Lecture 5

Asynchronous Communication

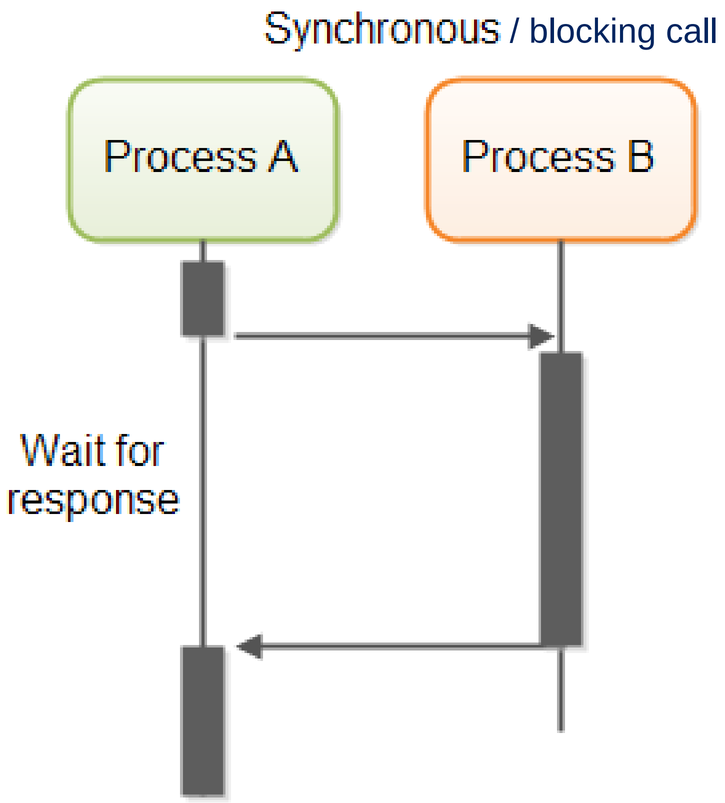
This Week

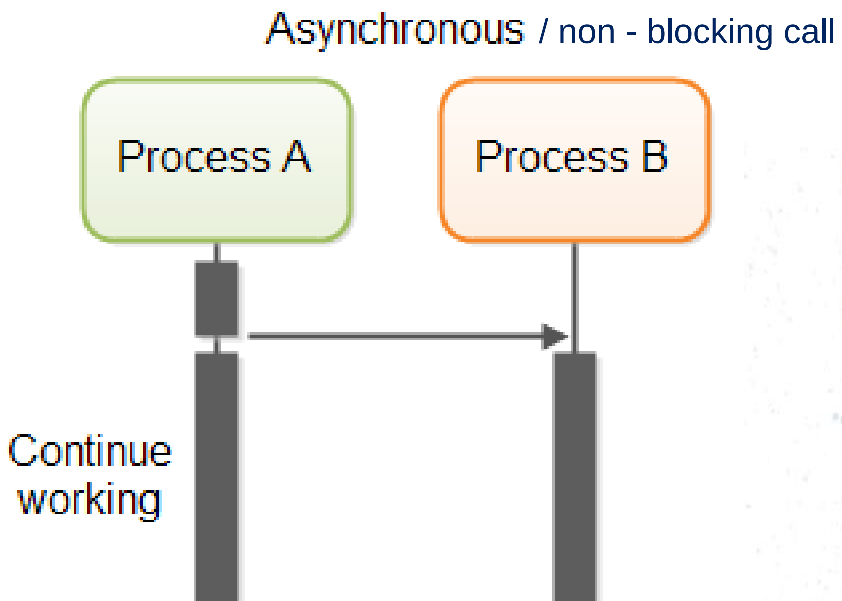
- The interactivity of an application that invokes slow methods at the remote end.
- We will see how this issue can be addressed using asynchronous calls.
- We will look at two main approaches of making asynchronous calls

  - Remote Callback functions

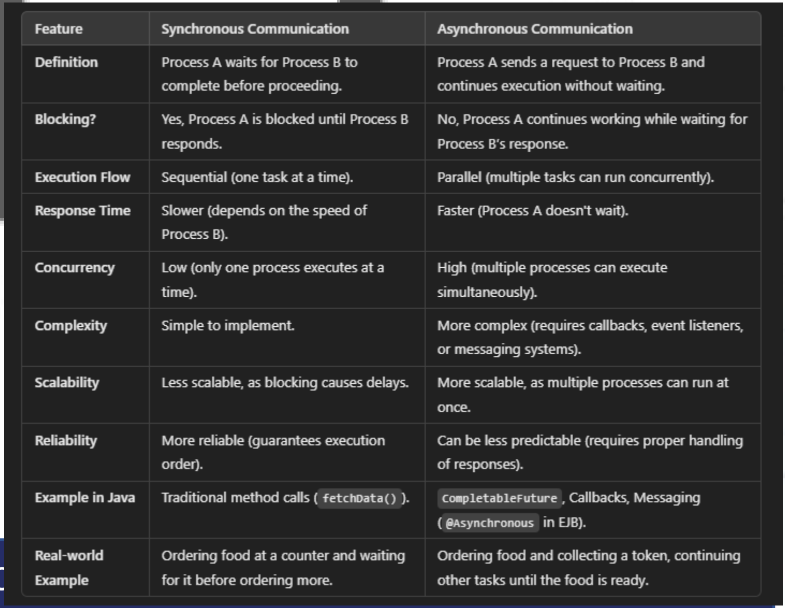
  - Asynchronous Messaging

When an application calls a slow remote method, it can impact interactivity and responsiveness. Instead of blocking the application while waiting for the response, we use asynchronous calls to continue execution without delay.





generally good because,
client dose not have wait



Feature	Synchronous Communication	Asynchronous Communication
Definition	Process A waits for Process B to complete before proceeding.	Process A sends a request to Process B and continues execution without waiting.
Blocking?	Yes, Process A is blocked until Process B responds.	No, Process A continues working while waiting for Process B's response.
Execution Flow	Sequential (one task at a time).	Parallel (multiple tasks can run concurrently).
Response Time	Slower (depends on the speed of Process B).	Faster (Process A doesn't wait).
Concurrency	Low (only one process executes at a time).	High (multiple processes can execute simultaneously).
Complexity	Simple to implement.	More complex (requires callbacks, event listeners, or messaging systems).
Scalability	Less scalable, as blocking causes delays.	More scalable, as multiple processes can run at once.
Reliability	More reliable (guarantees execution order).	Can be less predictable (requires proper handling of responses).
Example in Java	Traditional method calls (`fetchData()`).	`CompletableFuture`, Callbacks, Messaging (`@Asynchronous` in EJB).
Real-world Example	Ordering food at a counter and waiting for it before ordering more.	Ordering food and collecting a token, continuing other tasks until the food is ready.

Blocking Calls and Distributed Computing

- When a function is called, the caller typically must wait (block) until the called function completes & returns
- In a distributed computing system, blocking for a remote call can easily be a waste of resources

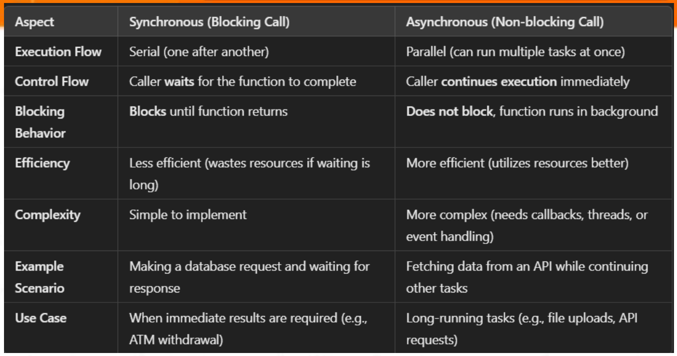
In a distributed system, blocking calls are inefficient because:

  - Especially if it is a call that could take a while

Remote calls take time (network delays , processing time).

  - i.e. Client waits while server performs a long job
  - Not utilising resources properly/efficiently there!

Synchronous vs. Asynchronous



Aspect	Synchronous (Blocking Call)	Asynchronous (Non-blocking Call)
Execution Flow	Serial (one after another)	Parallel (can run multiple tasks at once)
Control Flow	Caller **waits** for the function to complete	Caller **continues execution** immediately
Blocking Behavior	**Blocks** until function returns	**Does not block**, function runs in background
Efficiency	Less efficient (wastes resources if waiting is long)	More efficient (utilizes resources better)
Complexity	Simple to implement	More complex (needs callbacks, threads, or event handling)
Example Scenario	Making a database request and waiting for response	Fetching data from an API while continuing other tasks
Use Case	When immediate results are required (e.g., ATM withdrawal)	Long-running tasks (e.g., file uploads, API requests)

- Synchronous invocation = blocking call
  - Serial processing
  - Control is passed to called function
  - Caller cannot continue until called function returns
- Asynchronous invocation = non-blocking call
  - Parallel processing (or at least one way to implement it)
  - Control is returned immediately to the caller
  - Called function carries on in the background
  - At some later time, caller retrieves function’s return value

Local asynchronous Calls

- Reasons for using asynchronous calls
  - Maintain GUI responsiveness
  - Utilise resources of caller more efficiently (e.g. continue doing other work during a long-running call)
  - Java Swing event dispatching

Distributed Asynchronous calls

- In the client server model, the server is passive: the IPC is initiated by the client;

  In a client-server model, the client typically initiates a request, but in some scenarios, the server needs to communicate proactively.

- Some applications require the server to initiate communication upon certain events.
  - monitoring

    server notifies the client about system events.
  - games

    Multiplayer updates need to be sent without waiting for client requests.
  - auctioning

    Clients need instant notifications about bid changes.
  - voting/polling

    Live updates about voting results.
  - chat-room

    Messages are pushed to clients in real-time.
  - message/bulletin board
  - groupware

    Collaboration tools require updates to be shared in real-time.

When to use Asynchronous Calls?

RPC (Remote Procedure Call)

- *Every* RPC call is potentially long-running
  - Network/server failures are only detected after timeouts expire
  - Making every RPC call asynchronous increases code complexity, just on the *chance* a network failure occurs
- So use asynchronous calls only on functions that are *expected* to take a long time
  - Heavy processing tasks, intensive disk I/O tasks, etc.
  - For GUI clients, responsiveness is also a key issue

Remote asynchronous communication

- Remote Callback functions

way for a client to call a function on a remote server and have the server call back to the client once the operation is complete.

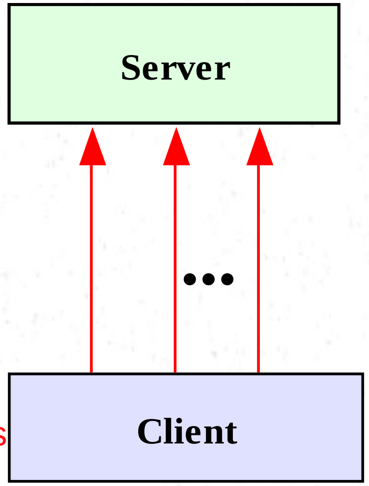
- Messaging (e.g. JMS, Microsoft Messaging Queuing)
- Both Java and .NET supports callback functions

systems exchange messages (requests, notifications, data) without requiring direct, synchronous communication.

Polling vs. Callback

In the absence of callback, a client will have to poll a passive server repeatedly if it needs to be notified that an event has occurred at the server end.

Polling

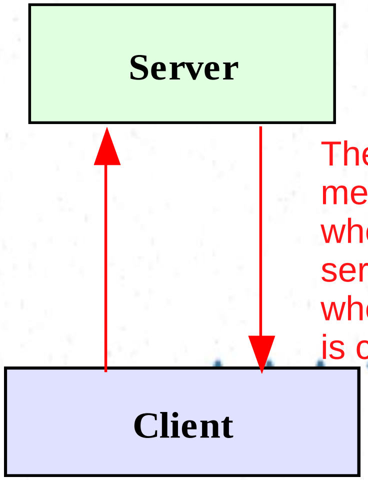


The client repeatedly asks the server if something has changed or if the desired event has occurred.

ex - client want stoke market change

A client issues a request to the server repeatedly until the desired response is obtained.

Callback



The client tells the server, "Call me back when you're done or when something happens." The server will call back to the client when the event occurs or the task is complete.

A client registers itself with the server, and wait until the server calls back.

→ **a remote method call**

Polling vs. Callback

- Blocking is like making a call and waiting for the other party to respond (if the other party is busy with some other call)

  a process or thread is stopped (or "blocked") until the requested operation is complete. This means the program halts and waits for the response or result before continuing further.

- Polling is like repeatedly making a telephone call and check whether the other party is available.

  client repeatedly checks with the server (or some resource) at regular intervals to see if a condition is met or if data is ready

- Callback is like making a call and leaving a message to other party to call back with certain information

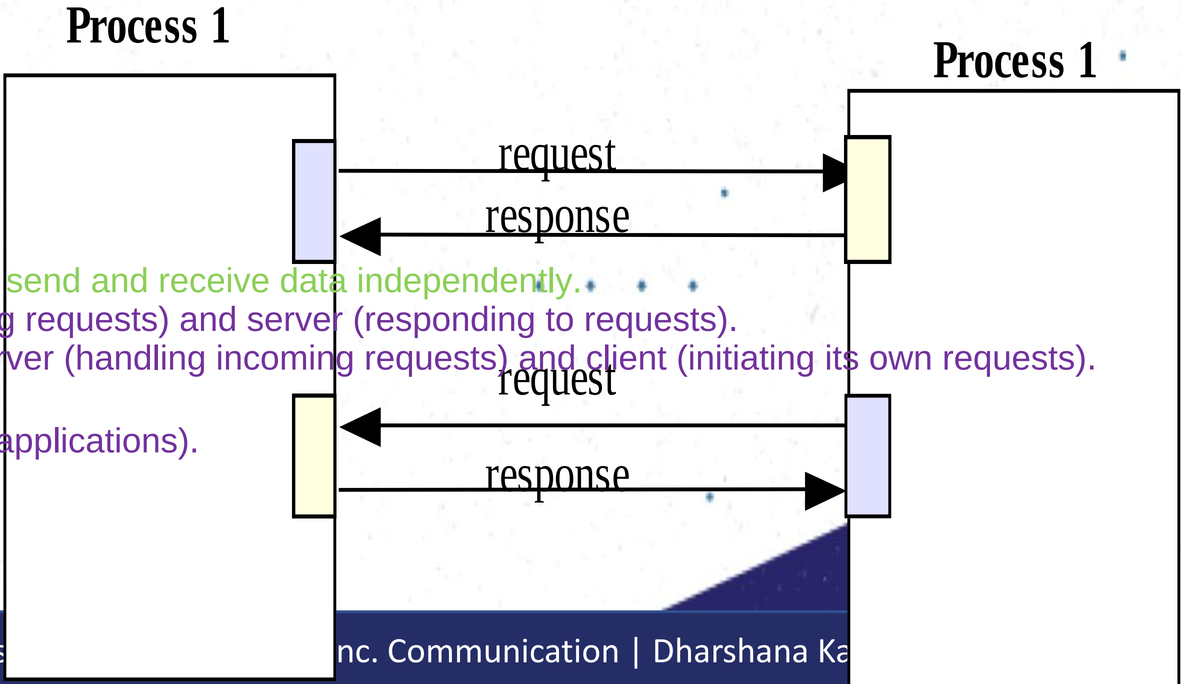
  - the client tells the server or system, "Let me know when the task is done" or "Call me back with the result."

performance and efficiency, callbacks are generally the best choice for long-running or asynchronous tasks.

Two-way communications

Both sides (client and server) can initiate communication at any time, which is known as duplex communication.

- Some applications require that both sides may initiate IPC.
- Using sockets, duplex communication can be achieved by using two sockets on either side.
- With connection-oriented sockets, each side acts as both a client and a server.



Using tools like sockets,
a system can be set up where both sides can send and receive data independently.
Client-Side: Can act as both the client (making requests) and server (responding to requests).
Server-Side: Similarly, can act as both the server (handling incoming requests) and client (initiating its own requests).

real-time communication systems (e.g., chat applications).

RMI Callbacks

RMI (Remote Method Invocation) allows Java objects to communicate over a network by calling methods on remote objects. RMI Callbacks enable a server to notify multiple clients when a specific event occurs.

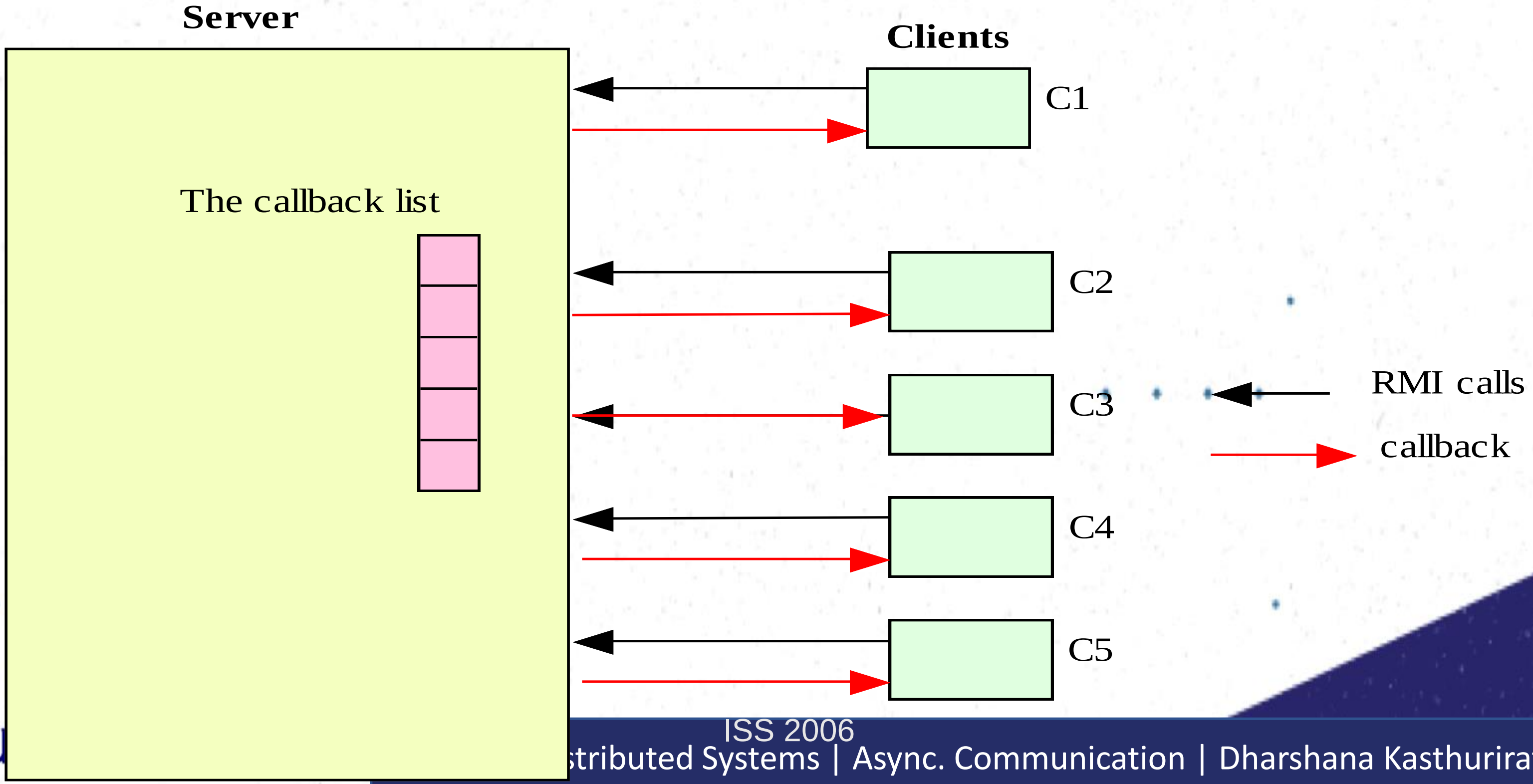
RMI Callbacks

1. Clients Register with the Server:Each client implements a remote interface and registers itself with the server.

2. Server Triggers a Callback: When an event occurs, the server calls a method on all registered clients.

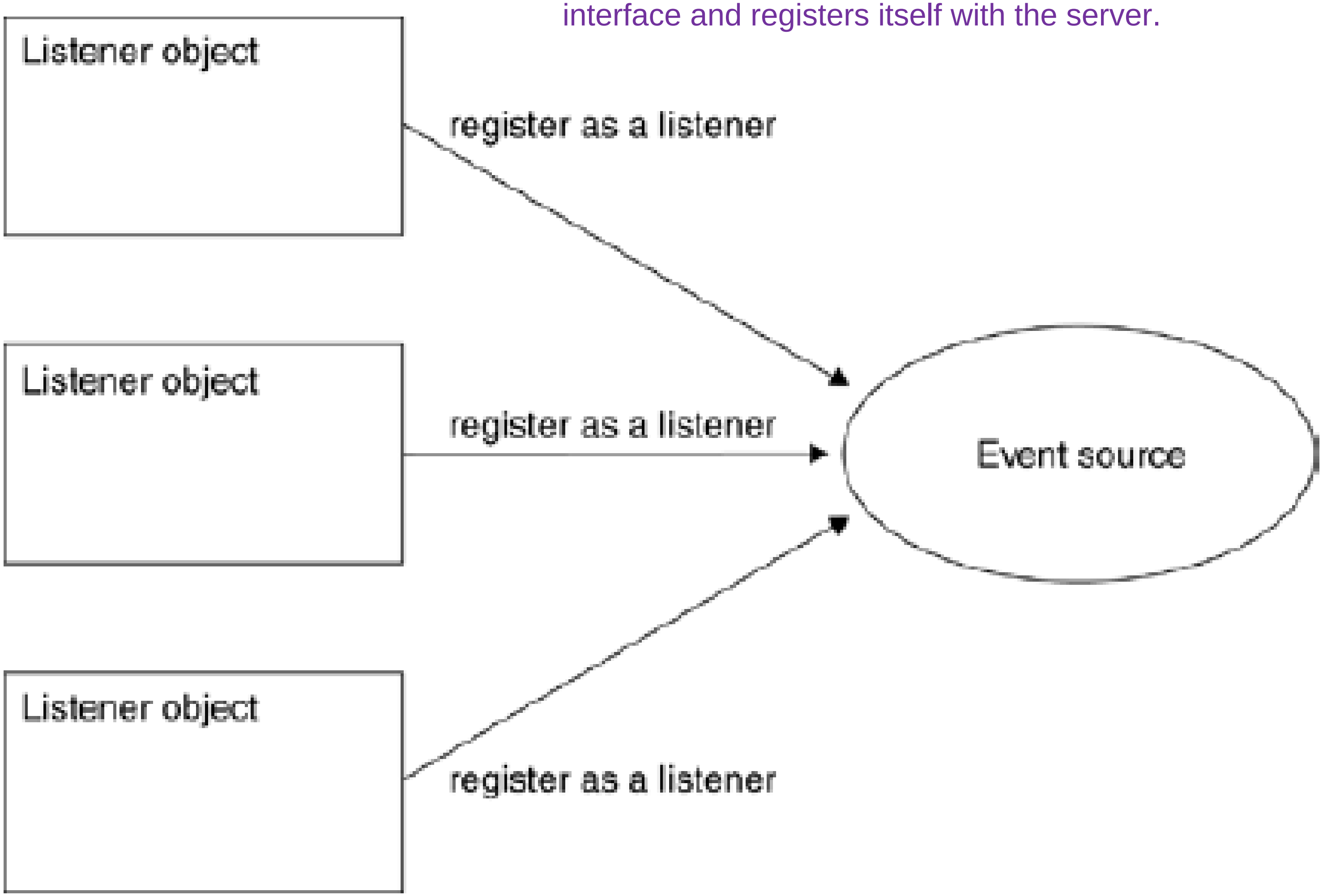
3. Clients Respond to the Callback: Each client receives the callback and executes a predefined method.The client can take appropriate action based on the event.

- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.

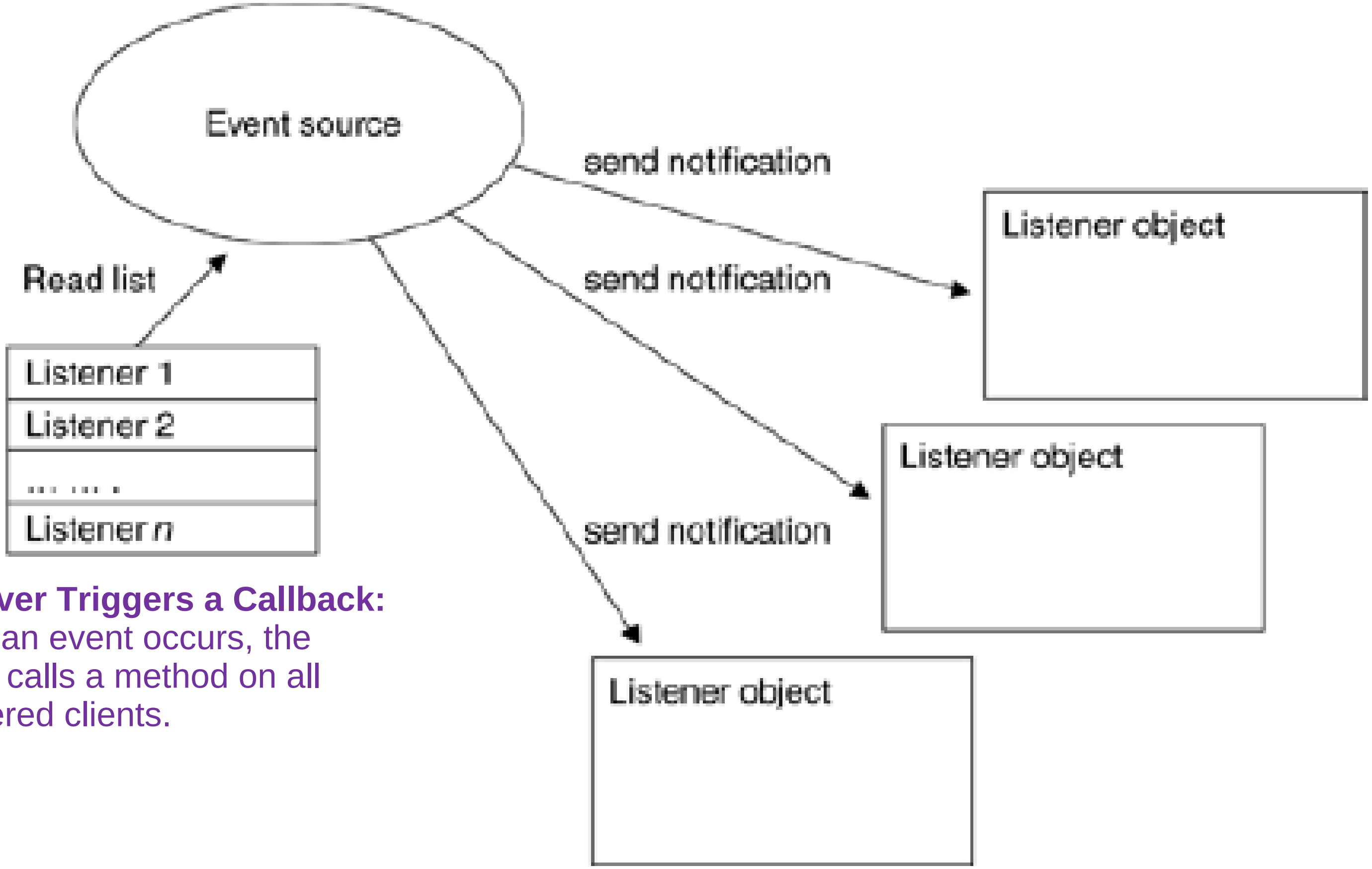


Multiple listeners

1. Clients Register with the Server: Each client implements a remote interface and registers itself with the server.



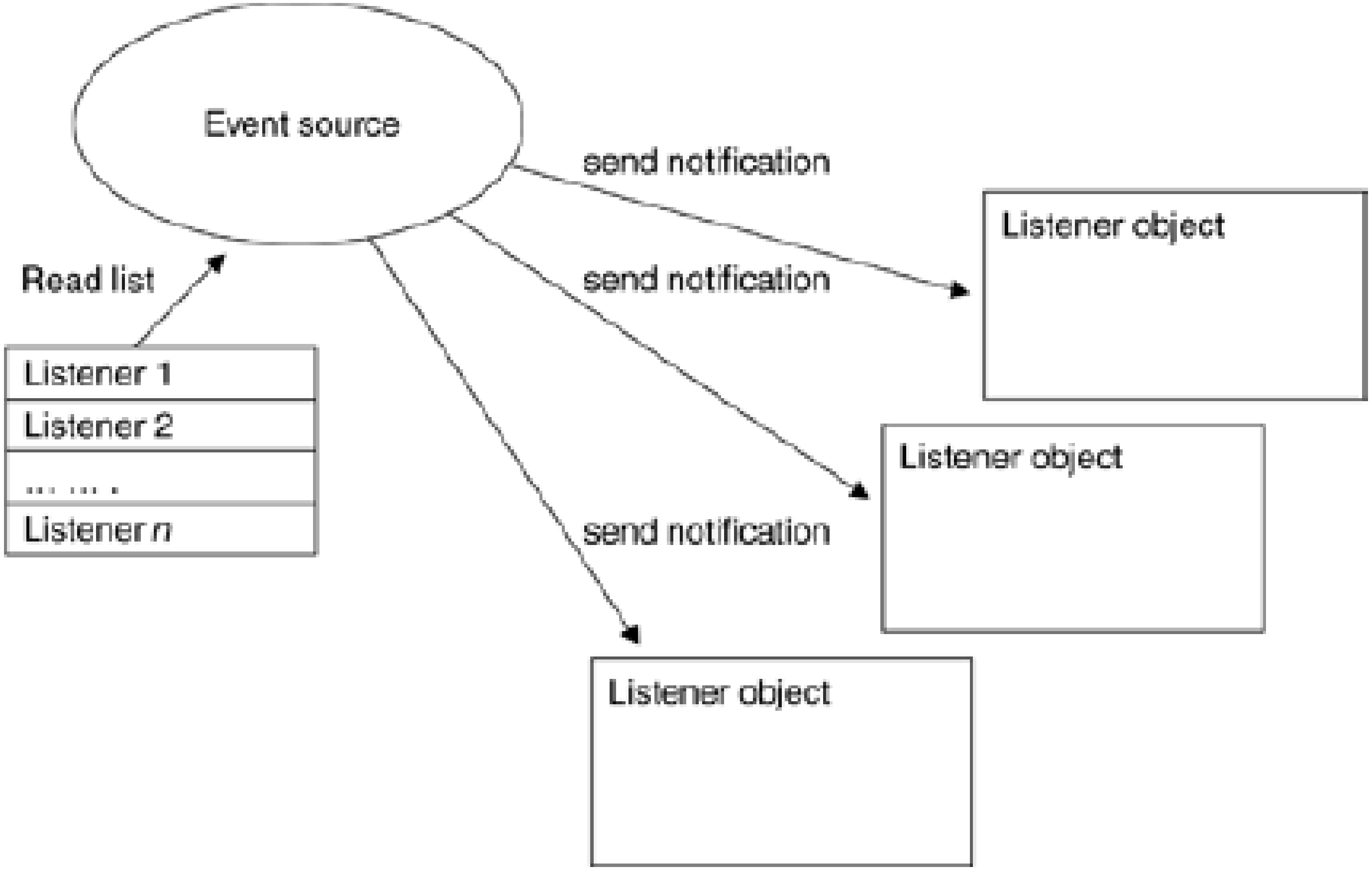
Callback notification to every registered listener



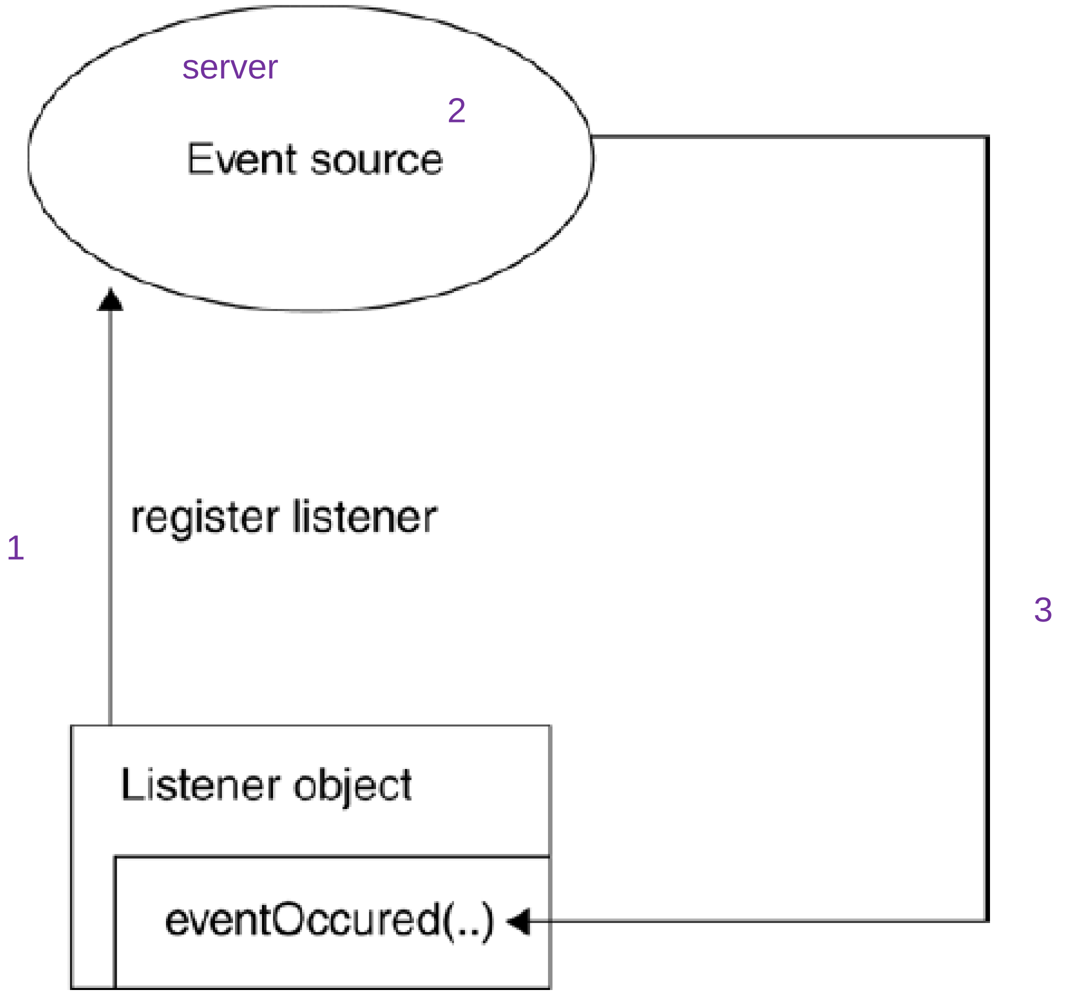
2. Server Triggers a Callback: When an event occurs, the server calls a method on all registered clients.

Callback notification to every registered listener

3. Clients Respond to the Callback: Each client receives the callback and executes a predefined method.The client can take appropriate action based on the event.



Callback implemented by invoking a method on a listening object



© Maert 2006

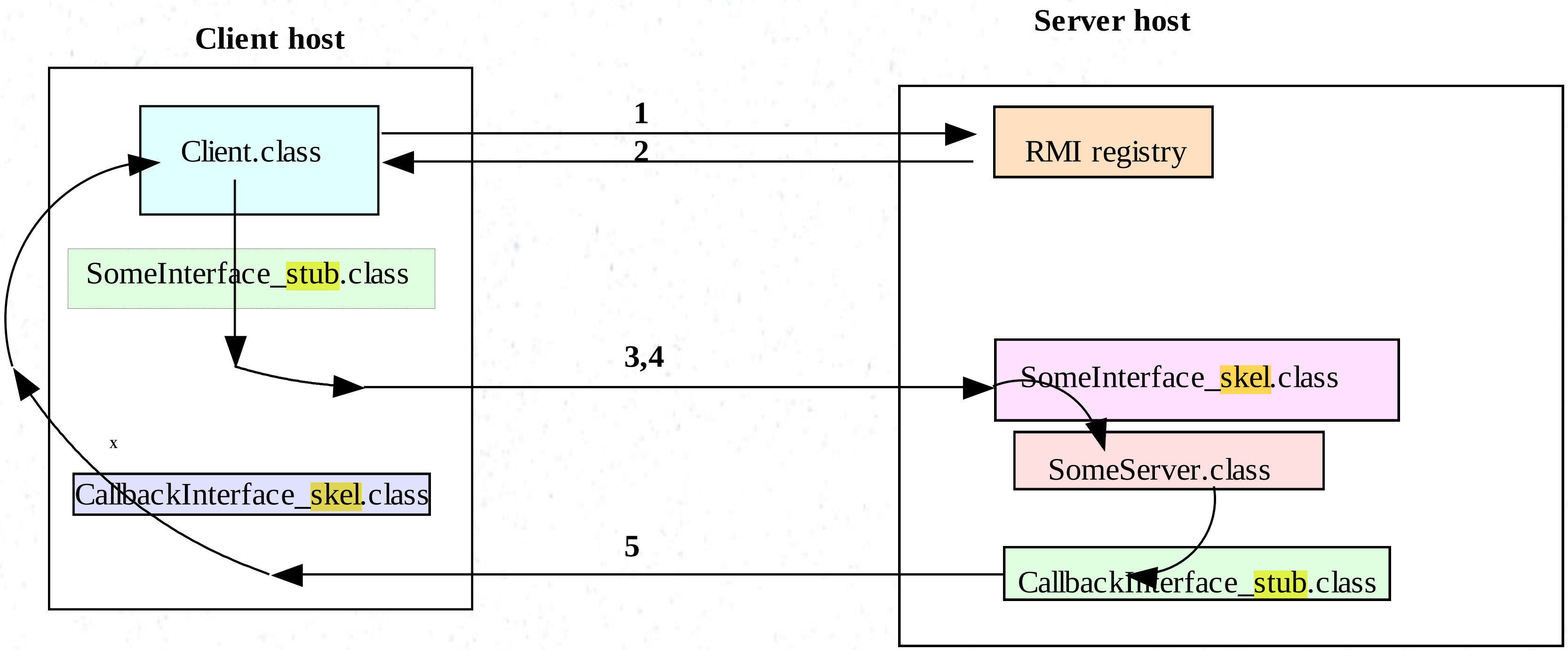
Client callback

- To provide client callback, the client-side software
  - supplies a remote interface,
  - instantiate an object which implements the interface,
  - passes a reference to the object to the server via a remote method call to the server.

Client callback

- The remote server:
  - collects these client references in a data structure.
  - when the awaited event occurs, the remote server invokes the callback method (defined in the client remote interface) to pass data to the client.
- Two sets of stub-skeletons are needed: one for the server remote interface, the other one for the client remote interface.

Callback Client-Server Interactions



1. Client looks up the interface object in the RMIregistry on the server host.
2. The RMIRegistry returns a remote reference to the interface object.
3. Via the server stub, the client process invokes a remote method to register itself for callback, passing a remote reference to itself to the server. The server saves the reference in its callback list.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the interface object.
5. When the anticipated event takes place, the server makes a callback to each registered client via the callback interface stub on the server side and the callback interface skeleton on the client side.

RMI Callback Example

- Temperature monitoring system
- Server will sense the temperature of the environment
- The Server will notify the client listeners of the changes in temperature
- Polling is not efficient thus we can use callback as a means asynchronous notifications
- In addition to the normal RMI classes/interfaces there will be a client interface defined as well (so that the server can ‘call back’)

RMI Callback Example

- Server Interface (same as in blocking RMI)

```
interface TemperatureSensor extends
java.rmi.Remote
{
  public double getTemperature() throws
     java.rmi.RemoteException;
  public void addTemperatureListener
     (TemperatureListener listener )
     throws java.rmi.RemoteException;
  public void removeTemperatureListener
     (TemperatureListener listener )
     throws java.rmi.RemoteException;
}
```

RMI Callback Example

- Listener Interface (Client side)

```
interface TemperatureListener extends
java.rmi.Remote
{
        public void temperatureChanged(double
temperature)
                 throws java.rmi.RemoteException;
}
```

- Defines the callback method

RMI Callback Example

- Server Implementation

```
public class TemperatureSensorServer extends
UnicastRemoteObject implements TemperatureSensor,
Runnable{

public void addTemperatureListener ( TemperatureListener
listener ) throws java.rmi.RemoteException{
   list.add (listener);
}
public void run(){
    for (;;) {   {
   if(checkTempChanged()){
   // Notify registered listeners
   notifyListeners();
   }
   }
}
```

RMI Callback Example

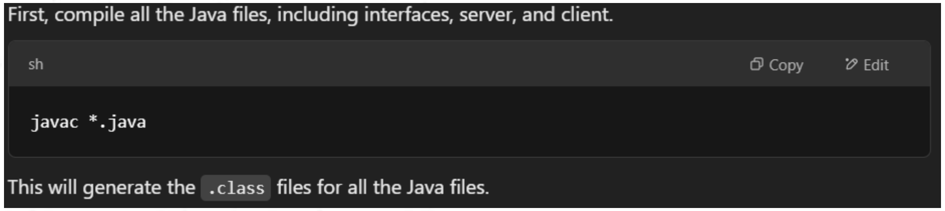
```
private void notifyListeners(){
for (Enumeration e = list.elements(); e.hasMoreElements(); ){
   TemperatureListener listener = (TemperatureListener)
   e.nextElement();
   listener.temperatureChanged (temp);
   list.remove( listener );
  }
}
public static void main(String args[]){
   TemperatureSensorServer sensor = new
   TemperatureSensorServer();
   String registration = "rmi://" + registry
   +"/TemperatureSensor";
   Naming.rebind( registration, sensor );
   Thread thread = new Thread (sensor);
   thread.start();
}
}
```

RMI Callback Example

Client implementation

```
public class TemperatureMonitor extends UnicastRemoteObject
   implements TemperatureListener{

public static void main(String args[]){
   Remote remoteService = Naming.lookup ( registration );
   TemperatureSensor sensor =   (TemperatureSensor)remoteService;
   double reading = sensor.getTemperature();
   System.out.println ("Original temp : " + reading);
   TemperatureMonitor monitor = new TemperatureMonitor();
   sensor.addTemperatureListener(monitor);
}
public void temperatureChanged(double temperature)
throws java.rmi.RemoteException
{
   System.out.println ("Temperature change event : " +
   temperature);
}
```

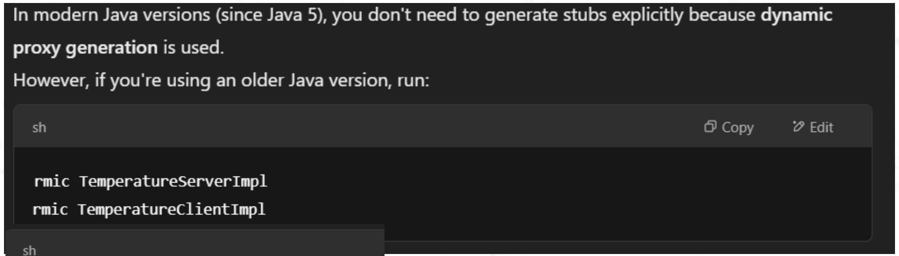
Running the example


First, compile all the Java files, including interfaces, server, and client.

```sh
javac *.java
```

This will generate the `.class` files for all the Java files.

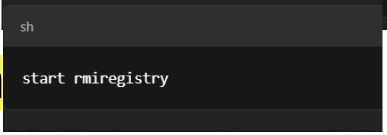
1. Compile the applications and generate stub/skeleton files for both


In modern Java versions (since Java 5), you don't need to generate stubs explicitly because **dynamic proxy generation** is used.

However, if you're using an older Java version, run:

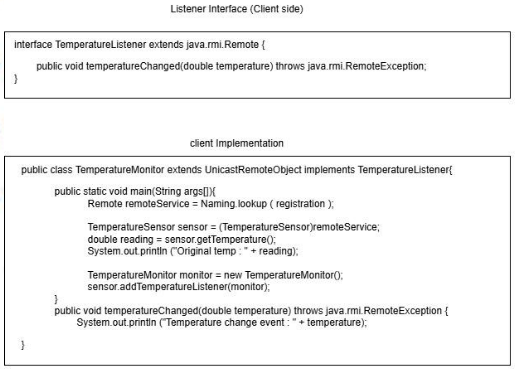
```sh
rmic TemperatureServerImpl
rmic TemperatureClientImpl
```

2. TemperatureSensorServer and TemperatureSensorMonitor.
3. Run the rmiregistry application



```sh
start rmiregistry
```

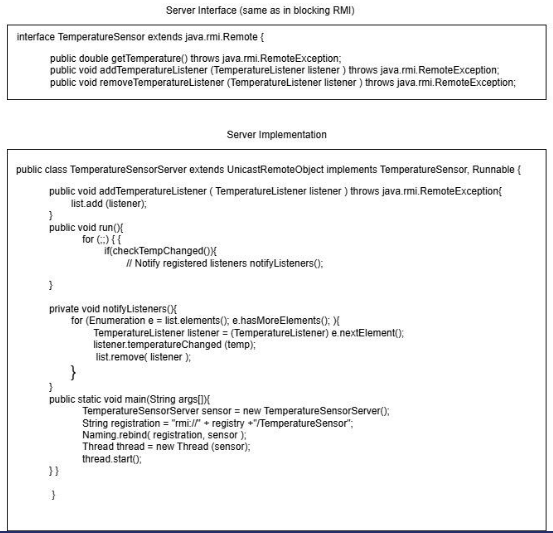
4. Run the TemperatureSensorServer. server
5. Run the TemperatureSensorMonitor. client


Listener Interface (Client side)

```java
interface TemperatureListener extends java.rmi.Remote {

      public void temperatureChanged(double temperature) throws java.rmi.RemoteException;
}
```

client Implementation

```java
 public class TemperatureMonitor extends UnicastRemoteObject implements TemperatureListener{

         public static void main(String args[]){
                 Remote remoteService = Naming.lookup ( registration );

                 TemperatureSensor sensor = (TemperatureSensor)remoteService;
                 double reading = sensor.getTemperature();
                 System.out.println ("Original temp : " + reading);

                 TemperatureMonitor monitor = new TemperatureMonitor();
                 sensor.addTemperatureListener(monitor);
         }
         public void temperatureChanged(double temperature) throws java.rmi.RemoteException {
               System.out.println ("Temperature change event : " + temperature);

 }
```


Server Interface (same as in blocking RMI)

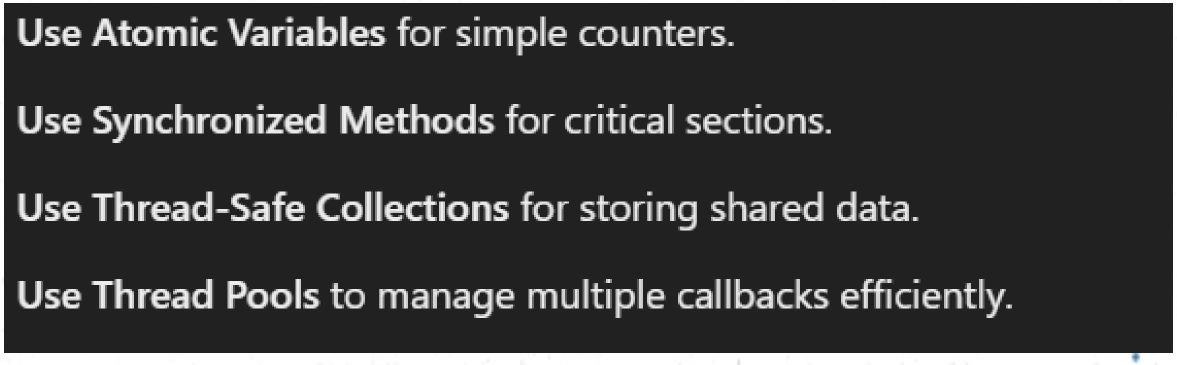
```java
interface TemperatureSensor extends java.rmi.Remote {

        public double getTemperature() throws java.rmi.RemoteException;
        public void addTemperatureListener (TemperatureListener listener ) throws java.rmi.RemoteException;
        public void removeTemperatureListener (TemperatureListener listener ) throws java.rmi.RemoteException;
```

Server Implementation

```java
public class TemperatureSensorServer extends UnicastRemoteObject implements TemperatureSensor, Runnable {

        public void addTemperatureListener ( TemperatureListener listener ) throws java.rmi.RemoteException{
              list.add (listener);
        }
        public void run(){
                for (;;) { {
                      if(checkTempChanged()){
                            // Notify registered listeners notifyListeners();

        }

        private void notifyListeners(){
              for (Enumeration e = list.elements(); e.hasMoreElements(); ){
                    TemperatureListener listener = (TemperatureListener) e.nextElement();
                    listener.temperatureChanged (temp);
                     list.remove( listener );
              }
        }
        public static void main(String args[]){
                TemperatureSensorServer sensor = new TemperatureSensorServer();
                String registration = "rmi://" + registry +"/TemperatureSensor";
                Naming.rebind( registration, sensor );
                Thread thread = new Thread (sensor);
                thread.start();
        } }

         }
```

Asynchronous Callback functions and thread safety

- Callback functions use threads in the background
- Main thread does the remote call and then a worker thread calls the callback function
- Main thread is running at the same time
- Have to handle thread safety issues manually



Use Atomic Variables for simple counters.

Use Synchronized Methods for critical sections.

Use Thread-Safe Collections for storing shared data.

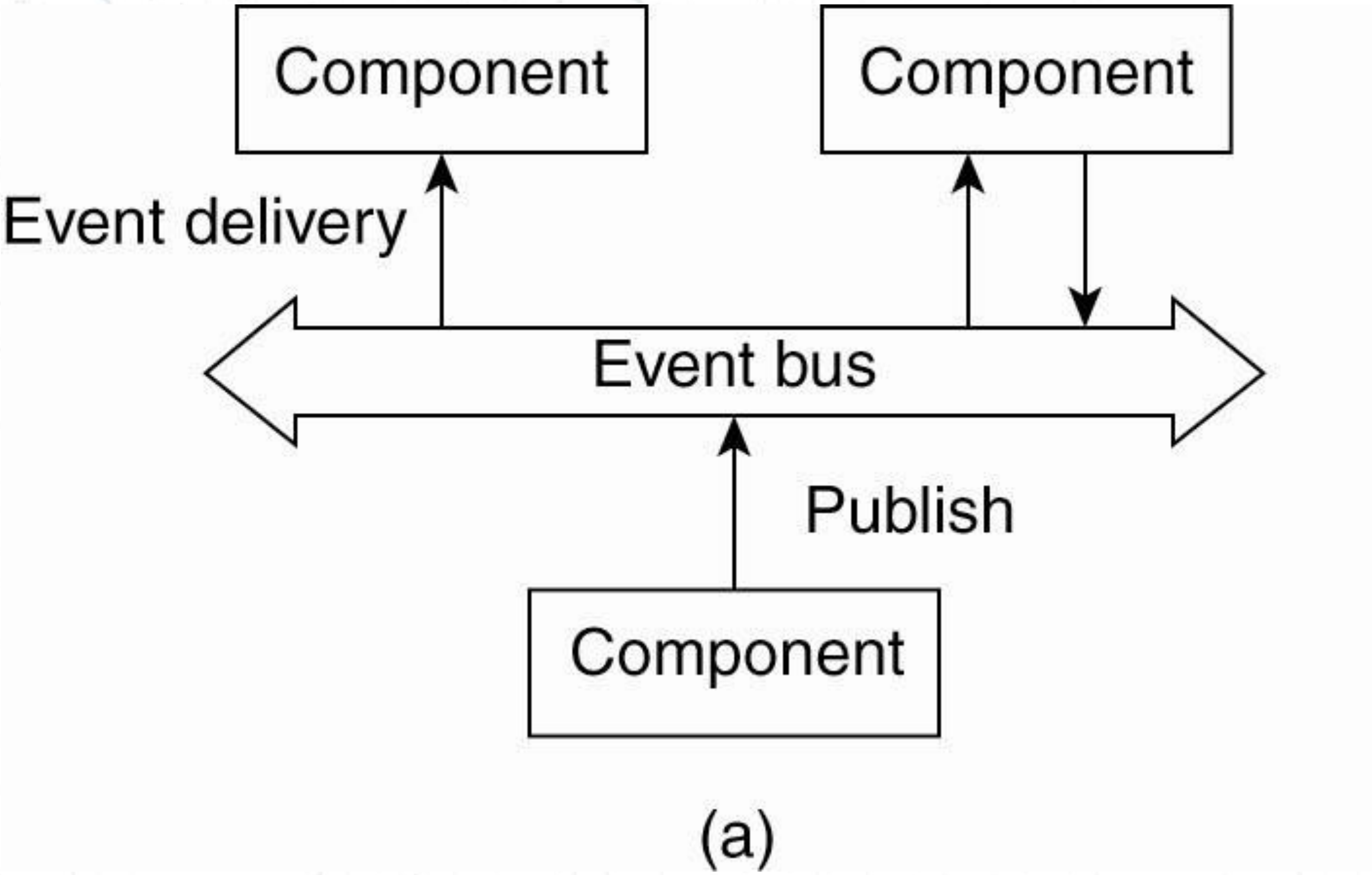
Use Thread Pools to manage multiple callbacks efficiently.

Asynchronous Messaging services

A communication model where messages are sent and received asynchronously.
Sender and receiver do not need to be online at the same time.
Supports event-driven architectures for loosely coupled systems.

Event based Architectures

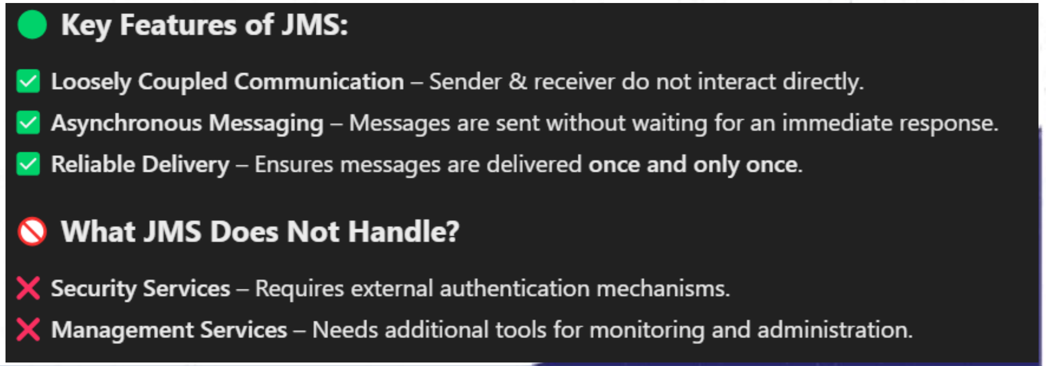
Designed for real-time systems where actions (events) trigger processes.



-

Java Message Service (JMS)

- A **specification** that describes a common way for Java programs to create, send, receive and read distributed enterprise messages
- *loosely coupled* communication
- *Asynchronous* messaging
- *Reliable* delivery
  - A message is guaranteed to be delivered once and only once.
- Outside the specification
  - Security services
  - Management services



● **Key Features of JMS:**

✅ **Loosely Coupled Communication** – Sender & receiver do not interact directly.

✅ **Asynchronous Messaging** – Messages are sent without waiting for an immediate response.

✅ **Reliable Delivery** – Ensures messages are delivered **once and only once**.

🚫 **What JMS Does Not Handle?**

❌ **Security Services** – Requires external authentication mechanisms.

❌ **Management Services** – Needs additional tools for monitoring and administration.

A JMS Application

everyone has a client

- JMS Clients
  - Java programs that send/receive messages
- Messages - The actual data exchanged between clients.
- Administered Objects
  - preconfigured JMS objects created by an admin for the use of clients

    Creates connections to the messaging system.
  - ConnectionFactory, Destination (queue or topic)

    Defines message queues (for PTP) or topics (for pub-sub).
- JMS Provider

  company. like IBM
  - messaging system that implements JMS and administrative functionality

activem

JMS Messaging Domains

JMS supports two messaging models:

- Point-to-Point (PTP)
  - built around the concept of message queues
  - each message has only one consumer

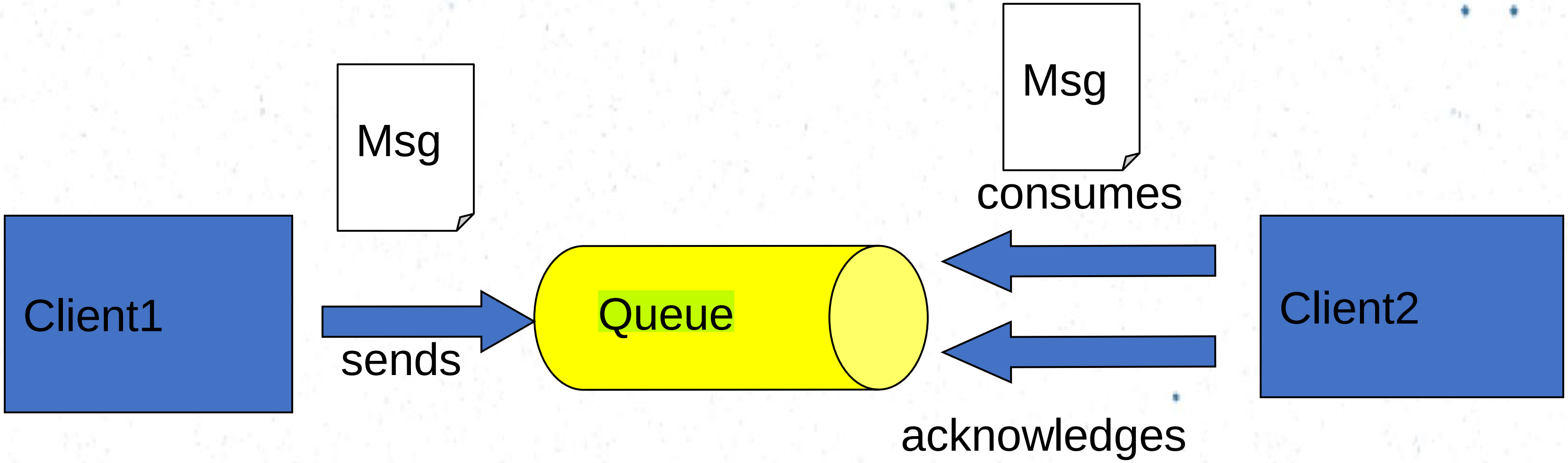
Each message is delivered to only one consumer.

Used in task queues, order processing, job execution.

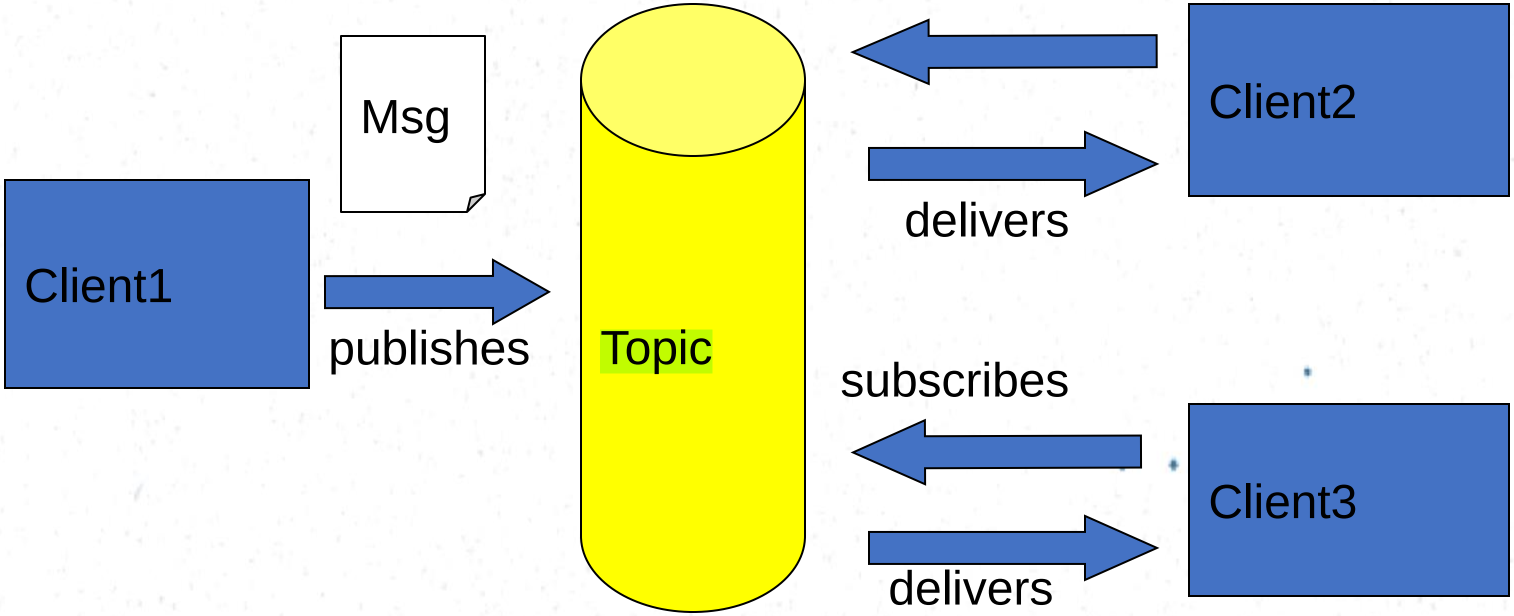
- Publish-Subscribe systems
  - uses a “topic” to send and receive messages
  - each message has multiple consumers

Uses topics instead of queues.
Each message can be received by multiple subscribers.
Used in news broadcasting, stock updates, live notifications.

Point-to-Point Messaging



Publish/Subscribe Messaging



Message Consumptions

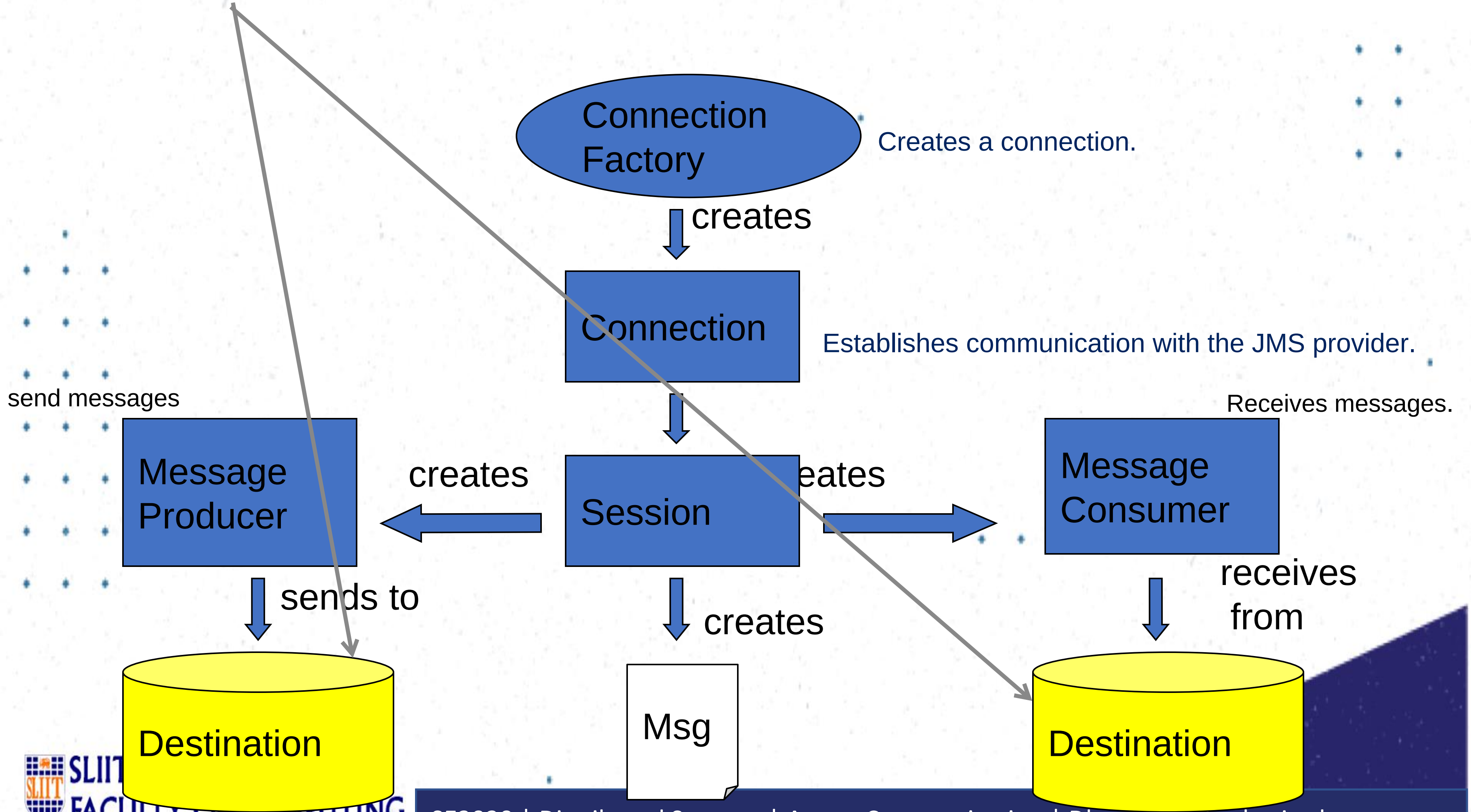
JMS supports two ways to consume messages:

- Synchronously
  - A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
  - The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit. The receiver explicitly fetches the message using the receive() method.
- Asynchronously
  - A client can register a *message listener* with a consumer.
  - Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage() method.

The JMS provider calls the onMessage() method whenever a message arrives.

JMS API Programming Model

Destination (Queue/Topic) – Defines where messages are sent or received.



Creates a connection.

Establishes communication with the JMS provider.

send messages

Receives messages.

JMS Client Example

Before sending or receiving messages, the JMS client must set up a connection to the JMS provider. The following steps are typical:

1

- Setting up a connection and creating a session

```
InitialContext jndiContext=new InitialContext();
```

Create an InitialContext

```
//look up for the connection factory
ConnectionFactory
  cf=jndiContext.lookup(connectionfactoryname);
```

Look Up the ConnectionFactory

```
//create a connection
Connection connection=cf.createConnection();
```

Create a Connection

```
//create a session
Session
  session=connection.createSession(false,Session.AUTO_ACKNOWL
  EDGE);
```

Create a Session

```
//create a destination object
Destination dest1=(Queue)
  jndiContext.lookup(“/jms/myQueue”); //for PointToPoint
Destination dest2=(Topic)jndiContext.lookup(“/jms/myTopic”);
  //for publish-subscribe
```

Look Up the Destination:
Depending on the messaging model you want to use, you can look up a Queue (for point-to-point messaging) or a Topic (for publish-subscribe messaging).javaCopyEdit

2.Producer Sample

JMS producer is responsible for sending messages to a destination.

- Setup connection and create a session
- Creating producer

Create a MessageProducer for the chosen destination.

```
MessageProducer producer=session.createProducer(dest1);
```

- Send a message

```
Message m=session.createTextMessage();
m.setText(“just another message”);
producer.send(m);
```

- Closing the connection

```
connection.close();
```

3. Consumer Sample (Synchronous)

synchronous consumption, the consumer explicitly calls a receive method to fetch the message.

- Setup connection and create a session
- Creating consumer

```
MessageConsumer consumer=session.createConsumer(dest1);
```

- Start receiving messages

```
connection.start();
Message m=consumer.receive();
```

The call to receive() blocks until a message is available (or times out).

Consumer Sample (Asynchronous)

the client registers a listener that is called automatically when a message arrives.

- Setup the connection, create a session
- Create consumer
- Registering the listener
  - MessageListener listener=new myListener();
  - consumer.setMessageListener(listener);
- myListener should have onMessage()

```
public void onMessage(Message msg){
  //read the massage and do computation
}
```

Create a listener that implements the onMessage() method.

Listener Example

```
public void onMessage(Message message) {
   TextMessage msg = null;
   try {
       if (message instanceof TextMessage) {
           msg = (TextMessage) message;
           System.out.println("Reading message: " + msg.getText());
       } else {
           System.out.println("Message of wrong type: " +
               message.getClass().getName());
       }
   } catch (JMSException e) {
       System.out.println("JMSException in onMessage(): " + e.toString());
   } catch (Throwable t) {
       System.out.println("Exception in onMessage():" + t.getMessage());
   }
}
```

JMS Messages

Message Structure:

- Message Header
  - used for identifying and routing messages
  - contains vendor-specified values, but could also contain application-specific data
  - typically name/value pairs
- Message Properties (optional)

Application-specific values that can help in message selection.

- Message Body(optional)
  - contains the data
  - five different message body types in the JMS specification

JMS Message Types

Message Type	Contains	Some Methods
TextMessage	String	getText,setText
MapMessage	set of name/value pairs	setString,setDouble,setLong,getDouble,getString
BytesMessage	stream of uninterpreted bytes	writeBytes,readBytes
StreamMessage	stream of primitive values	writeString,writeDouble, writeLong,readString
ObjectMessage	serialize object	setObject,getObject

More JMS Features

- Durable subscription
  - by default a subscriber gets only messages published on a topic while a subscriber is alive
  - durable subscription retains messages until a they are received by a subscriber or expire
- Request/Reply
  - by creating temporary queues and topics
    - Session.createTemporaryQueue()

      so that the producer can set a reply destination. The correlation ID is used to match replies to requests.
  - producer=session.createProducer(msg.getJMSReplyTo());
    reply= session.createTextMessage("reply");
    reply.setJMSCorrelationID(msg.getJMSMessageID);
    producer.send(reply);

SLIIT FACULTY OF COMPUTING

More JMS Features

Sessions can be transacted so that a combination of sending and receiving operations are part of a single atomic transaction. If an error occurs, the session can roll back.

- Transacted sessions
  - session=connection.createSession(true,0)
  - combination of queue and topic operation in one transaction is allowed
  - void onMessage(Message m) {

```
    try { Message m2=processOrder(m);
        publisher.publish(m2); session.commit();
} catch(Exception e) { session.rollback(); }
```

More JMS Features

Persistent: The message is stored and guaranteed to survive a provider failure.
Nonpersistent: Offers higher performance but with no guarantee of message recovery.

- Persistent/nonpersistent delivery
  - producer.setDeliveryMethod(DeliveryMode.NON_PERSISTENT);
  - producer.send(mesg, DeliveryMode.NON_PERSISTENT ,3,1000);
- Message selectors

MS allows the use of SQL-like expressions to filter messages based on header properties.

  - SQL-like syntax for accessing header:

subscriber = session.createSubscriber(topic, “priority > 6 AND type = ‘alert’ ”);

  - Point to point: selector determines single recipient
  - Pub-sub: acts as filter

JMS Providers

- SunONE Message Queue (SUN)
- MQ JMS (IBM)
- WebLogic JMS (BEA)
- JMSCourier (Codemesh)
- Apache ActiveMQ

JMS API in a JEE Application

- Since the J2EE1.3 , the JMS API has been an integral part of the platform
- JEE components can use the JMS API to send messages that can be consumed asynchronously by a specialized Enterprise Java Bean
  - message-driven bean

Microsoft Messaging Queue

- .NET Equivalent of JMS

https://msdn.microsoft.com/en-us/library/ms731089.aspx

Summary

- Asynchronous Communication helps to make non blocking calls among distributed components
- It maximizes the performance and response time of distributed Systems
- Callback functions and Messaging Services are two common ways of implementing asynchronous communication
- Some calls may have to be synchronous (blocking) if further processing cannot be done without the information in server response