BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Introduction to Middleware

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

Introduction to Middleware

- What is a Distributed Transaction ?
- What is Middleware ?
- Middleware in Distributed computing
- Different forms of Middleware

What is a Transaction ?

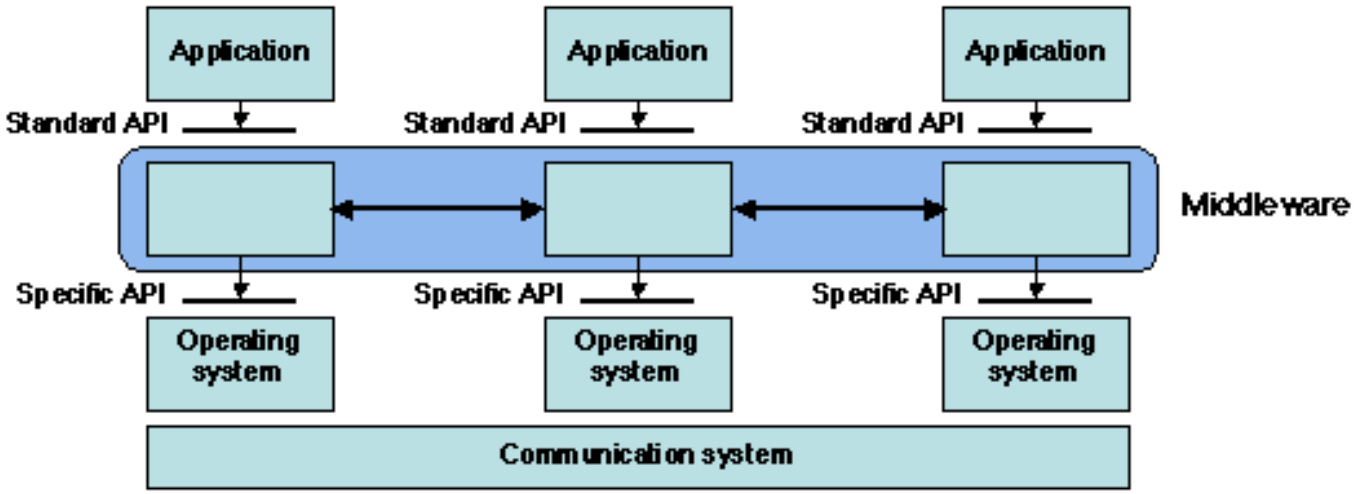
Transaction

- “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)

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.



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

Thank You!

In our next session:

Socket Data structures

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Sockets Overview

Srikanth Gunturu

Guest Faculty
BITS, WILP

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 – 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

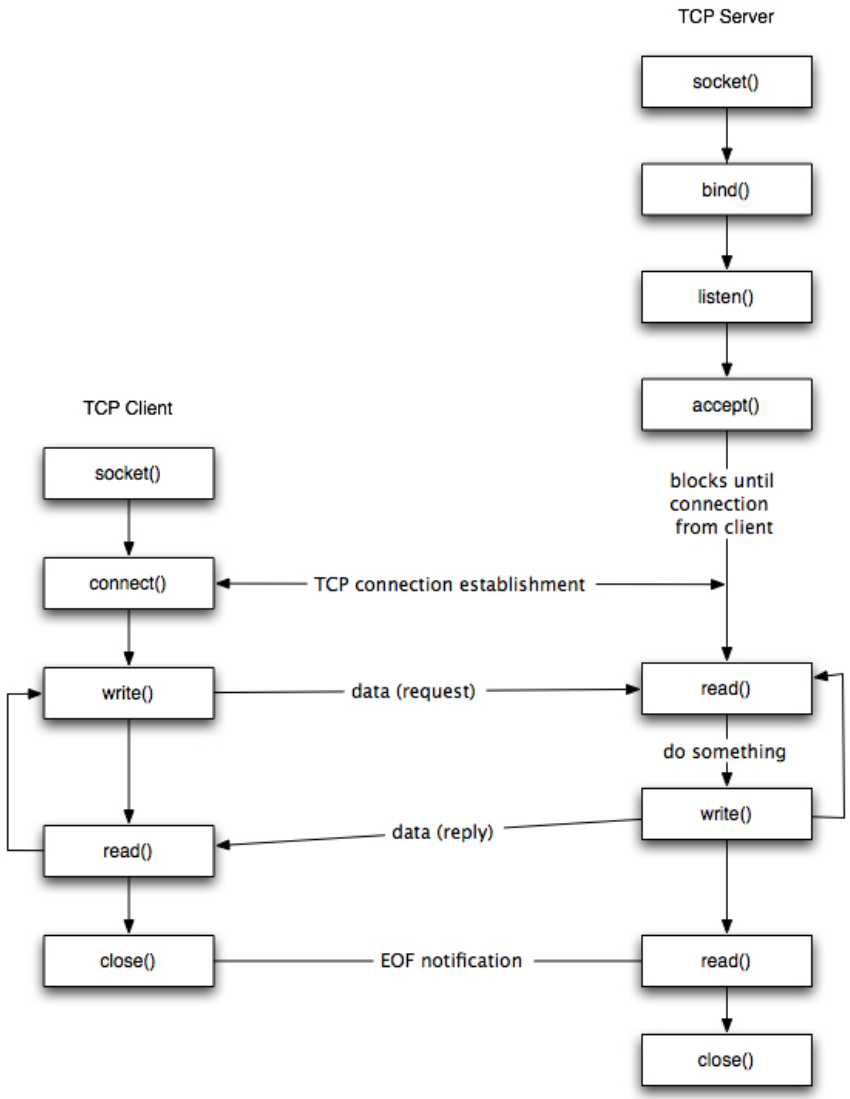
getpeername —after a connection arrives, obtain the remote machine's endpoint address from a socket

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

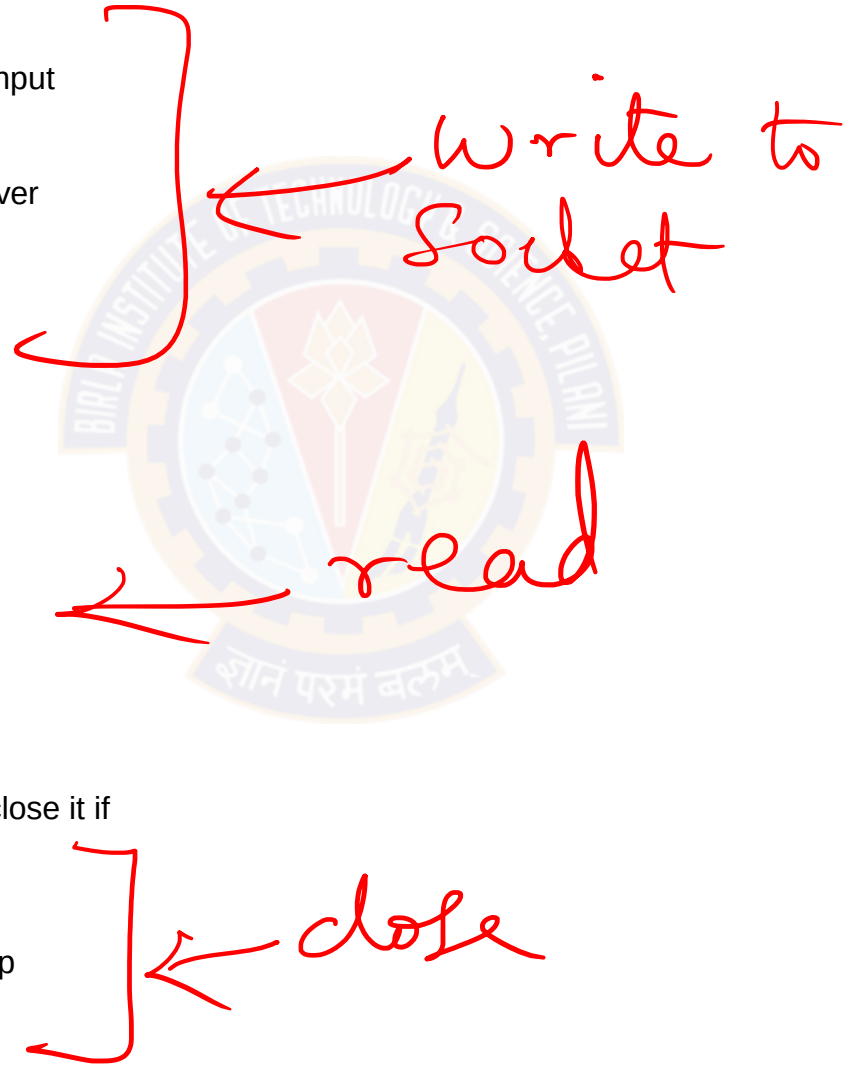
```
int main()
{
int socket_fd;
struct sockaddr_in serv_addr;
int port_number;
char * IP_address;
char client_message[256];
int message_result;
// Set the IP address (IPv4)
IP_address = new char [sizeof(“ 127.0.0.1” )];
strcpy(IP_address, “ 127.0.0.1” ); // could instead have
copied “ localhost”
 // 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);
```

IPv4

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;
}
```



Thank You!

In our next session:

Early Middleware Technologies

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Early Middleware Technologies

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

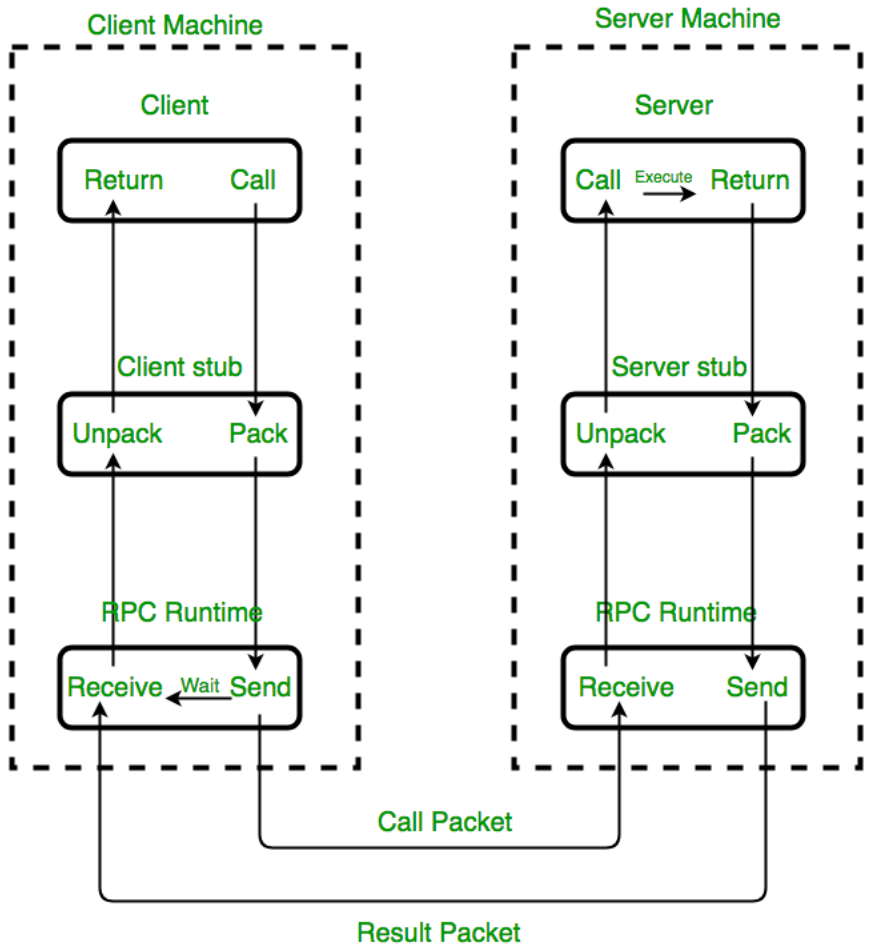
Early Middleware Technologies

- 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**

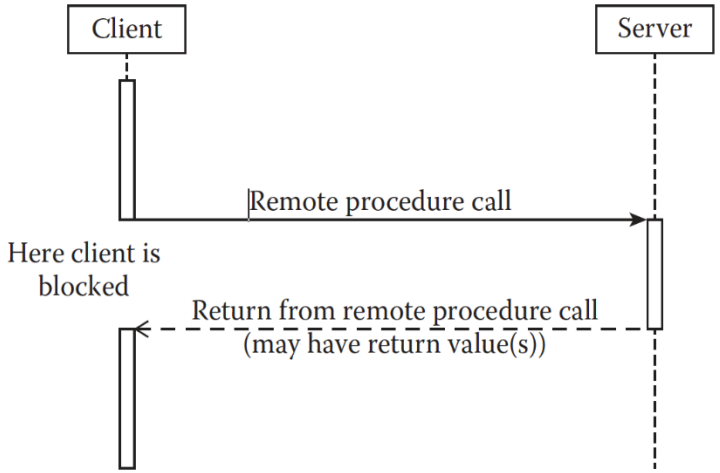


Implementation of RPC mechanism

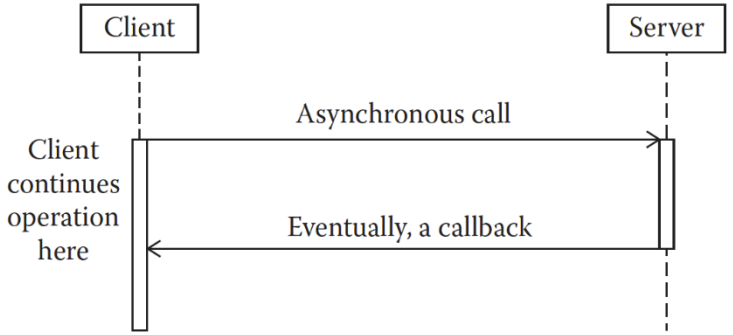
Remote Procedure Calls (RPC)

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



Synchronous RPC



Asynchronous RPC

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

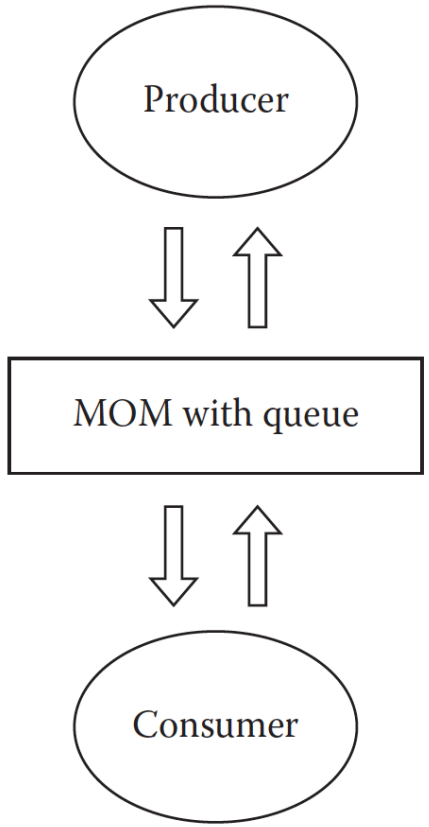
Component or Object Oriented Middleware ?

- 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.



Thank You!

In our next session:

Object Oriented Middleware – CORBA Basics

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CORBA Basics

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

CORBA Overview

- CORBA Introduction
- CORBA Basics

CORBA

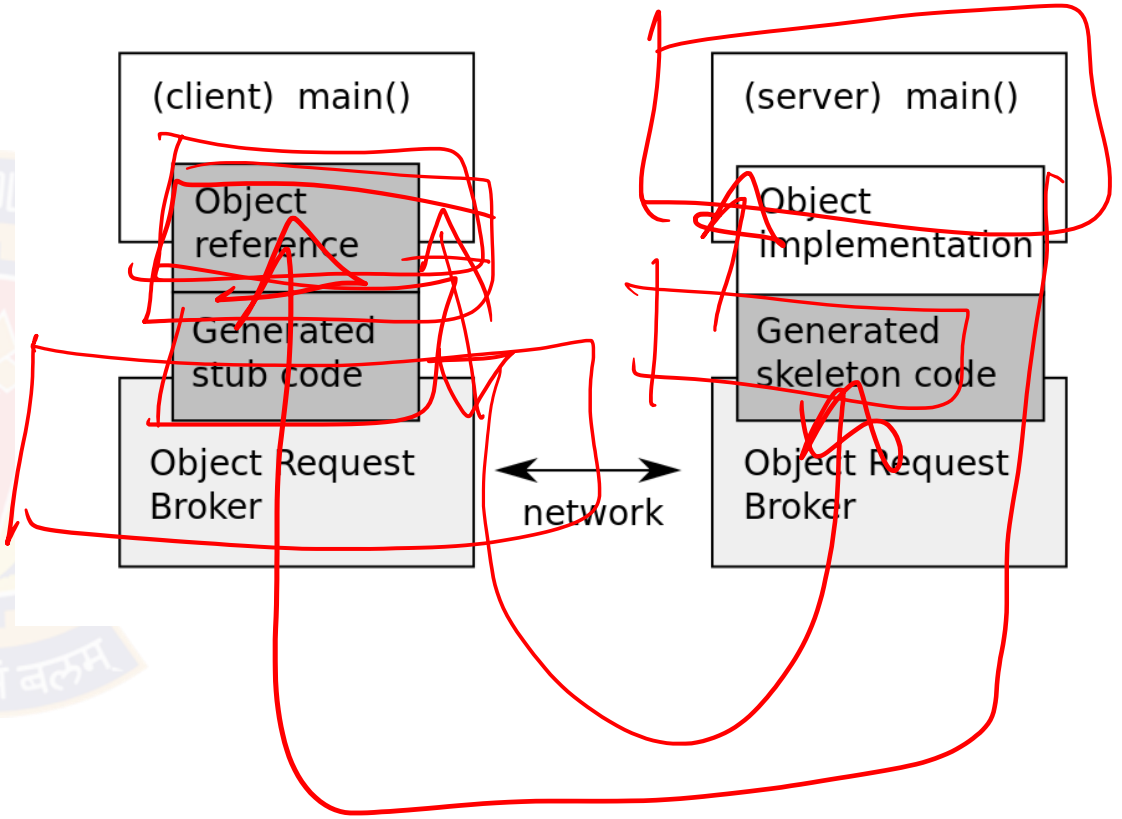
Evolution and History

- 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

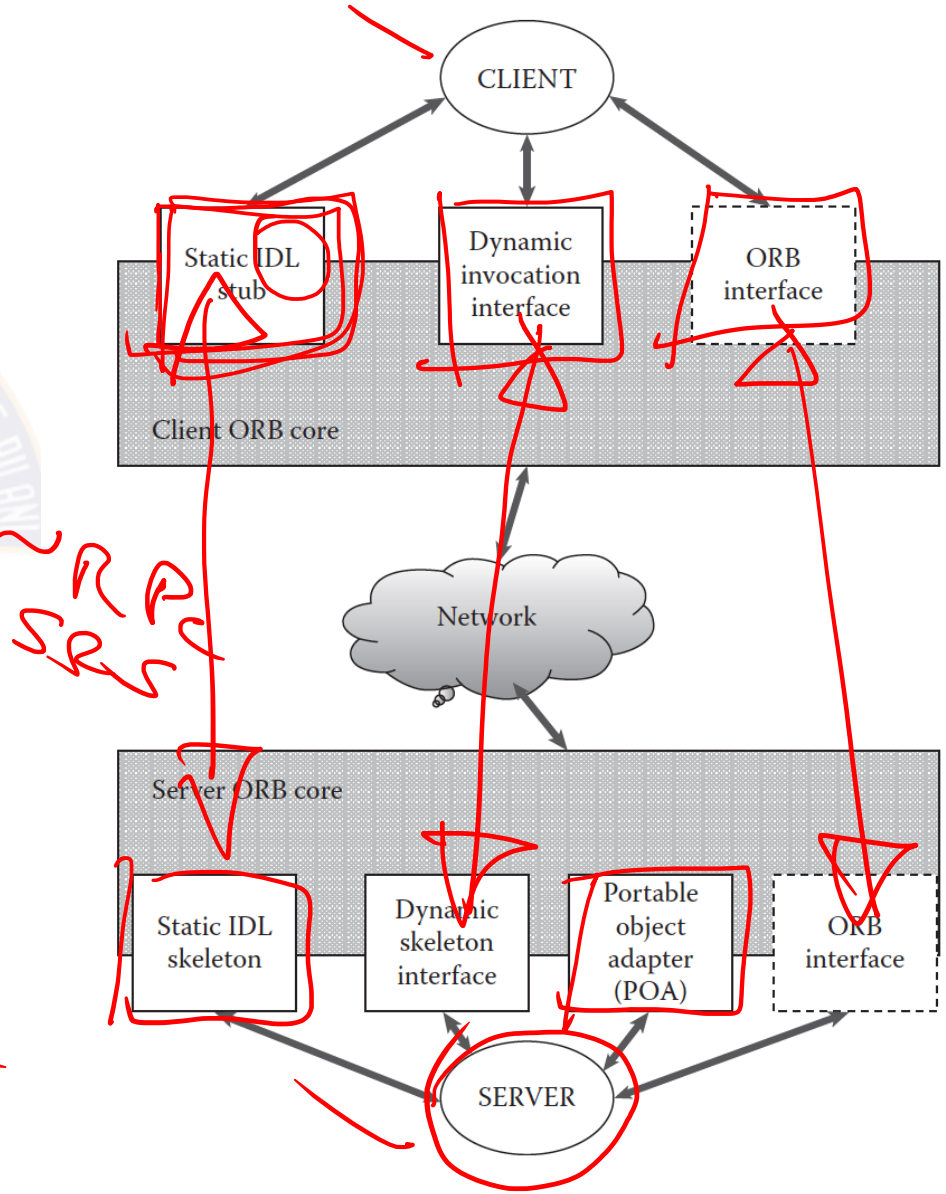
- Primarily client-server oriented
- Can be seen as RPC on remotely located objects (due to distributed OO nature)
- Uses General Inter-Orb Protocol (on TCP/IP) to communicate and transfer data between client and server
- Client side proxy object
  - Represents remote object in local client space
  -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



CORBA

Architecture

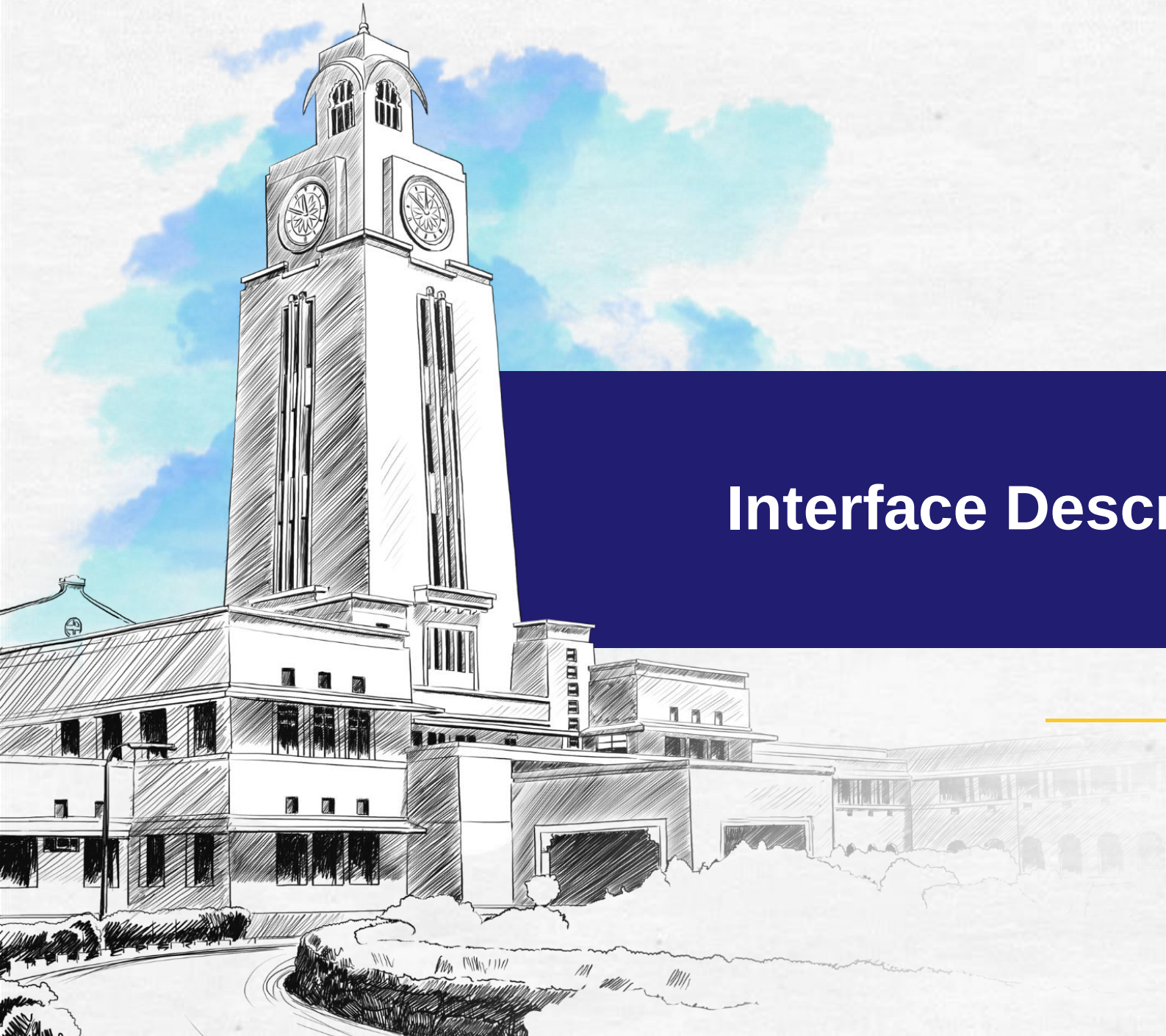
- CORBA supports static and dynamic interfaces using Interface Description language IDL
  - Static interface defined at compile time using IDL compiler
  - Dynamic interface that defines invocation at runtime
- IDL stub is client side implementation that calls CORBA library API for client operation
- 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.



Thank You!

In our next session:

Interface Description Language (IDL)



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Interface Description Language (IDL)

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

CORBA IDL

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

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

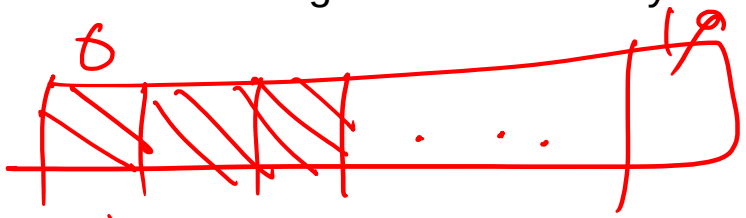
```
struct theData {
  long firstvalue;
  string secondvalue;
};

typedef theData StructType;
```

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

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

- Arrays as params
  - No guarantee on array indexing



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

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

- Sequence as params
  - Sends only the data present in the sequence at the time of method call

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

Interface Description Language (IDL)

Attributes

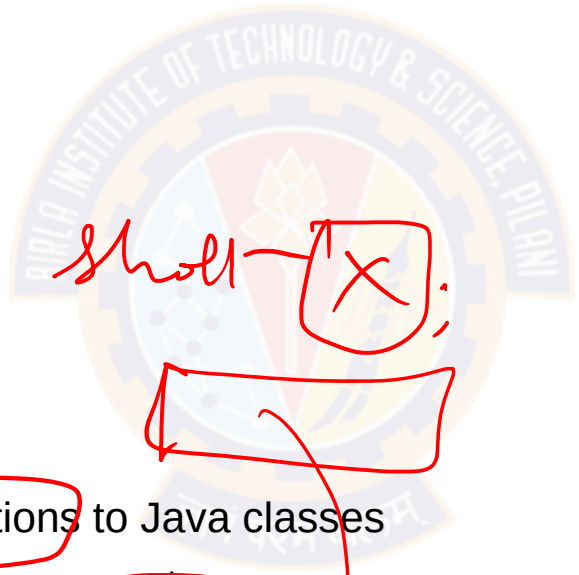
- Attribute gets/sets a particular variable on the servant
- Example:
  - A remote procedure named the_value with no parameters passed, that returns a Short value
  - A remote procedure named the_value with a Short value passed as a parameter, that does not return any values (has a void value as the 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)

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)

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);
}
```

No memory management involved!

Interface Description Language (IDL)

Example code – inout

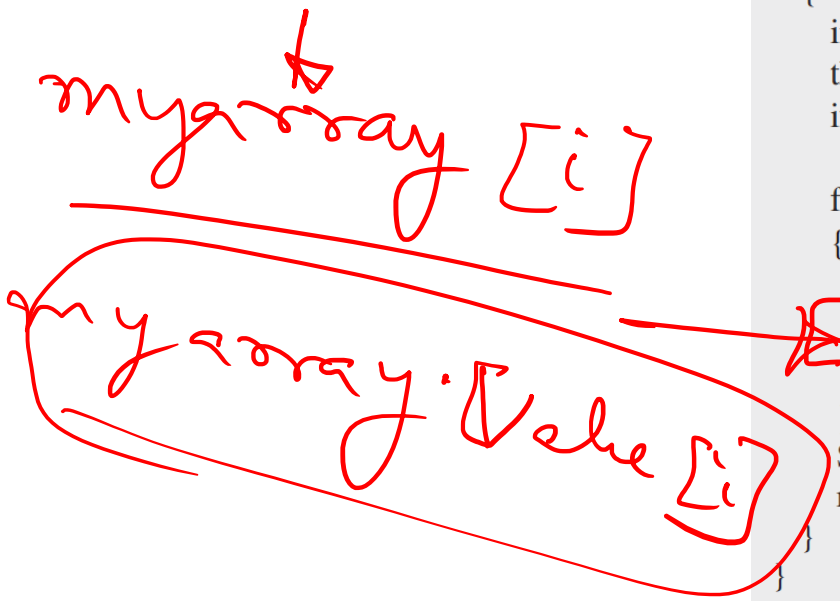
- IDL definition

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

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

- Java code (servant)

```
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;
    }
}
```



Thank You!

In our next session:

CORBA Addressing

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Introduction to Enterprise Application Integration (EAI)

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

Introduction to Enterprise Application Integration

- EAI – Overview
- Message Channels
  - Point to Point messaging
  - Publish-subscribe model

Enterprise Application Integration (EAI)

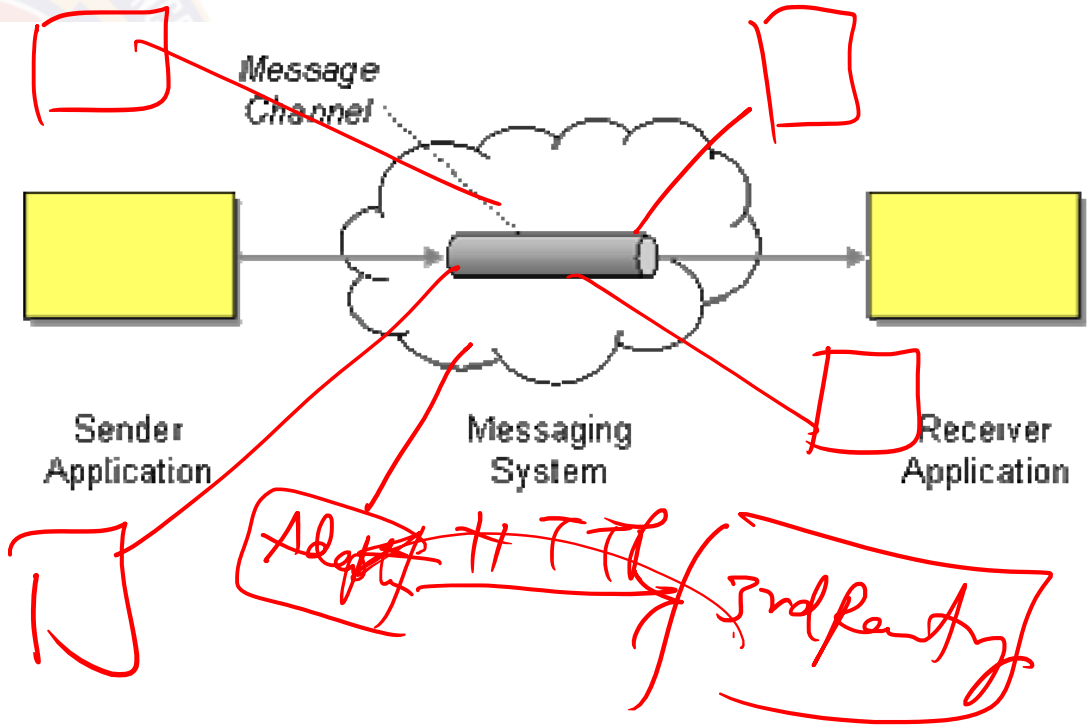
Introduction

- 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

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 – Example (JMS)

- Configuring Message channels (Queues and Topics)

```
j2eeadmin -addJmsDestination jms/mytopic topic
j2eeadmin -addJmsDestination jms/myqueue queue
```

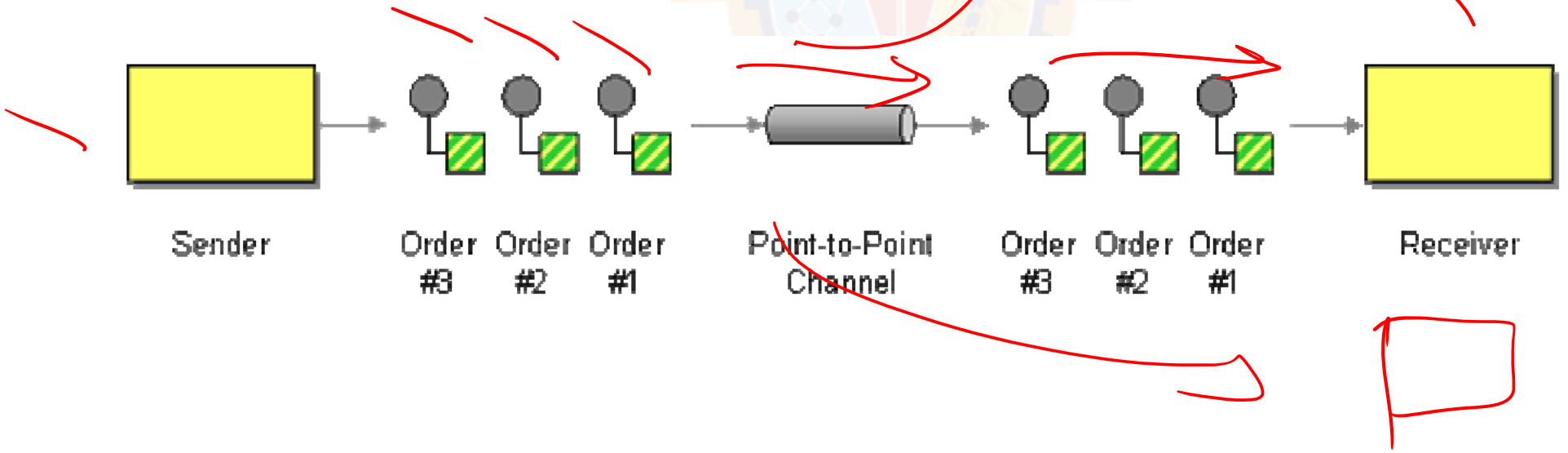
- Accessing Message channels

```
Context jndiContext = new InitialContext();
Queue myQueue = (Queue) jndiContext.lookup("jms/myqueue");
Topic myTopic = (Topic) jndiContext.lookup("jms/mytopic");
```

Enterprise Application Integration (EAI)

Point to point channel

- Point-to-Point Channel ensures that only one receiver consumes any given message, irrespective of number of receivers in that channel
- Increases scalability in multi consumer applications
- JMS Implementation uses 'Queue' interface to implement Point to point channel
  - QueueSender – send messages
  - QueueReceiver – receive messages



Enterprise Application Integration (EAI)

Point to point channel – Example (JMS)

- Sending message

```
Queue queue = // obtain the queue via JNDI
QueueConnectionFactory factory = // obtain the connection factory via JNDI
QueueConnection connection = factory.createQueueConnection();
QueueSession session = connection.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
QueueSender sender = session.createSender(queue);

Message message = session.createTextMessage("The contents of the message.");


sender.send(message);
```

Enterprise Application Integration (EAI)

Point to point channel – Example (JMS)

- Receiving message

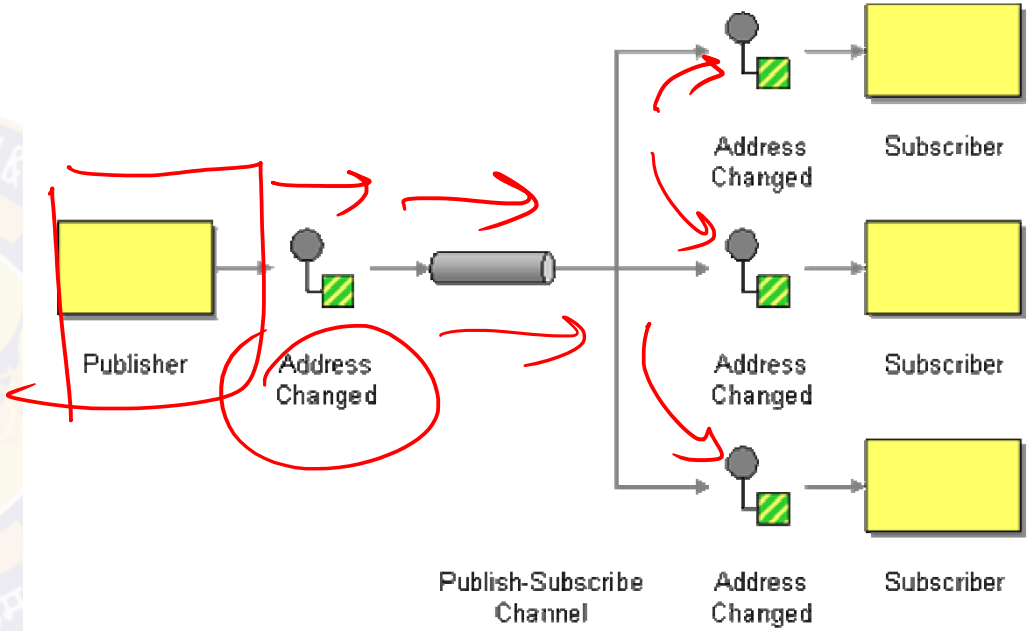
```
Queue queue = // obtain the queue via JNDI
QueueConnectionFactory factory = // obtain the connection factory via JNDI
QueueConnection connection = factory.createQueueConnection();
QueueSession session = connection.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = session.createReceiver(queue);

TextMessage message = (TextMessage) receiver.receive();
String contents = message.getText();
```

Enterprise Application Integration (EAI)

Publish - subscribe channel

- Used for broadcasting messages to each recipient in the channel
- Ensures that each recipient gets only one copy of the message
- Helps in debugging channel without disturbing traffic
- JMS Implementation uses 'Topic' interface to implement Point to point channel
  - TopicPublisher – send messages
  - TopicSubscriber – receive messages



Enterprise Application Integration (EAI)

Publish - subscribe channel – Example (JMS)

- Sending message

```
Topic topic = // obtain the topic via JNDI
TopicConnectionFactory factory = // obtain the connection factory via JNDI
TopicConnection connection = factory.createTopicConnection();
TopicSession session = connection.createTopicSession(true, Session.AUTO_ACKNOWLEDGE);
TopicPublisher publisher = session.createPublisher(topic);

Message message = session.createTextMessage("The contents of the message.");

publisher.publish(message);
```

Enterprise Application Integration (EAI)

Publish - subscribe channel – Example (JMS)

- Receiving message

```
Topic topic = // obtain the topic via JNDI
TopicConnectionFactory factory = // obtain the connection factory via JNDI
TopicConnection connection = factory.createTopicConnection();
TopicSession session = connection.createTopicSession(true, Session.AUTO_ACKNOWLEDGE);
TopicSubscriber subscriber = session.createSubscriber(topic);

TextMessage message = (TextMessage) subscriber.receive();
String contents = message.getText();
```

Thank You!

In our next session:

Middleware Security

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Middleware Security

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

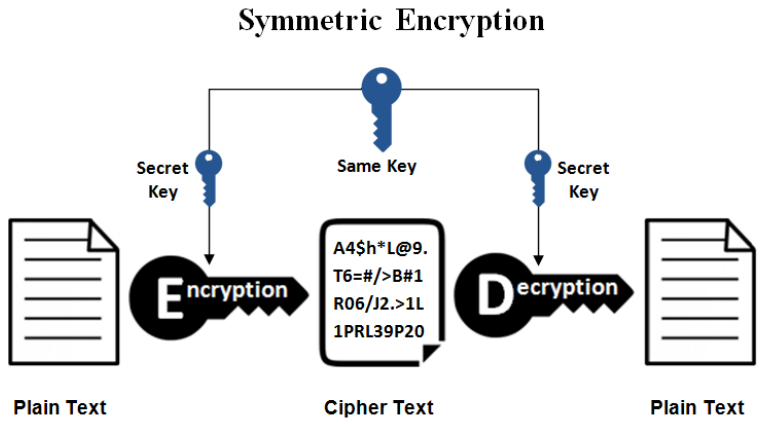
Middleware Security

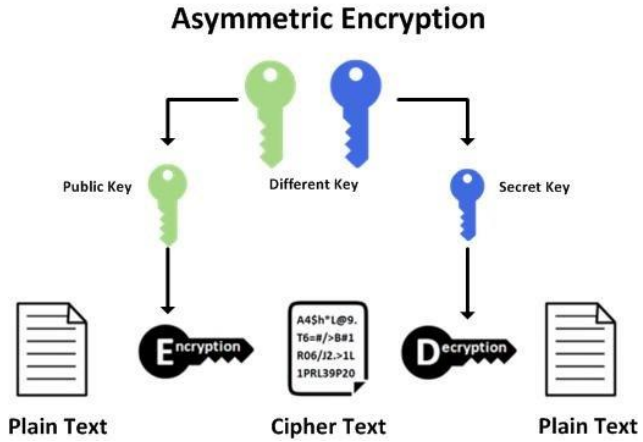
- Symmetric and Asymmetric keys
- Digital Signatures
- Message Authentication Codes
- Secure Socket Layer / Transport Layer Security

Middleware Security

Symmetric Cryptography and Asymmetric/Public Keys

- Symmetric key cryptography - uses the same key to encrypt the text and to decrypt the text
  - Both the parties need to know the key (offline sync of the keys)
- Asymmetric/Public key cryptography - uses two mathematically related keys
  - Public key
  - Private key
  - Message encrypted using either of the keys can be decrypted with the other
  - Often used to establish secure connection between two entities, to negotiate Symmetric key





Middleware Security

Digital Signatures

- Validates the authenticity and integrity of the digital message using asymmetric keys
- Sender side:
  - Digital message is hashed (ex: SHA-1)
  - Sender’s private key is used to encrypt it
  - Message is sent along with digital signature
- Recipient side:
  - Digital signature is decrypted using sender’s public key
  - Message is hashed and compared with decrypted signature
  - If they match, message is accepted as not tampered

Middleware Security

Message Authentication Codes

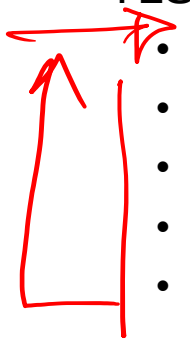
- Function same as digital signature, but by using symmetric keys
- Sender and receiver share same encryption key and use it for encrypting/decrypting the hash
- Caveat – A receiver can use the MAC to pretend to be the original sender to a third party
  - Workaround - Hash based MAC (HMAC) – uses hashing with secret key

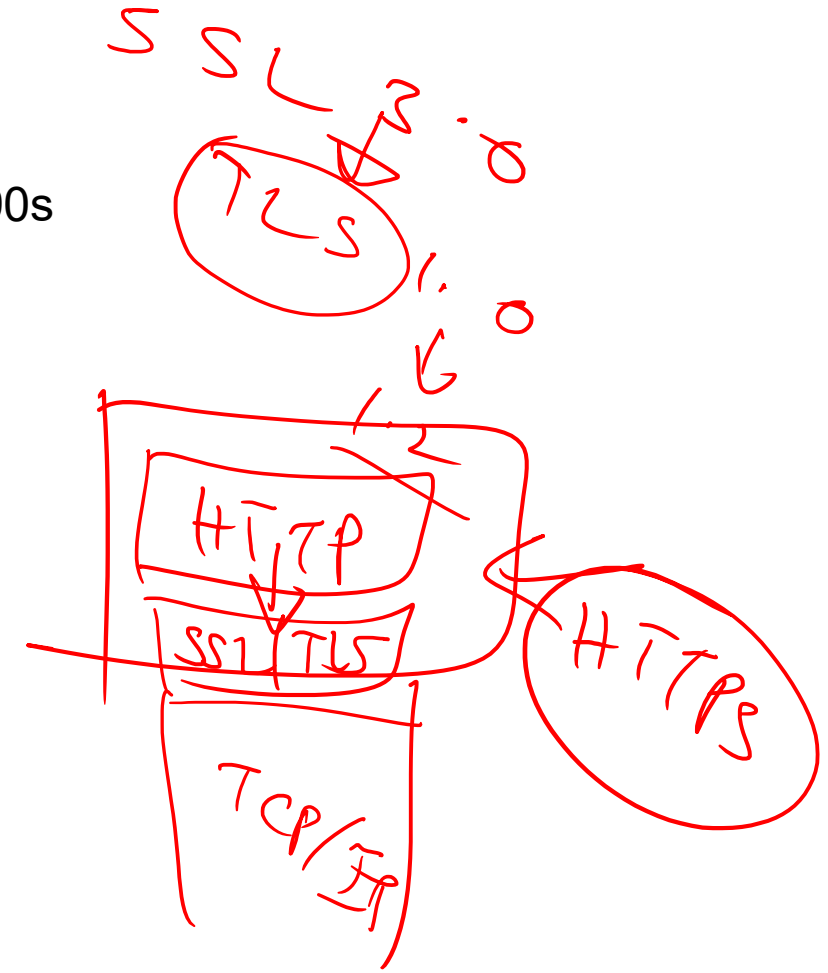
HMAC (key, message) =
hash (
(secret key XOR outer-padding) concatenated with
hash(
(secret key XOR inner padding) concatenated with message
)
)
where
outer padding = 0x5c0x5c…0x5c (block long hex constant)
Inner padding = 0x360x36…0x36 (block long hex constant)

Middleware Security

Secure Socket Layer and Transport Layer Security

- SSL Evolved in 1990s as security mechanism for HTTP (i.e. HTTPS)
- Transport Layer Security (TLS) evolved as follow-on for SSL in late ‘90s
- SSL/TLS run on TCP/IP, above HTTP – making it HTTPS
- SSL 2.0 Major Limitations (RFC 6175)
  - MD5 is no longer considered secure
  - Subject to man-in-the-middle attacks and forced session terminations
  - Same key used for message integrity and encryption
- TLS Sub protocols (RFC 5246)
  - Record protocol – Encrypts message using MAC
  - Handshake protocol – Agrees on the algorithm to use in the session
  - Alert protocol – Error handling
  - Cipher spec protocol – Changes cipher strategies/signals
  - Application data protocol – Provides data transparency to applications

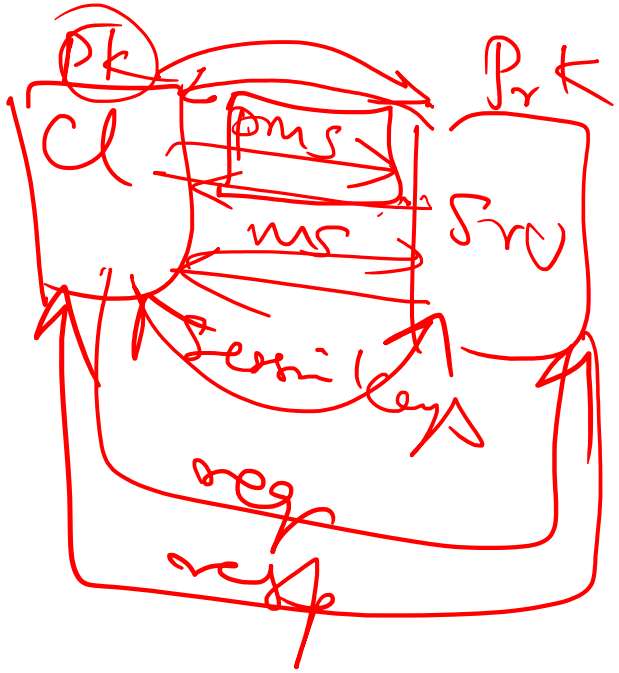




Middleware Security

TLS Handshake Protocol flow

- Hello message from client to server, hello response from server to client:
  - Used to negotiate:
    - TSL or SSL version to be used,
    - Session ID,
    - Cipher Suite
      - Cipher suites are combination of cryptographic algorithms for:
        - Key exchange (Diffie-Hellman, RSA, etc.)
        - Cipher (AES, etc.)
        - MAC (SHA256, etc.)
    - Compression Method
  - As part of the Hello exchange, the server sends a certificate to the client, this includes the server's public key (for example, RSA)
- The Client then does the following:
  - Generates a premaster secret—see Information Security Stack Exchange (2014b):
    - This 48-byte premaster secret is generated by concatenating protocol versions with some randomly generated bytes.
    - The client then encrypts the 48-byte premaster secret with the server's RSA public key (from the certificate).
  - Sends it to the server
- The Server decrypts the premaster secret using its private key.
- Both client and server generate a master secret using the premaster secret, then immediately delete the premaster secret.
- The client and the server use the master secret to generate the session keys—these are symmetric keys used to encrypt and decrypt data transferred during the session (AES, for example).
- The client can now send the server a message that is encrypted with the session key and authenticated with the MAC (for example, HMAC with SHA256).
- The server determines that the MAC was authentic, and similarly sends back an encrypted message with a MAC that the client also determines is authentic.



Thank You!

In our next session:

JMS Server setup and configuration

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Integration Styles

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

Integration Styles

- Introduction - Application Integration Criteria
- Application Integration Options
  - File Transfer
  - Shared Database
  - Remote Procedure Invocation
  - Messaging

Integration Styles

Introduction – Application Integration Criteria

- Enterprise integration is the task of making (possibly heterogenous) applications work together to produce a unified set of functionality.
- Integration Criteria
  - Application integration –need to reuse, interact and integrate other applications
  - Application coupling – Minimum dependency /
  - Integration simplicity – Minimize customization needed to integrate applications
  - Integration technology – Reduce the need for specialized hardware/software for integration
  - Data format – Agreement on data format / translation
  - Data timeliness – Reduce staleness of data due to integration nuances
  - Data or functionality – Ability to share both data and functionality
  - Asynchronicity – Ability to proceed without being blocked for complex batch transactions

Integration Styles

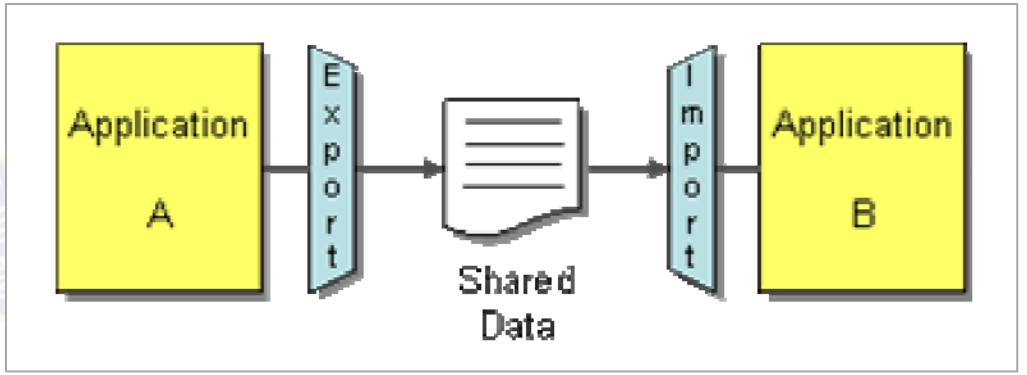
Application Integration Options

- No one fit for all methodology – option chosen depends on context
- Major integration options:
  - File Transfer – applications communicate via files
  - Shared Database – applications share common data using shared database
  - Remote Procedure Invocation – Applications expose procedures to be invoked by integrating applications
  - Messaging – applications connect/communicate via common messaging system

Integration Styles

File Transfer

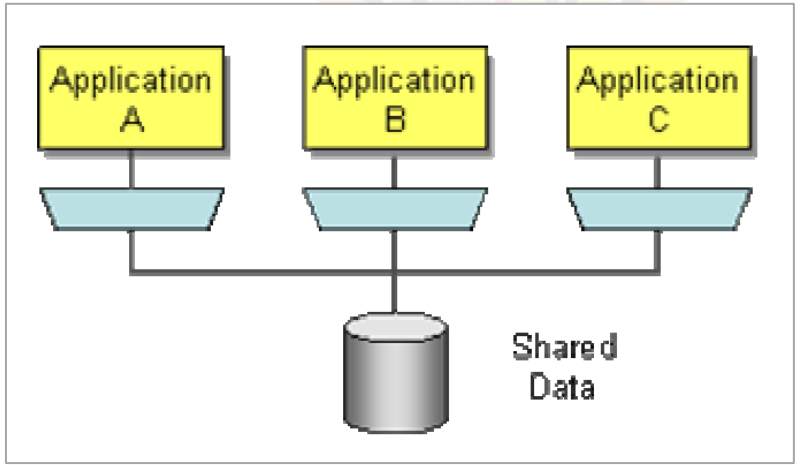
- Each application produces files with data that it needs to share with others
- Integrators (middleware) handle transforming files into various formats and access aspects
  - Legacy file formats (Mainframe)
  - Text format (Unix variants)
  - CSV format (Windows based systems)
  - XML/JSON formats (Web based systems)
- Files are produced/consumed at a frequency demanded by business
  - Hourly, Nightly, Weekly etc.
  - Ex: Batch processing applications like medical insurance claims (nightly)
- Less work on integration side, more work for application developers
- Staleness of data, in the generation of real time processing



Integration Styles

Shared Database

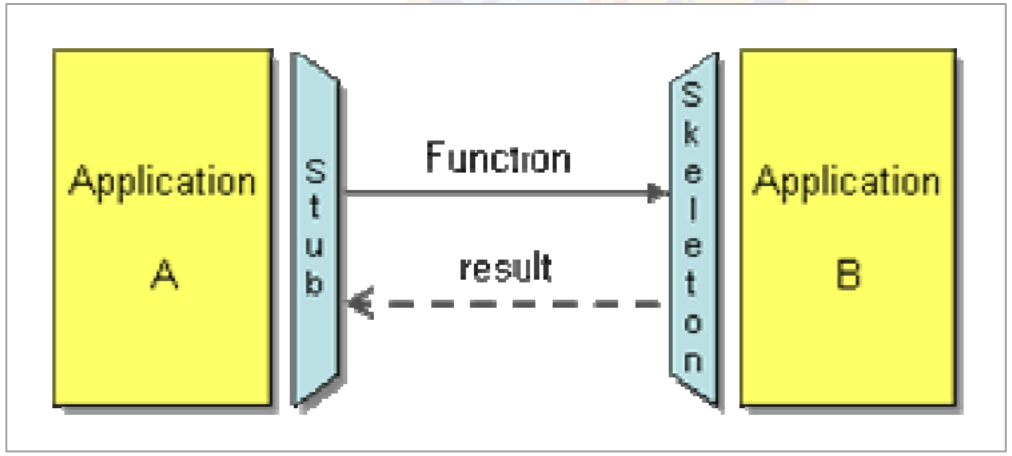
- Integrate applications by having them store their data in a single Shared Database
- Consistency in data format and ease of access
- Challenges
  - Deciding on a unified schema that fits all applications' needs
  - External packages that do not conform to enterprise schema
  - Performance bottlenecks on database
    - Distributed database locking / deadlock issues



Integration Styles

Remote Procedure Invocation

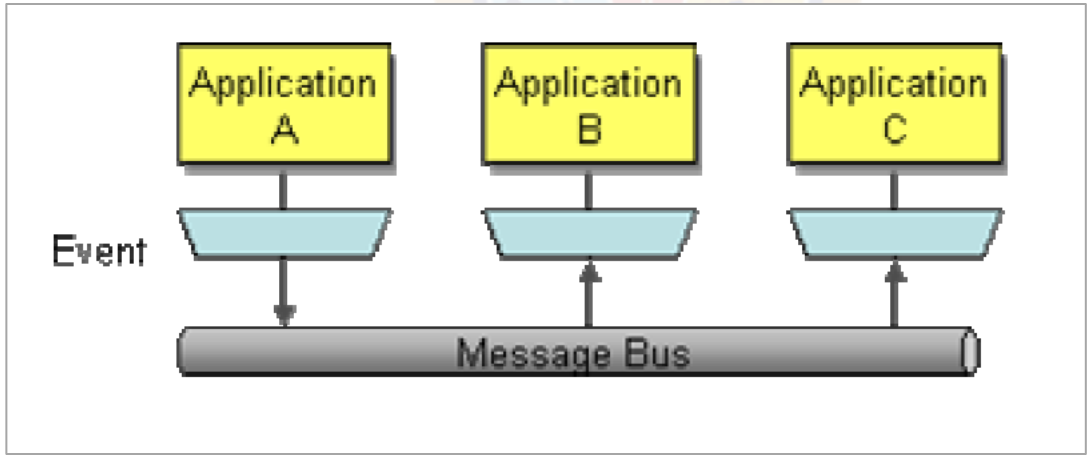
- Applies the principle of encapsulation to integrating applications
- Applications provide interface for the methods/data they want to expose to other applications to integrate with
- Each application maintains its own encapsulation/integrity of data
- Promotes certain degree of (tight) coupling between applications
- Examples: CORBA, COM, .NET Remoting etc.



Integration Styles

Messaging

- Transfers packets of data frequently, immediately, reliably, and asynchronously, using customizable formats
- Supports asynchronous communication seamlessly
- Allows loose coupling of applications
- Supports behavioral collaboration of applications with event based communication



Thank You!

In our next session:

Messaging Systems

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Messaging Systems

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

Messaging Systems

- Basic Concepts
- Message Channel
- Message
- Pipes and Filters
- Message Router
- Message Translator
- Message Endpoint

Messaging Systems

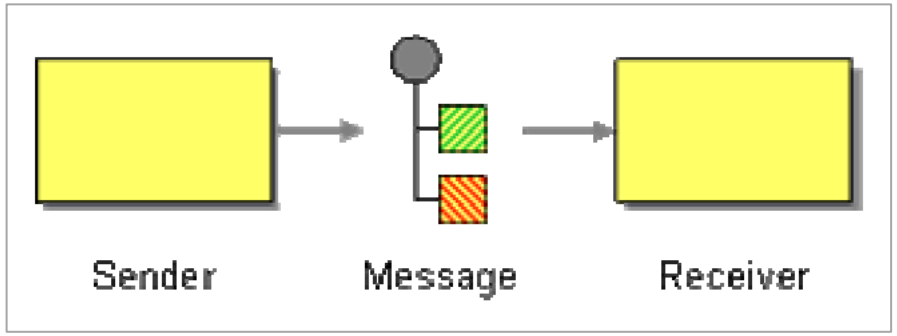
Basic Concepts

- Channel – A virtual pipe that connects sender and receiver
- Message – Atomic packet of data transmitted through a channel
- Multi-step Delivery – Architecture that's used to send data across the remote network
- Routing - Process of navigating the channel topology to get the message from sender to receiver
- Transformation – Process of adapting data to a format accepted over the network in question
- Endpoint – A logical port on the channel/system where the message is directed to, to reach with the destination/receiver application

Messaging Systems

Message

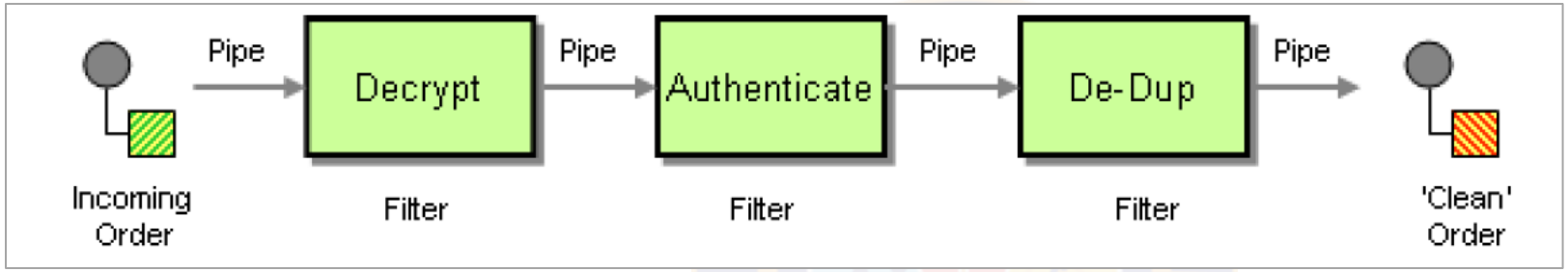
- Unit of data that's transmitted between sender/receiver
- Consists of:
  - Header - Information used by the messaging system that describes the data being transmitted
  - Body - The actual data being transmitted
- Message sequence – Used if the data can not be fit into a single message
- Types of messages – Command, Data, Event etc.
- Formats: JMS Message, SOAP Message, .NET Message etc.
- JMS Message – Represented by type "Message"
  - TextMessage
  - ByteMessage
  - ObjectMessage
  - StreamMessage
  - MapMessage



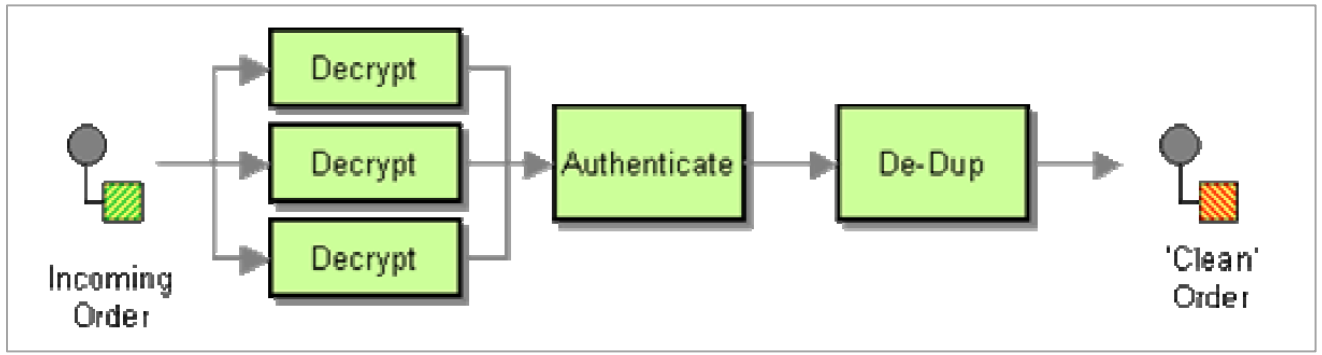
Messaging Systems

Pipes and Filters

- Used to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes)



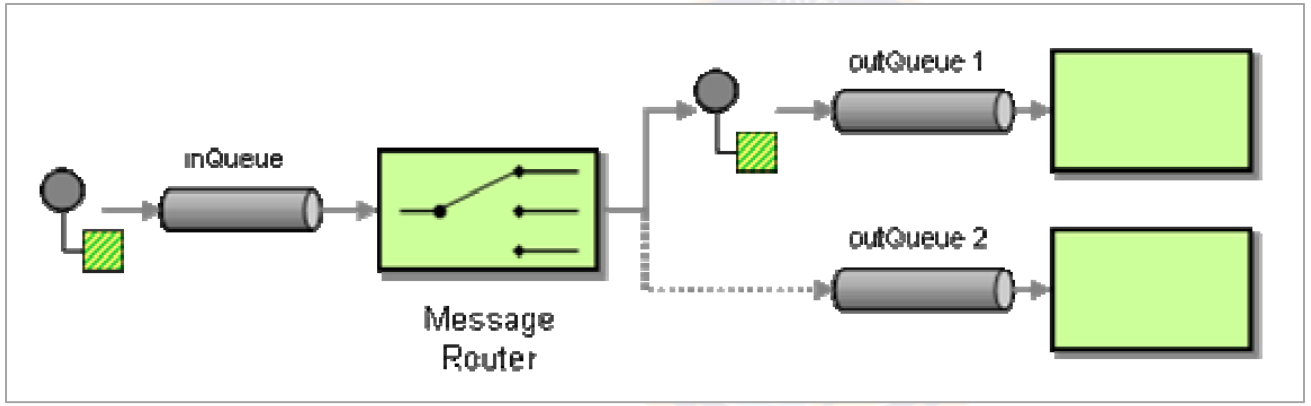
- Pipe uses Message Channel or in-memory queue as implementation
- Allows pipeline processing to cater to volume and speed
- Parallel processing of messages at a particular layer can be performed using multiple parallel filters at the bottleneck



Messaging Systems

Message Router

- Message Router is a special filter which consumes a Message from one Channel and republishes it to a different channel depending on a set of conditions

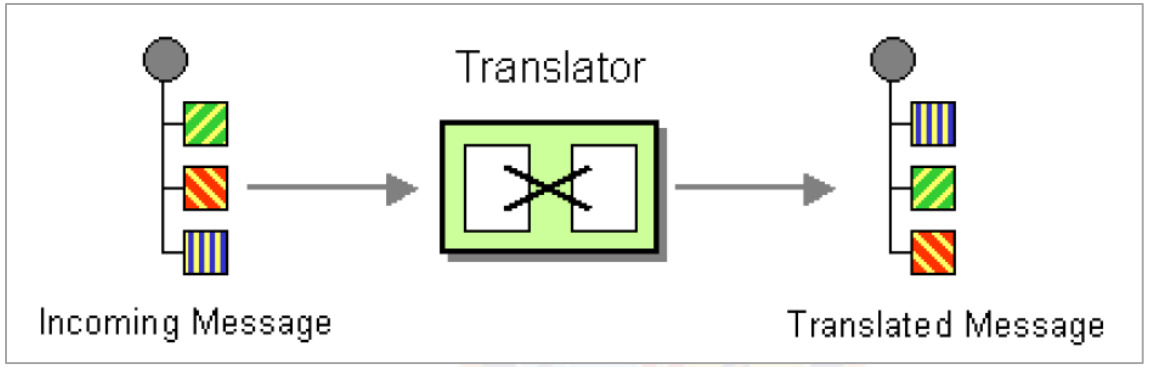


- Represents single place to maintain decision logic for message routing
- Can act as a load balancer as well
- Types
  - Content based Router
  - Context based Router

Messaging Systems

Message Translator

- Message Translator, is a special filter between other filters or applications to translate one data format into another

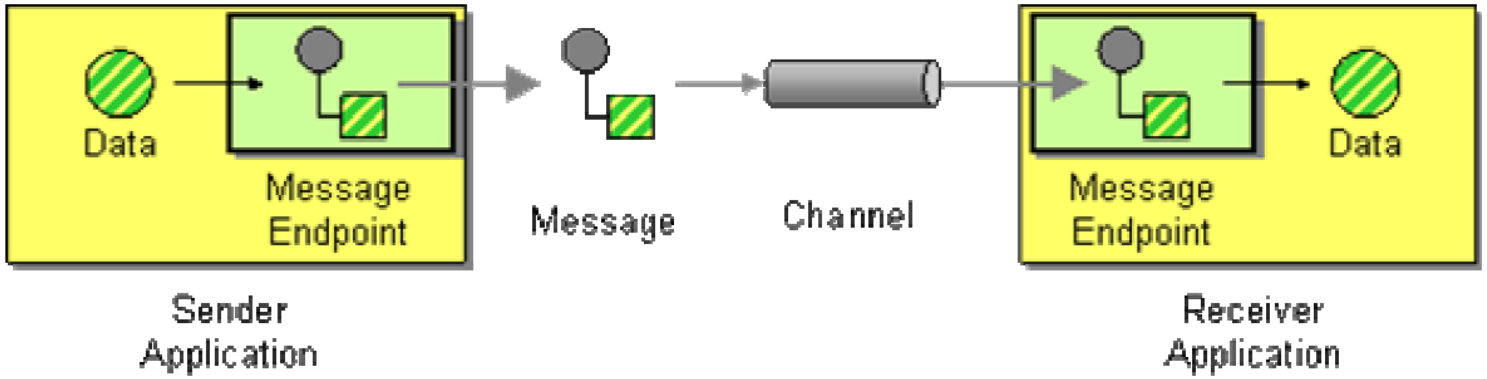


- Equivalent to “adapter” pattern of GoF patterns
- Decouples from data format dependency
- Message Translation Levels:
  - Data structures
  - Data types
  - Data representation
  - Transport

Messaging Systems

Message Endpoint

- Message Endpoint is a client of the messaging system that can be used to connect application to the message channel, to send messages



- Message Endpoint code is custom to both the application and the messaging system's client API
- Encapsulates the messaging system from the rest of the application, and customizes a general messaging API for a specific application and task
- One instance of Endpoint can be used to either send or receive messages, not both
- JMS Implementation – MessageProducer, MessageConsumer

Thank You!

In our next session:

Message Channels

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Message Construction

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

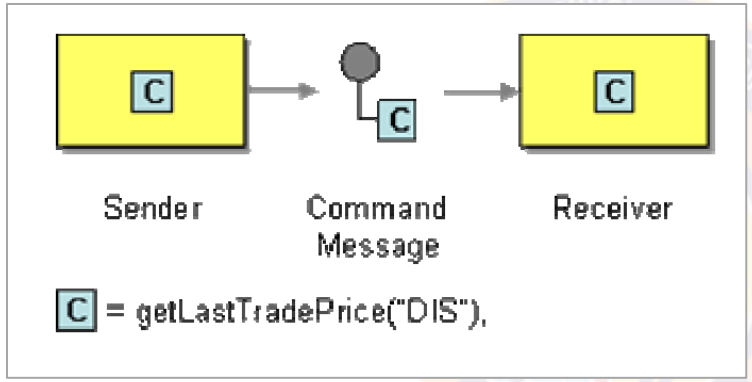
Message Construction

- Command Message
- Document Message
- Event Message
- Request-Reply
- Return Address
- Correlation Identifier
- Message Sequence
- Message Expiration
- Format Indicator

Message Construction

Command Message

- Command Message is used to invoke a procedure in another application, reliably.
- Asynchronous invocation of procedures, as opposed to RPC

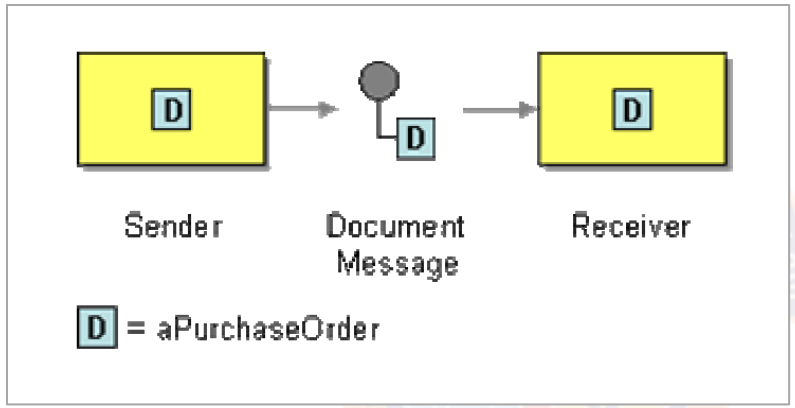


- Message contains command for the other application, there's no other specific indication on message
- Usually sent on point-to-Point channel
- Ex: SOAP messages

Message Construction

Document Message

- Document Message is used to reliably transfer a data structure between applications

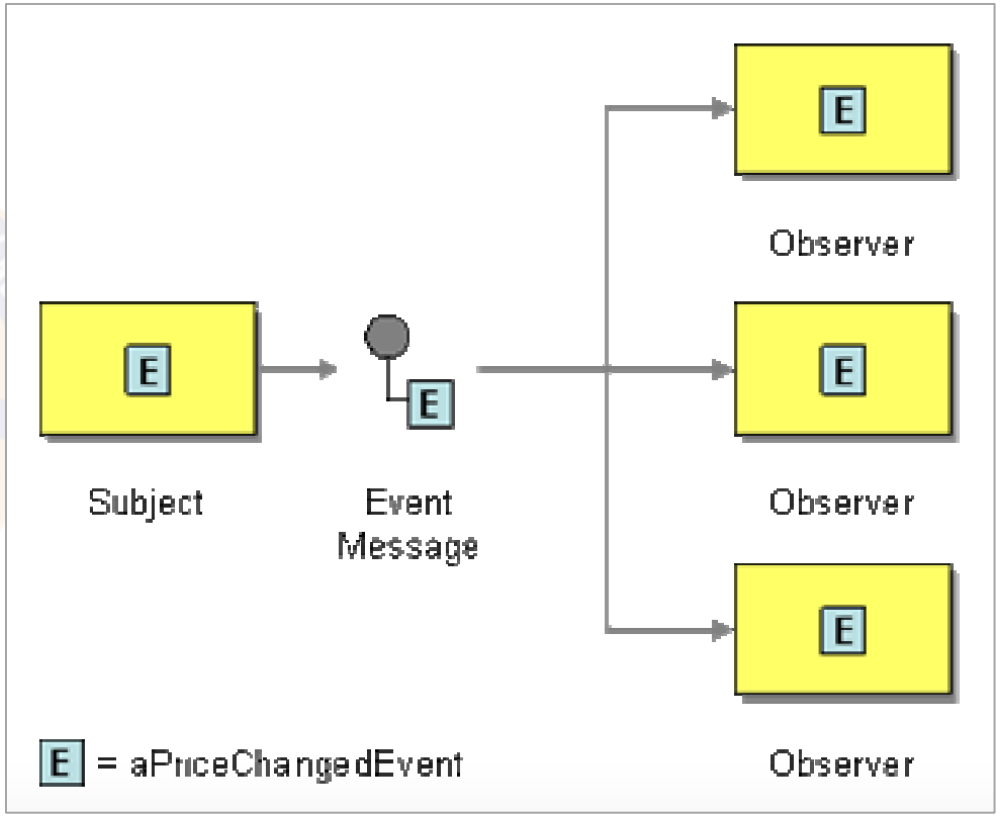


- Can contain any kind of message in the system, except that it represents a data object
- Usually sent on a point-to-point channel
- Ex: XML documents sent across the network

Message Construction

Event Message

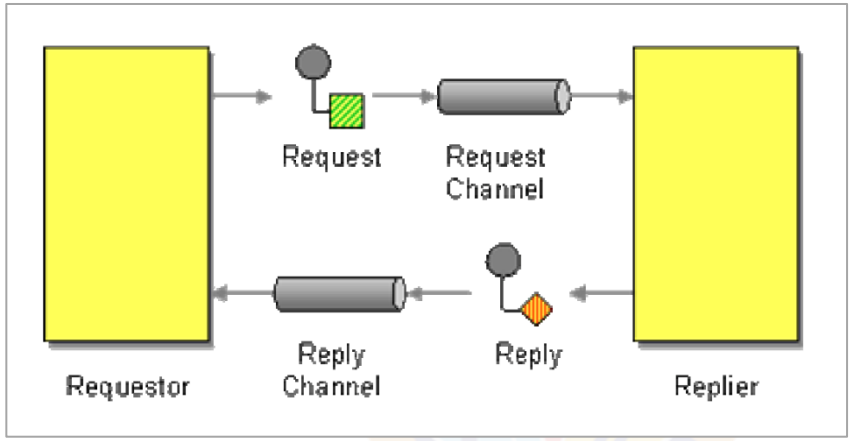
- Document Message is used to reliably send event notifications between application
- Sender will create an event object and sends it in a message (no special indicator used)
- Push model – sends event and the related state change info
- Pull model – sends only the event details and optionally additional info to obtain state details
  - Update
  - State Request
  - State Reply



Message Construction

Request-Reply

- Facilitates two way communication between sender/receiver using two-way channel

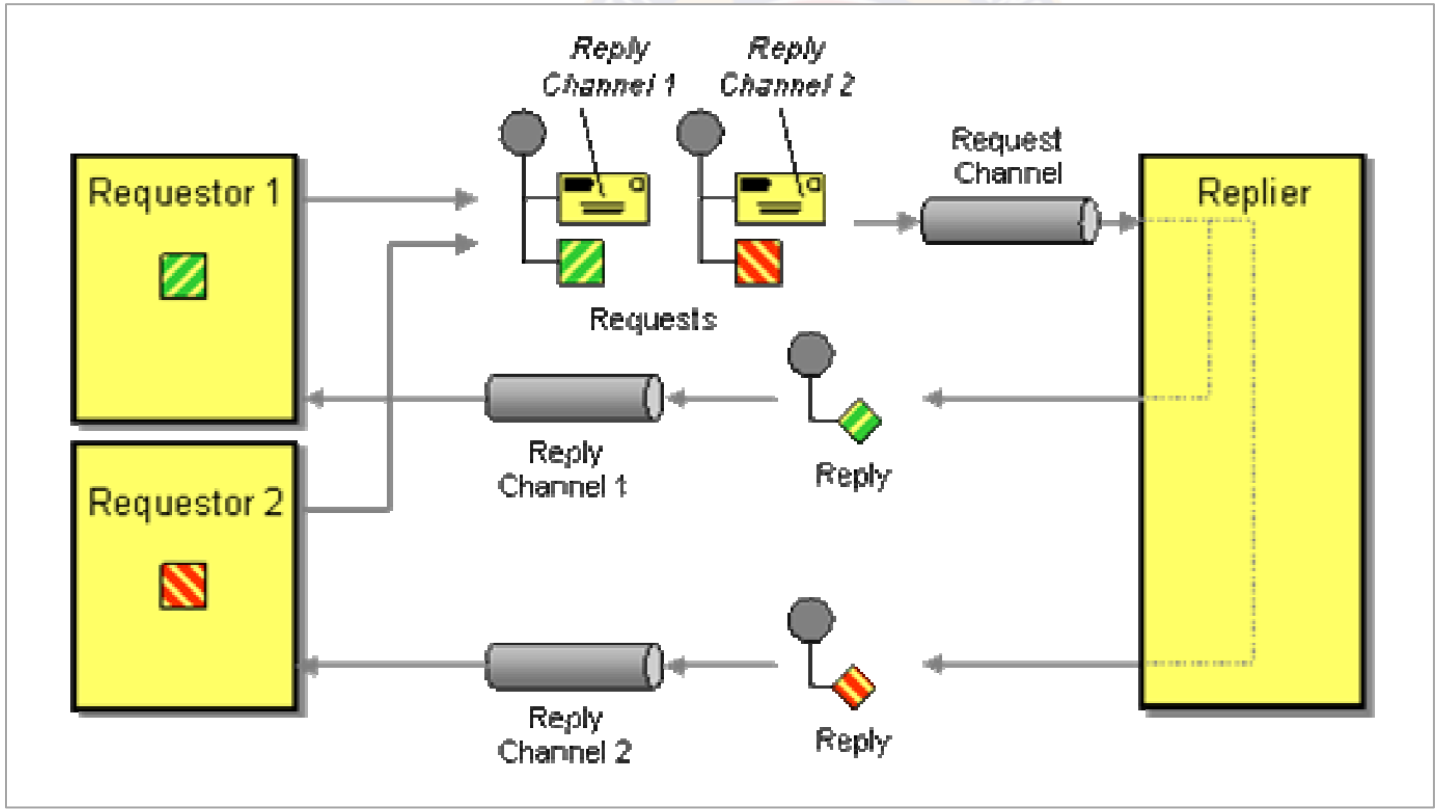


- Requestor – sends request message and waits for reply
- Replier – receives request message and sends the reply
- Request channel could be point-to-point or publish-subscribe
- Reply channel is almost always point-to-point
- Reply can be:
  - Synchronous (blocking)
  - Asynchronous (callback)

Message Construction

Return Address

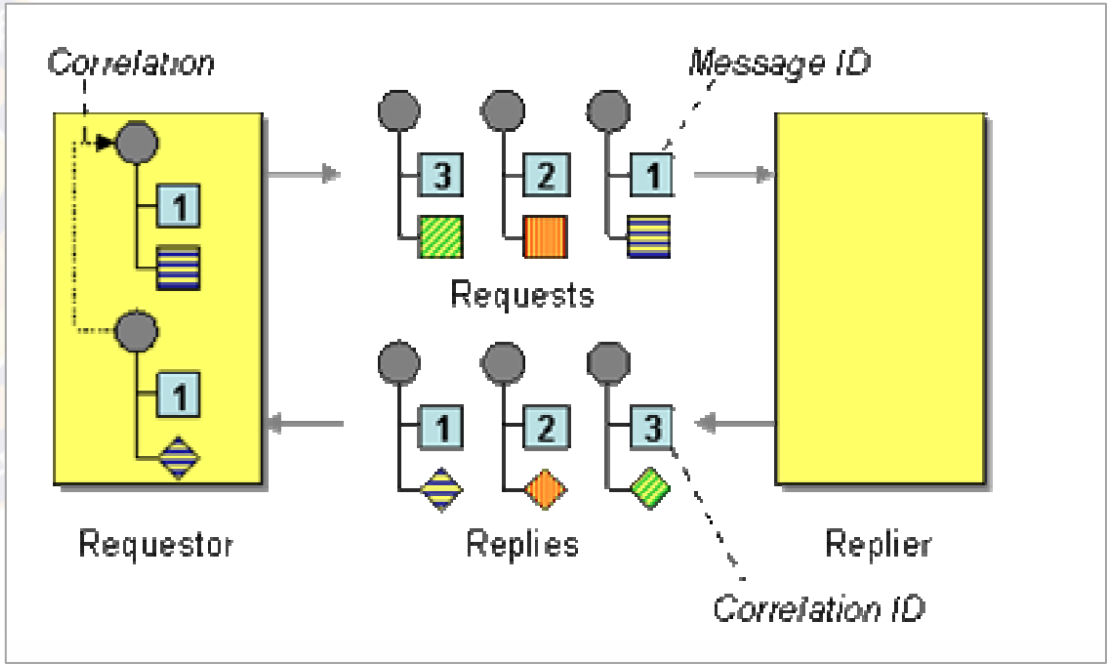
- Asynchronous request-reply model has callback mechanism, which needs the receiver to know where to send the response, especially with multiple senders in question
- Return Address is supplied along with the Request Message, that will be used by receiver to send the reply back (Ex: JMS – JMSReplyTo field on request object)



Message Construction

Correlation Identifier

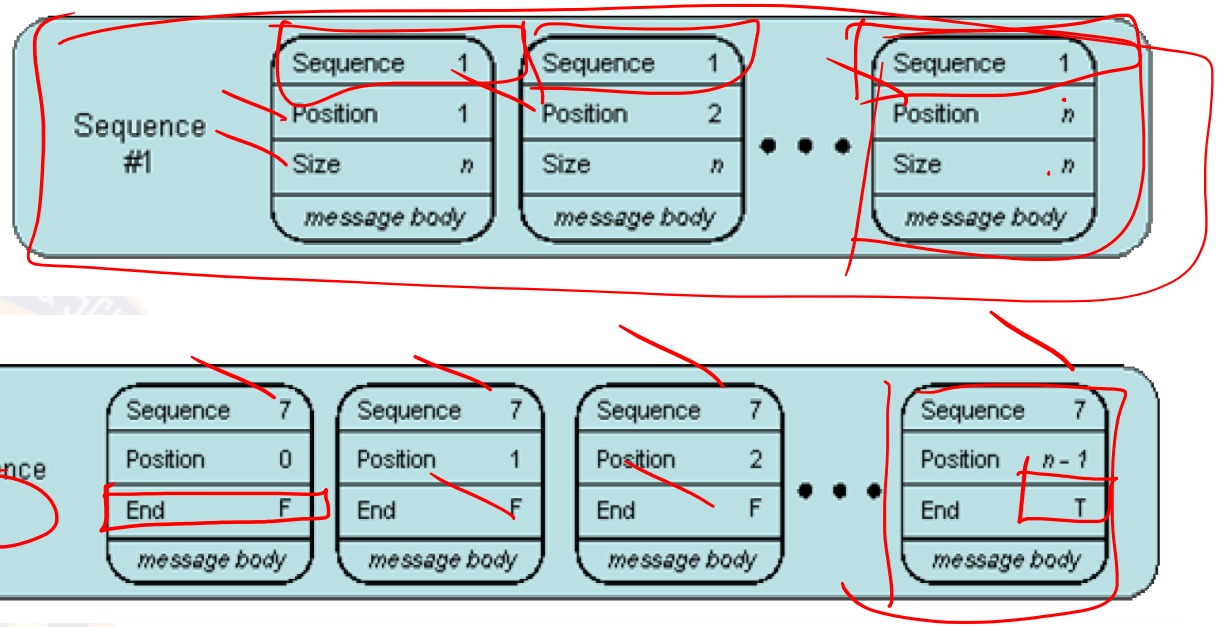
- Each reply in request-reply model should contain a Correlation Identifier that uniquely indicates a which request this reply is for.
- Correlation Identifier has 6 parties to it:
  - Requestor – Sender App
  - Replier – Replier App
  - Request – Contains Request ID
  - Reply – Contains Correlation ID
  - **Request ID – A unique token from the request**
  - **Correlation ID – Unique token in response copied from Request ID**
- Correlation ID (and even Request ID) are specified in respective message headers
- Ex: JMS - JMSCorrelationID and JMSMessageID



Message Construction

Message Sequence

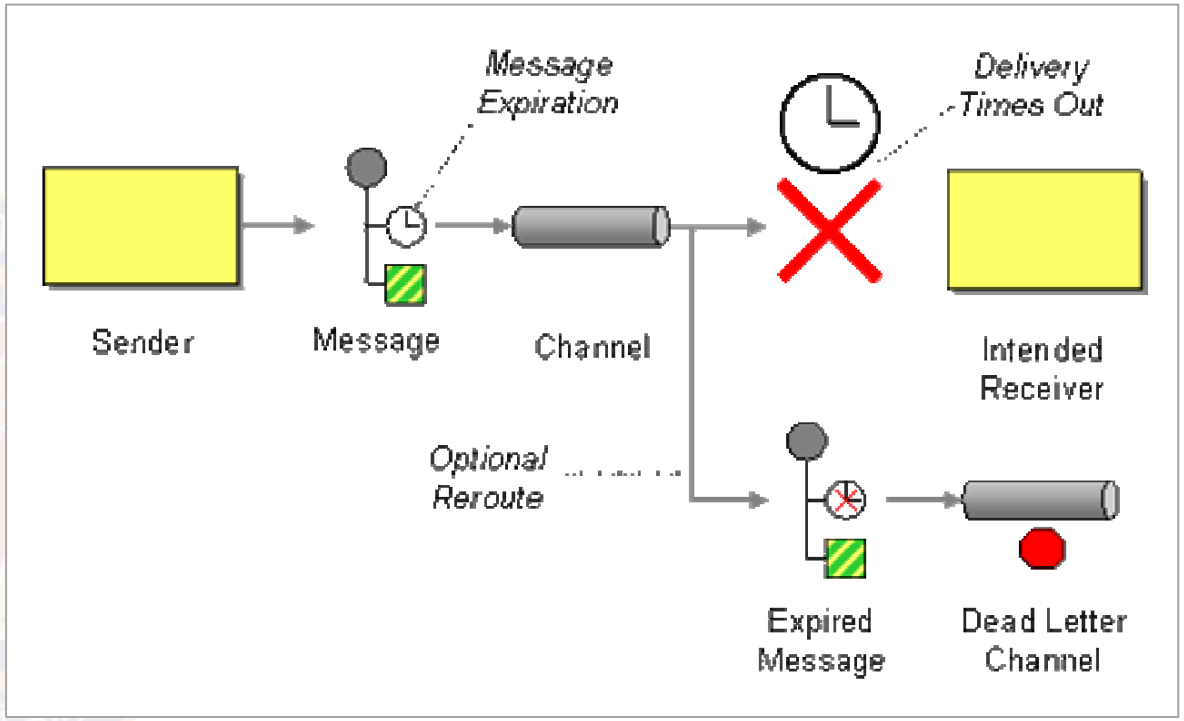
- Message Sequence is used whenever a large set of data may need to be broken into message-size chunks
- Each message in Message Sequence is marked with sequence identification fields
- Message Sequence identification fields
  - Sequence identifier
  - Position identifier
  - Size or End indicator
- Example Use cases:
  - Large document transfer
  - Multi item query
  - Distributed query



Message Construction

Message Expiration

- Message Expiration sets the time limit after which the message becomes stale
- One the time limit expires, the message is not considered by consumers for processing, moved to Invalid Message Channel
- It could be configured to be rerouted to Dead Letter Channel for further handing
- Message Expiration is a timestamp
  - Relative (longevity), using sent time property
  - Absolute
- JMS implementation - MessageProducer.setTimeToLive(long)



Message Construction

Format Indicator

- The format indicator enables the sender to tell the receiver the format of the message
- This handles versioning and future changes
- Format indicator can be implemented as:
  - Version Number
  - Foreign Key
  - Format Document

```
<?xml version="1.0"?>
```

```
<!DOCTYPE greeting SYSTEM "hello.dtd">
```

```
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
```

Thank You!

In our next session:

Message Routing

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Message Routing

Srikanth Gunturu

Guest Faculty
BITS, WILP

In this segment

Message Routing

- Simple Routers
- Composed Routers
- Architectural Patterns – Right option to choose

Message Routing

Simple Routers

- Simple Routers are variants of Message Router and route messages from one inbound channel to one or more outbound channels
- Flavors include:
  - Content based router
  - Message Filter
  - Recipient List
  - Splitter
  - Aggregator
  - Resequencer

Message Routing

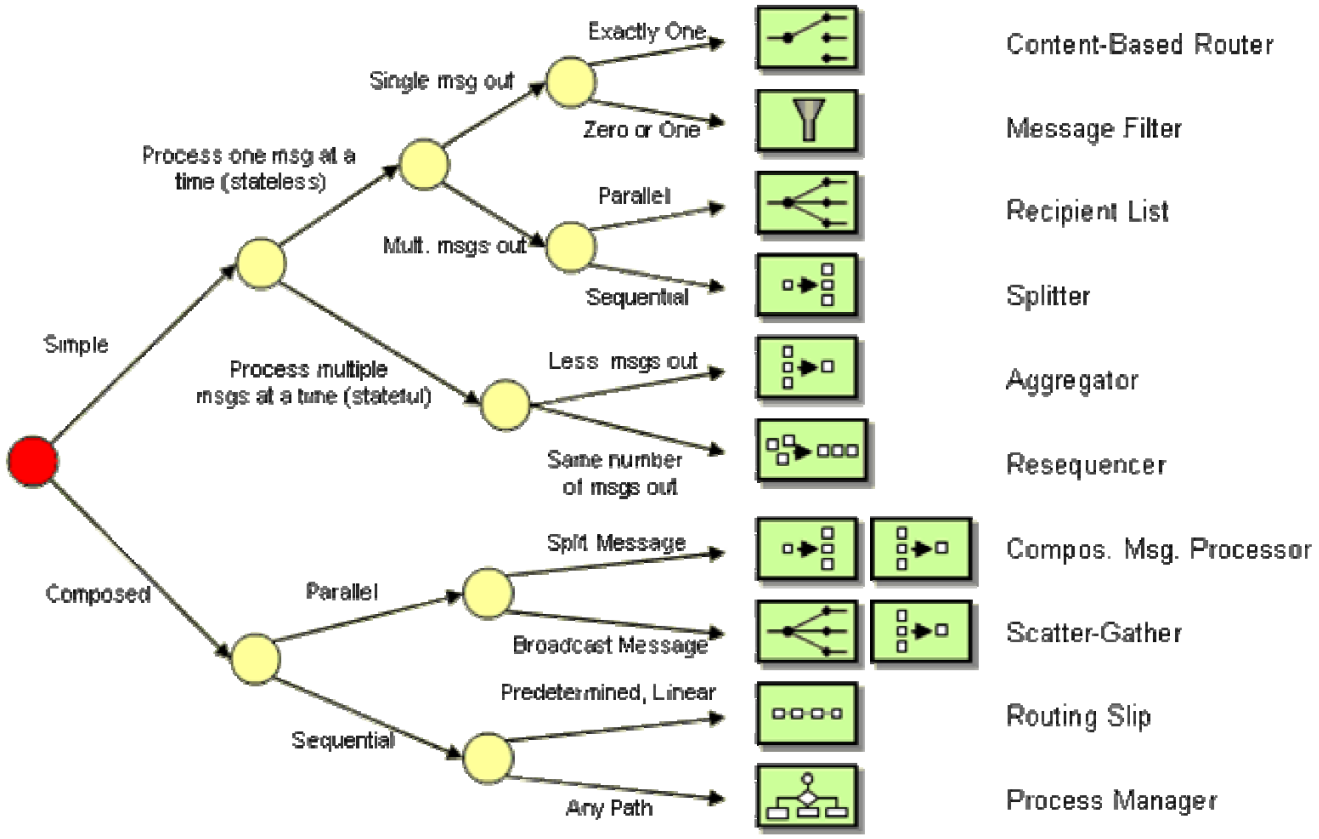
Composed Routers

- Composed Routers combine multiple simple routers to create more complex message flows
- Flavors include:
  - Composed Message Processor
  - Scatter-Gather
  - Routing Slip
  - Process Manager

Message Routing

Architectural Styles – Right Option to Choose

- Follows a hub-and-spoke architectural style, using central Message Broker
- How to choose the right variant



Thank You!

In our next session:

Message Transformation