver's computer.

What is Messaging?



Why Messaging?

- Remote Communication
 - Messaging enables separate applications to communicate and transfer data.
 - Two objects that reside in the same process can simply share the same data in memory.
- Platform/Language Integration
 - A messaging system can be a universal translator between the applications that works with each one's language and platform on its own terms, yet allows them to all communicate through a common messaging paradigm
- Asynchronous Communication
 - Messaging enables a send and forget approach to communication.
 - The sender does not have to wait for the receiver to receive and process the message

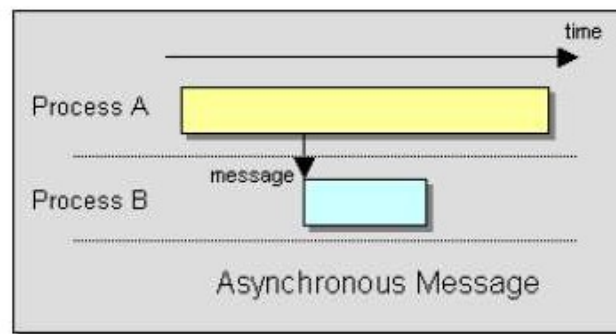
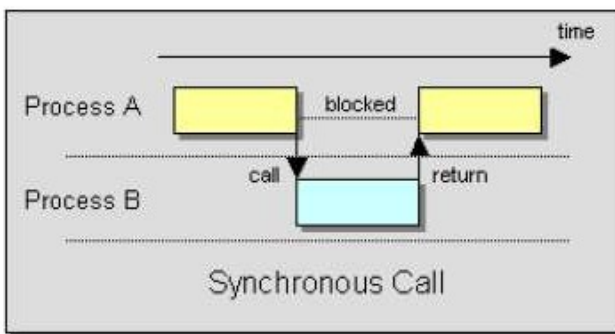


Why Messaging?

- Variable Timing
 - Messaging allows the sender to batch requests to the receiver at its own pace, and for the receiver to consume the requests at its own different pace.
- Throttling
 - Asynchronous communication enables the receiver to control the rate at which it consumes requests, so as not to become overloaded by too many simultaneous requests.
- Reliable Communication
 - Messaging provides reliable delivery that a remote procedure call (RPC) cannot.
- Disconnected Operation
 - Some applications are specifically designed to run disconnected from the network, yet to synchronize with servers when a network connection is available.

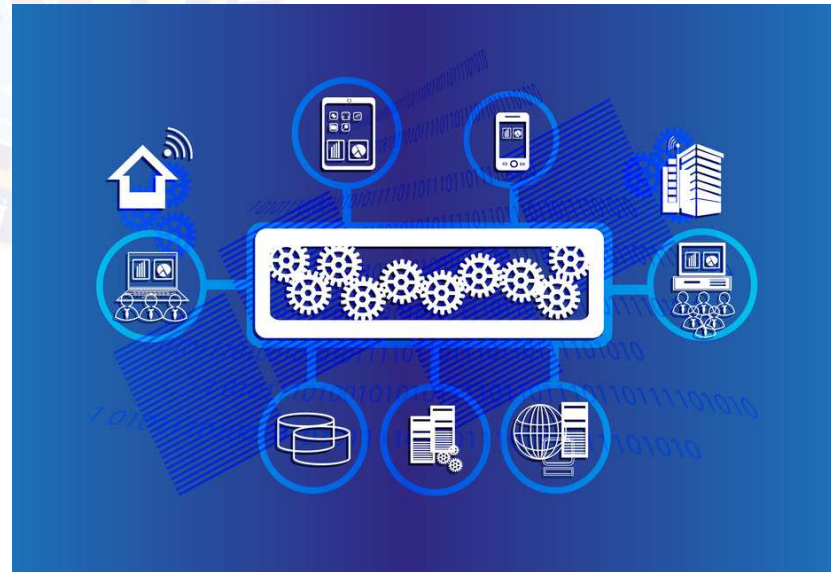
Why Messaging?

- Mediation
- The messaging system acts as a mediator—as in the Mediator pattern - between all of the programs that can send and receive messages.
- Thread Management
- Asynchronous communication means that one application does not have to block while waiting for another application to perform a task, unless it wants to.



Integrated Applications

- Integrated applications are independent applications that can each run by itself, but coordinate with each other in a loosely coupled way.
- This enables each application to focus on one comprehensive set of functionality and yet delegate to other applications for related functionality.
- Integrated applications communicating asynchronously don't have to wait for a response.
- They can proceed without a response or perform other tasks concurrently until the response is available.
- Integrated applications tend to have a broad time constraint, such that they can work on other tasks until a result becomes available.

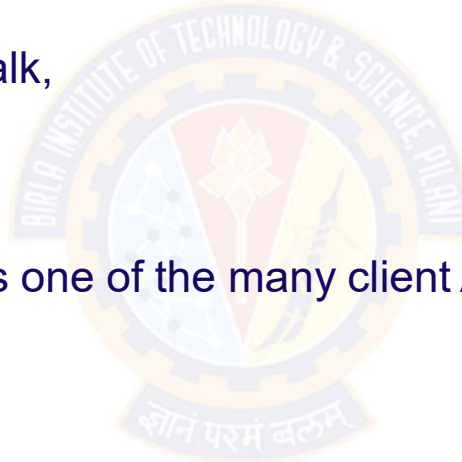


Commercial Messaging Systems

- The apparent benefits of integrating systems using an asynchronous messaging solution has opened up a significant market for software vendors creating messaging middleware and associated tools.
- We can group the messaging vendors' products into the following four categories:
- 1. Operating Systems :
- Messaging has become such a common need that vendors have started to integrate the necessary software infrastructure into the operating system or database platform.
- For example, the Microsoft Windows 2000 and Windows XP operating systems include the Microsoft Message Queuing (MSMQ) service software.
- Oracle offers Oracle AQ as part of its database platform.
- 2. Application Servers:
- Sun Microsystems first incorporated the Java Messaging Service (JMS) into version 1.2 of the J2EE specification.
- Now all J2EE application servers such as IBM WebSphere, BEA WebLogic, etc. provide an implementation for this specification.

Commercial Messaging Systems

- 3. EAI Suites:
 - Products from these vendors offer proprietary—but functionally rich—suites that encompass messaging, business process automation, workflow, portals, and other functions.
 - Key players in this marketplace are
 1. IBM WebSphere MQ, Microsoft BizTalk,
 2. TIBCO, WebMethods, SeeBeyond,
 3. Vitria, CrossWorlds, and others.
 - Many of these products include JMS as one of the many client API's they support.
- 4. Web Services Toolkits:
 - Web services have garnered a lot of interest in the enterprise integration communities.
 - Standards bodies and consortia are actively working on standardizing reliable message delivery over web services (i.e., WS-Reliability, WS-ReliableMessaging, and ebMS).
 - A growing number of vendors offer tools that implement routing, transformation, and management of web services-based solutions



Enterprise Application Integration (EAI)

Introduction

- An integration framework composed of a collection of technologies and services which form a middleware or "middleware framework" to enable integration of systems and applications across an enterprise
- *"unrestricted sharing of data and business processes among any connected application or data sources in the enterprise" - Gartner*
- Application of EAI
 - Data integration
 - Vendor/3rd party independence
 - Common façade
- Patterns
 - Mediation
 - Federation



Messaging System

Message Channels - Basic Messaging Concepts

- Messaging involves certain basic concepts.
- Once you understand these concepts, you can make sense of the technology even before you understand all of the details about how to use it.
- These basic messaging concepts are:
- 1. Channels — Messaging applications transmit data through a Message Channel, a virtual pipe that connects a sender to a receiver.
- A newly installed messaging system doesn't contain any channels; you must determine how your applications need to communicate and then create the channels to facilitate it.
- 2. Messages — A Message is an atomic packet of data that can be transmitted on a channel.
- Thus to transmit data, an application must break the data into one or more packets, wrap each packet as a message, and then send the message on a channel.
- Likewise, a receiver application receives a message and must extract the data from the message to process it.
- The message system will try repeatedly to deliver the message (e.g., transmit it from the sender to the receiver) until it succeeds.

Messaging System - Enterprise Application Integration (EAI)

Message Channels - Basic Messaging Concepts

- 3. Multi-step delivery — In the simplest case, the message system delivers a message directly from the sender's computer to the receiver's computer.
- However, actions often need to be performed on the message after it is sent by its original sender but before it is received by its final receiver.
- For example, the message may have to be validated or transformed because the receiver expects a different message format than the sender.
- A Pipes and Filters architecture describes how multiple processing steps can be chained together using channels.
- 4. Routing — In a large enterprise with numerous applications and channels to connect them, a message may have to go through several channels to reach its final destination.
- The route a message must follow may be so complex that the original sender does not know what channel will get the message to the final receiver.
- Instead, **the original sender sends the message to a Message Router.**
- An application component and filter in the pipes-and-filters architecture, which will determine how to navigate the channel topology and direct the message to the final receiver, or at least to the next router.

Messaging System - Enterprise Application Integration (EAI)

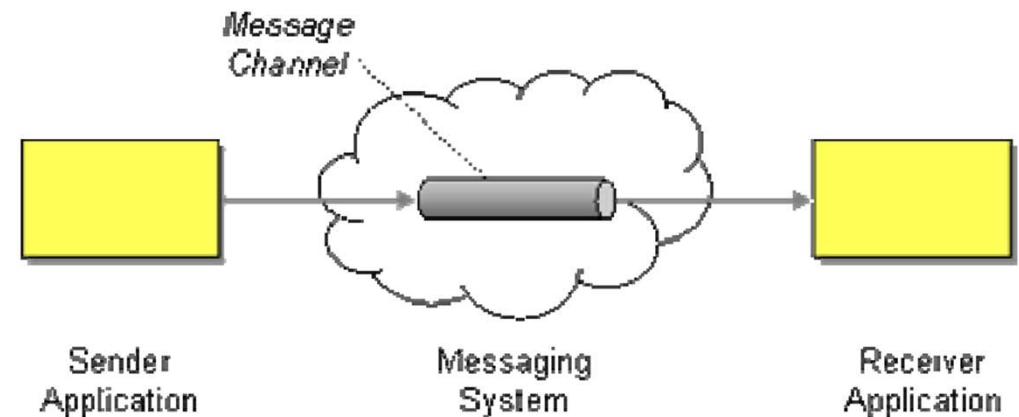
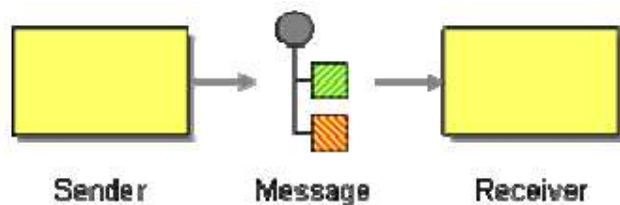
Message Channels - Basic Messaging Concepts

- 5. Transformation — Various applications may not agree on the format for the same conceptual data
- The sender formats the message one way, yet the receiver expects it to be formatted another way.
- To reconcile this, the message must go through an intermediate filter, a Message Translator, that converts the message from one format to another.
- 6. Endpoints — An application does not have some built-in capability to interface with a messaging system.
- Rather, it must contain a layer of code that knows both how the application works and how the messaging system works, bridging the two so that they work together.
- This bridge code is a set of coordinated Message Endpoints that enable the application to send and receive messages.

Enterprise Application Integration (EAI)

Message Channels - Overview

- Message Channel works as a **logical addressing system** in Messaging
- Messaging Channel acts as a medium of communication between two specific applications in a complex enterprise, where in the sender application **selects which particular "channel" of the messaging system** to use, that designates the target application/ group of applications.
- Design challenges
 - One to one or one to many
 - Data type channel
 - Invalid and dead message handling
 - Crash proof design / Guaranteed delivery
 - Non-messaging client access - Channel adapter
 - Communications backbone – Message bus



Enterprise Application Integration (EAI)

Message Channels - Overview

- An enterprise has two separate applications that need to communicate, preferably by using Messaging.
- **How does one application communicate with another using messaging?**
- Once a group of applications have a messaging system available, it's tempting to think that any application can communicate with any other application any time desired.
- Yet the messaging system does not magically connect all of the applications.
- The application sending out the information knows what sort of information it is, and the applications that would like to receive information aren't looking for just any information, but for particular sorts of information they can use.
- Messaging system is a set of connections that enable applications to communicate by transmitting information in predetermined, predictable ways.
- Connect the applications using a Message Channel, where one application writes information to the channel and the other one reads that information from the channel.
- When an application has information to communicate, it doesn't just fling the information into the messaging system, it adds the information to a particular Message Channel.
- An application receiving information doesn't just pick it up at random from the messaging system; it retrieves the information from a particular Message Channel.

Message Channels - Overview

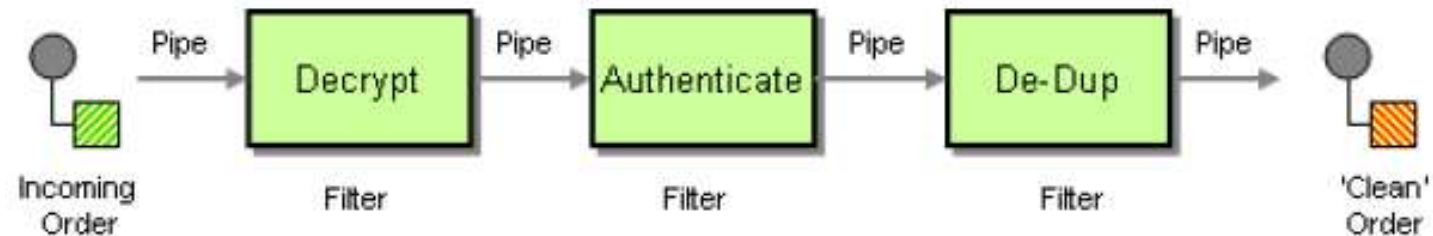
- A message consists of two basic parts:
- 1. Header – Information used by the messaging system that describes the data being transmitted, its origin, its destination.
- 2. Body – The data being transmitted; generally ignored by the messaging system and simply transmitted as-is.
- This concept is not unique to messaging. Both postal service mail and e-mail send data as discrete mail messages.
- An Ethernet network transmits data as packets, as does the IP part of TCP/IP such as the Internet.
- Streaming media on the Internet is actually a series of packets.
- To the messaging system, all messages are the same: Some body of data to be transmitted as described by the header.
- Use a Command Message to invoke a procedure in another application.
- Use a Document Message to pass a set of data to another application.
- Use an Event Message to notify another application of a change in this application.
- If the other application should send a reply back, use Request-Reply.
- If an application wishes to send more information than one message can hold, break the data into smaller parts and send the parts as a Message Sequence.
- If the data is only useful for a limited amount of time, specify this use-by time as a Message Expiration.
- Since all the various senders and receivers of messages must agree on the format of the data in the messages, specify the format as a Canonical Data Model.

Message Channels - Overview

- Pipes and Filters
- In many enterprise integration scenarios, a single event triggers a sequence of processing steps, each performing a specific function.
- For example, **let's assume a new order arrives in our enterprise in the form of a message.**
- One requirement may be that the message is encrypted to prevent eavesdroppers from spying on a customer's order.
- A second requirement is that the messages contain authentication information in the form of a digital certificate to ensure that orders are placed only by trusted customers.
- In addition, duplicate messages could be sent from external parties (remember all the warnings on the popular shopping sites to click the 'Order Now' button only once?).
- To avoid duplicate shipments and unhappy customers, we need to eliminate duplicate messages before subsequent order processing steps are initiated.
- To meet these requirements, we need to transform a stream of possibly duplicated, encrypted messages containing extra authentication data into a stream of unique, simple plain-text order messages without the extraneous data fields.
- Use **the Pipes and Filters architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes).**

Message Channels - Overview

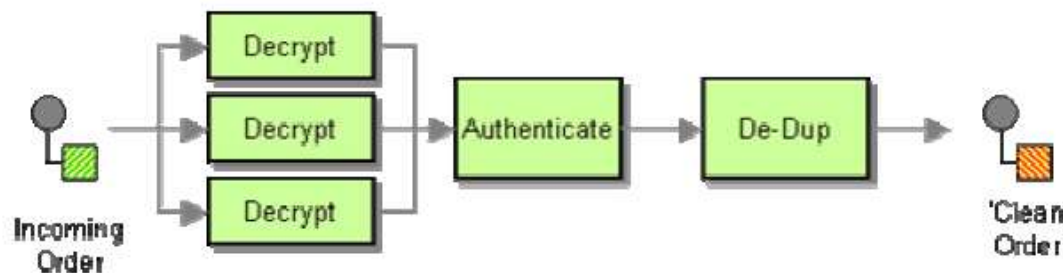
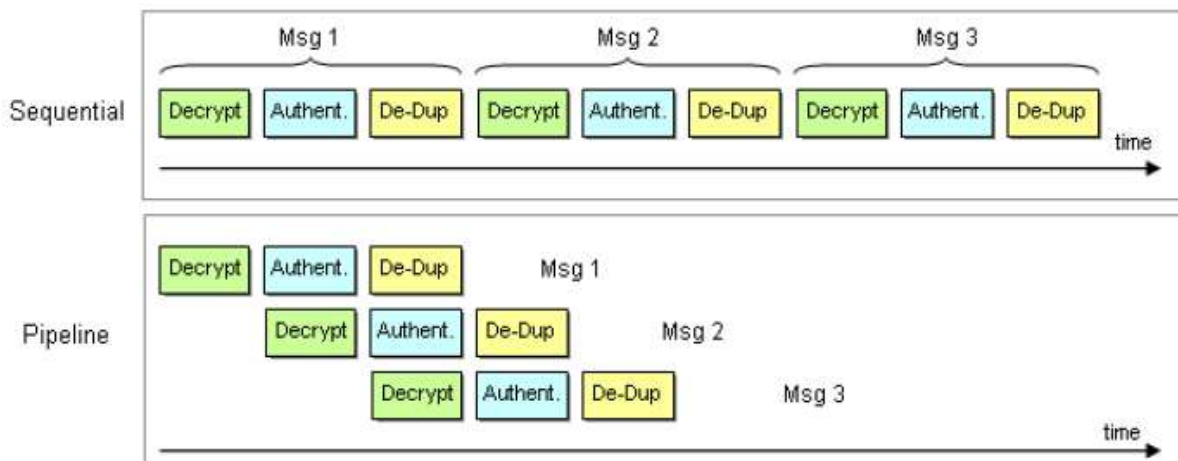
- Pipes and Filters



- Each filter exposes a very simple interface:
 - It receives messages on the inbound pipe,
 - Processes the message, and
 - Publishes the results to the outbound pipe.
 - The pipe connects one filter to the next, sending output messages from one filter to the next.
 - The connection between filter and pipe is sometimes called port.
 - In the basic form, each filter component has one input port and one output port.
-
- When applied to our example problem, the Pipes and Filters architecture results in three filters, connected by two pipes (see picture).
 - We need one additional pipe to send messages to the decryption component and one to send the clear-text order messages from the de-duper to the order management system.
 - This makes for a total of four pipes.

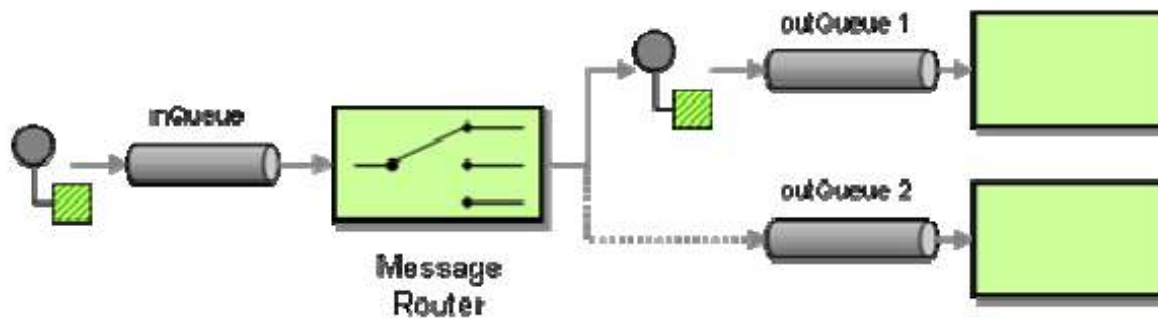
Message Channels – Pipes and Filter

- Pipes and Filters describe a fundamental architectural style for messaging systems:
- Individual processing steps ("filters") are chained together through the messaging channels ("pipes").



Message Channels - Overview

- Message Router Pattern
- How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?
- Insert a special filter
- a Message Router, which consumes a Message from one Message Channel and
- republishes it to a different Message Channel depending on a set of conditions.

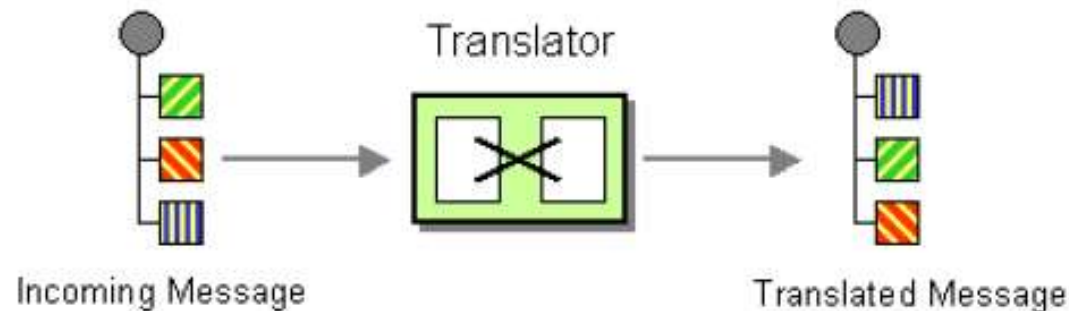


Message Channels - Overview

- Message Router Pattern
- An advantage of the Pipes and Filters is the composability of the individual components.
- It allows us to insert additional steps into the chain without having to change existing components.
- This opens up the option of decoupling two filters by inserting another filter in between that determines what step to execute next.
- **The Message Router differs from the most basic notion of Pipes and Filters** in that it connects to multiple output channels.
- The components surrounding the Message Router are completely unaware of the existence of a Message Router.
- A key property of the Message Router is that **it does not modify the message contents**.
- It only concerns itself with the destination of the message.
- The key benefit of using a Message Router is that the decision criteria for the destination of a message are maintained in a single location.
- If new message types are defined, new processing components are added, or the routing rules change, we need to change only the Message Router logic and all other components remain unaffected.
- Also, since all messages pass through a single Message Router, incoming messages are guaranteed to be processed one-by-one in the correct order.

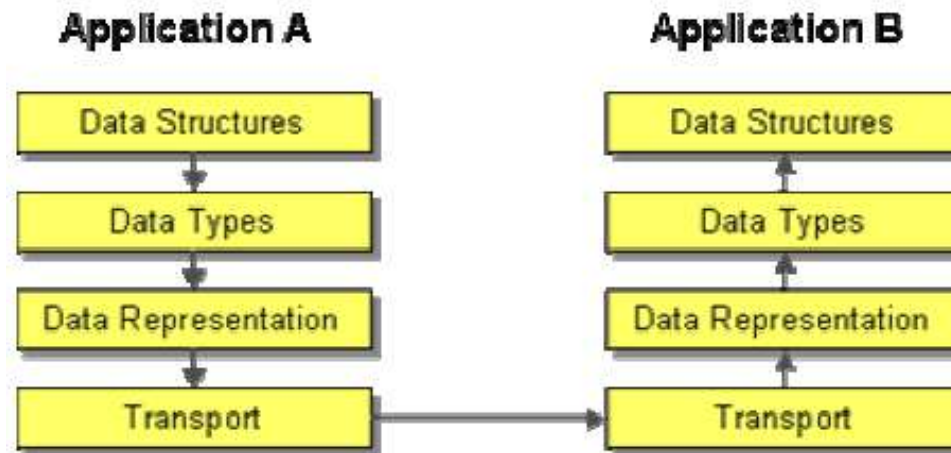
Message Channels - Overview

- Message Translator Pattern
- How can systems using different data formats communicate with each other using messaging?
- Use a special filter, a Message Translator, between other filters or applications to translate one data format into another.
- The Message Translator is the messaging equivalent of the Adapter pattern described in [GoF].
- An adapter converts the interface of a component into a another interface so it can be used in a different context.



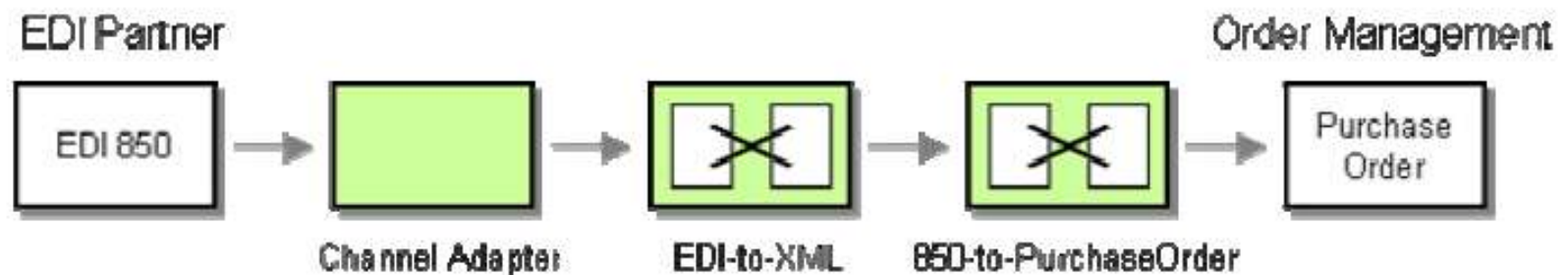
Message Channels - Overview

- Message Translator Pattern – Chaining Transformation
- For example, let's assume an EDI 850 Purchase Order record represented as a fixed-format file has to be translated to an XML document sent over http to the order management system which uses a different definition of the *Order* object.
- The required transformation spans all four levels:
- the transport changes from a file to HTTP,
- the data format changes from fixed-format to XML,
- data types and data formats have to be converted to comply with the *Order* object defined by the order management system.
- The beauty of a layered model is that we can treat one layer without regard to the lower layers and therefore can choose to work at different levels of abstraction.



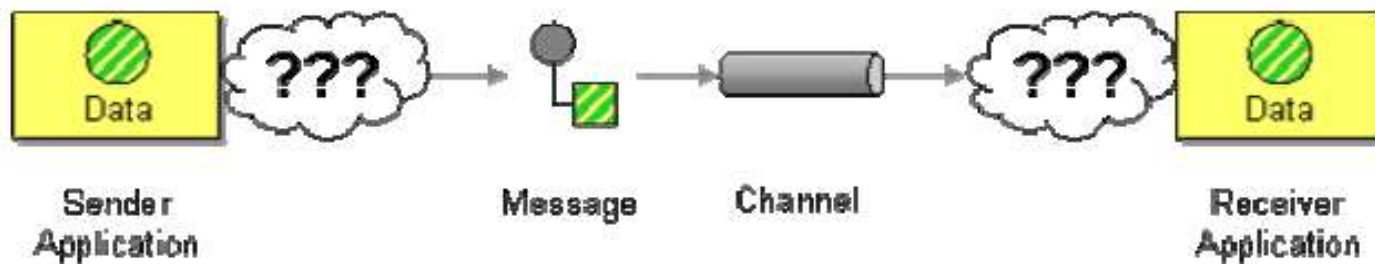
Message Channels - Overview

- Message Translator Pattern – Chaining Transformation
- Chaining multiple *Message Translator* units using *Pipes and Filters* results in the following architecture (see picture).
- Creating one *Message Translator* for each layer allows us to reuse these components in other scenarios.
- For example, the *Channel Adapter* and the *EDI-to-XML Message Translator* can be generic enough to deal with any incoming EDI document.



Message Channels - Overview

- Message End Point Pattern
- Applications are communicating by sending Messages to each other via Message Channels.
- How does an application connect to a messaging channel to send and receive messages?
- The application and the messaging system are two separate sets of software.
- The application provides functionality for some type of user
- The messaging system manages messaging channels for transmitting messages for communication.
- Because the application and the messaging system are separate, they must have a way to connect and work together.

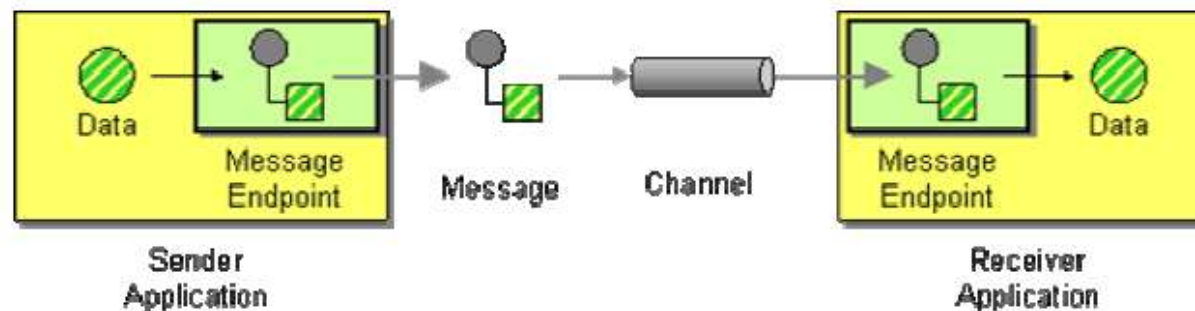


Applications disconnected from a message channel

Message Channels - Overview

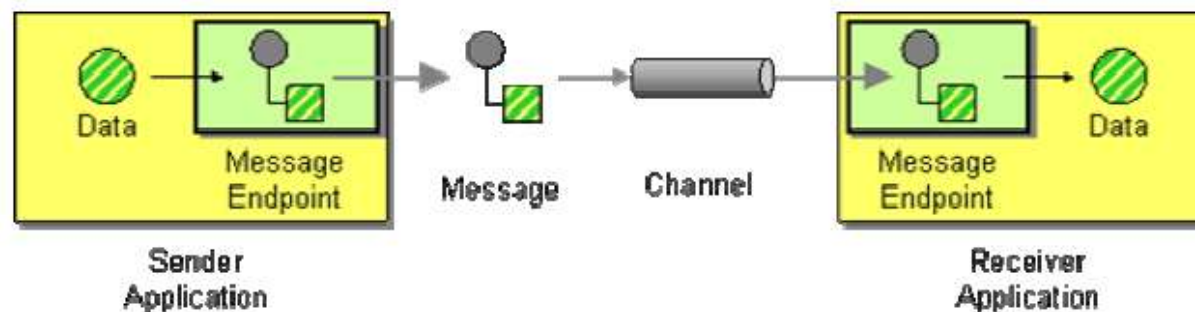
Message End Point Pattern

- A messaging system is a type of server, capable of taking requests and responding to them.
- Like a database accepting and retrieving data, a messaging server accepts and delivers messages.
- A messaging system is a messaging server.
- A server needs clients, and an application that uses messaging is a client of the messaging server.
- The messaging server, like a database server, has a client API that the application can use to interact with the server.
- The API is not application-specific; it is domain-specific, where messaging is the domain.
- The application must contain a set of code that connects and unites the messaging domain with the application to allow the application to perform messaging.
- Connect an application to a messaging channel using a Message Endpoint.
- A client of the messaging system that the application can then use to send or receive messages.



Message Channels - Overview

- Message End Point Pattern
- Message Endpoint code is custom to both the application and the messaging system's client API.
- The Message Endpoint encapsulates the messaging system from the rest of the application, and customizes a general messaging API for a specific application and task.
- A Message Endpoint can be used to send messages or receive them, but one instance does not do both.
- A Message Endpoint is a specialized Channel Adapter, one that has been custom developed for and integrated into its application.
- A Message Endpoint should be designed as a Messaging Gateway to encapsulate the messaging code and hide the message system from the rest of the application.





Thank You!

Any Questions?
