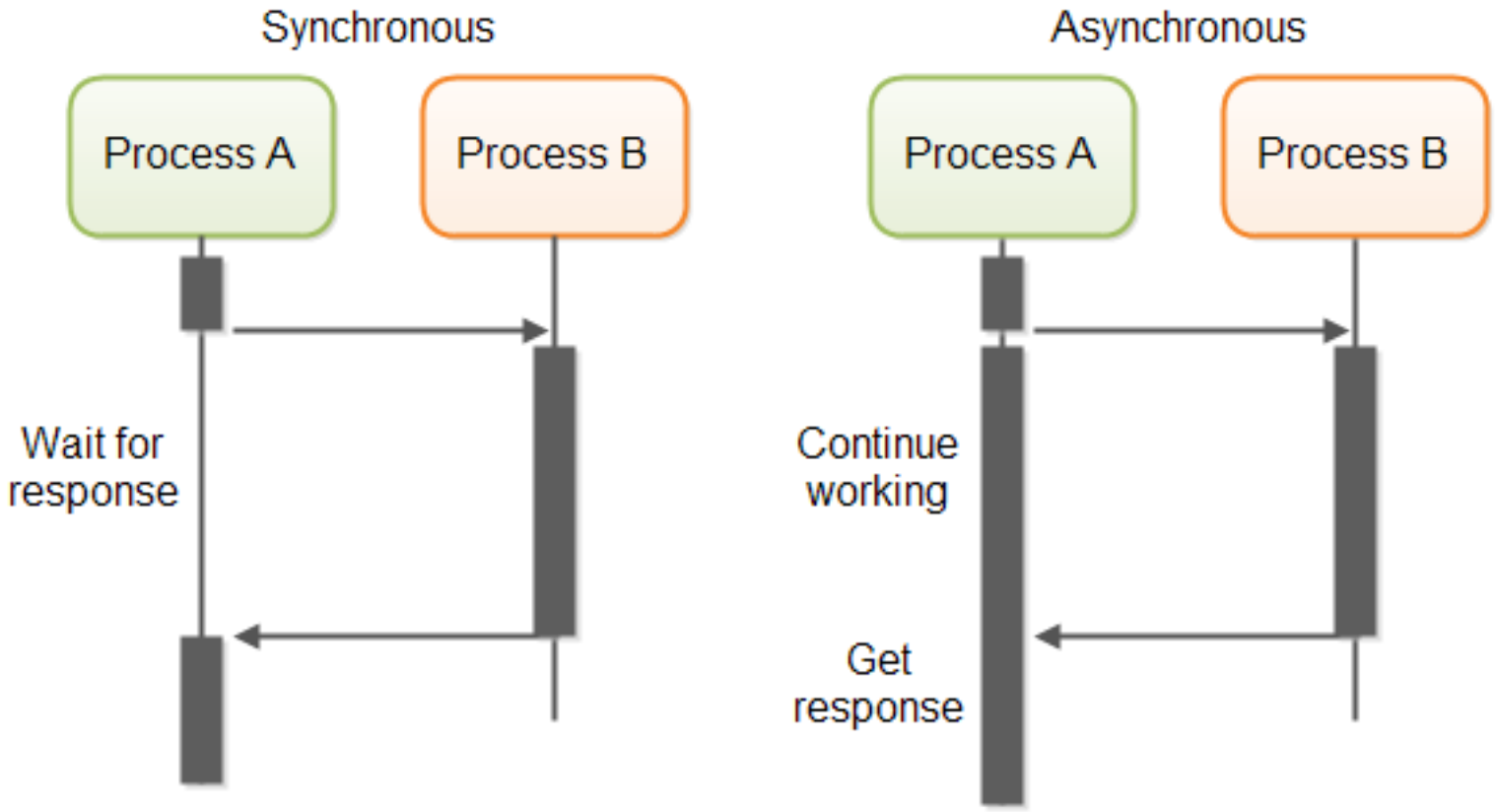
Lecture 5

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


Synchronous
Process A
Process B
Wait for response
Asynchronous
Process A
Process B
Continue working
Get response

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
  - Especially if it is a call that could take a while
  - i.e. Client waits while server performs a long job
  - Not utilising resources properly/efficiently there!

Synchronous vs. Asynchronous

- 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;
- Some applications require the server to initiate communication upon certain events.
    - monitoring
    - games
    - auctioning
    - voting/polling
    - chat-room
    - message/bulletin board
    - groupware

When to use Asynchronous Calls?

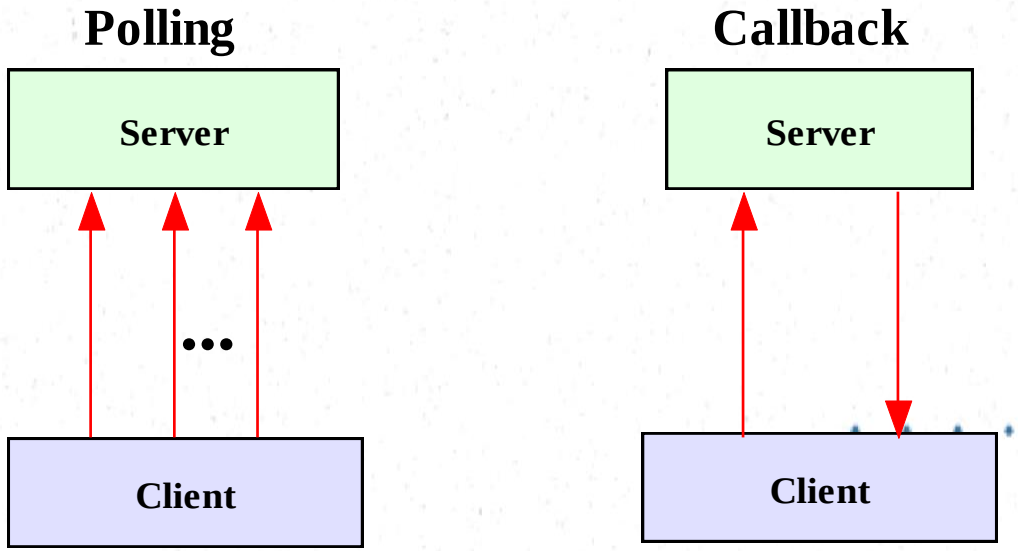
- *Every* RPC call is potentially long-running
  - Network/server failures are only detected after timeouts expire
  - Making every RPC call asynchronous increases code complexity, just on the *chance* a network failure occurs
- So use asynchronous calls only on functions that are *expected* to take 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
- Messaging (e.g. JMS, Microsoft Messaging Queuing)
- Both Java and .NET supports callback functions

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.

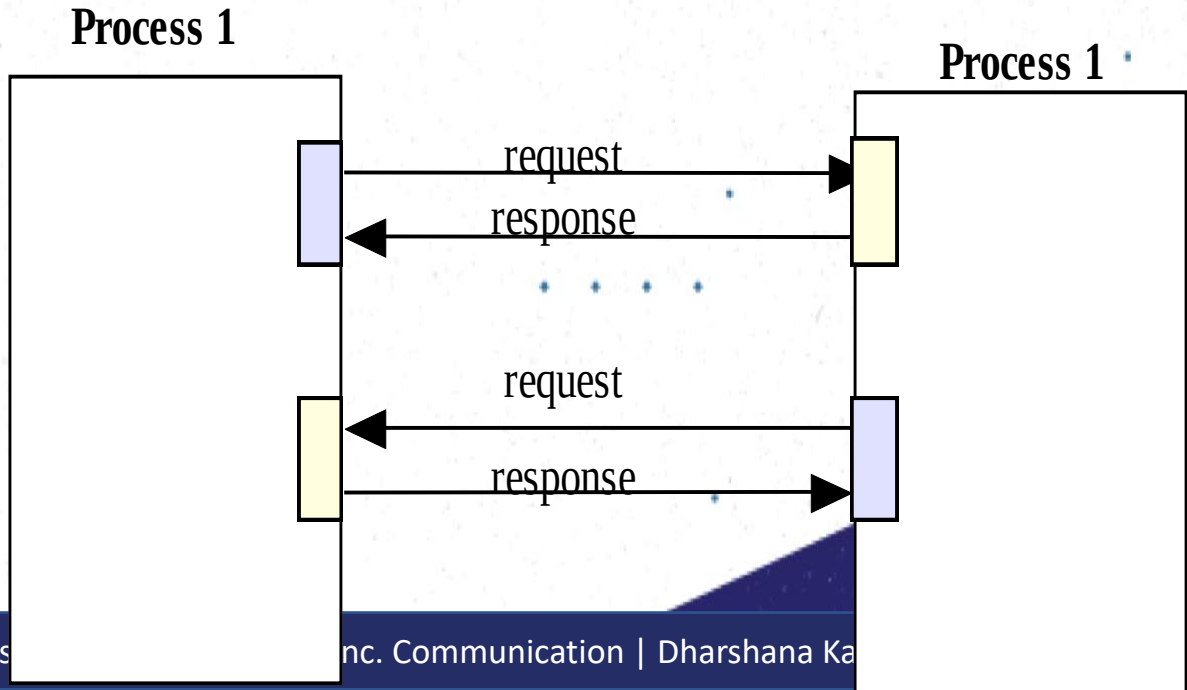


Polling vs. Callback

- Blocking is like making a call and waiting for the other party to respond (if the other party is busy with some other call)
- Polling is like repeatedly making a telephone call and check whether the other party is available.
- Callback is like making a call and leaving a message to other party to call back with certain information

Two-way communications

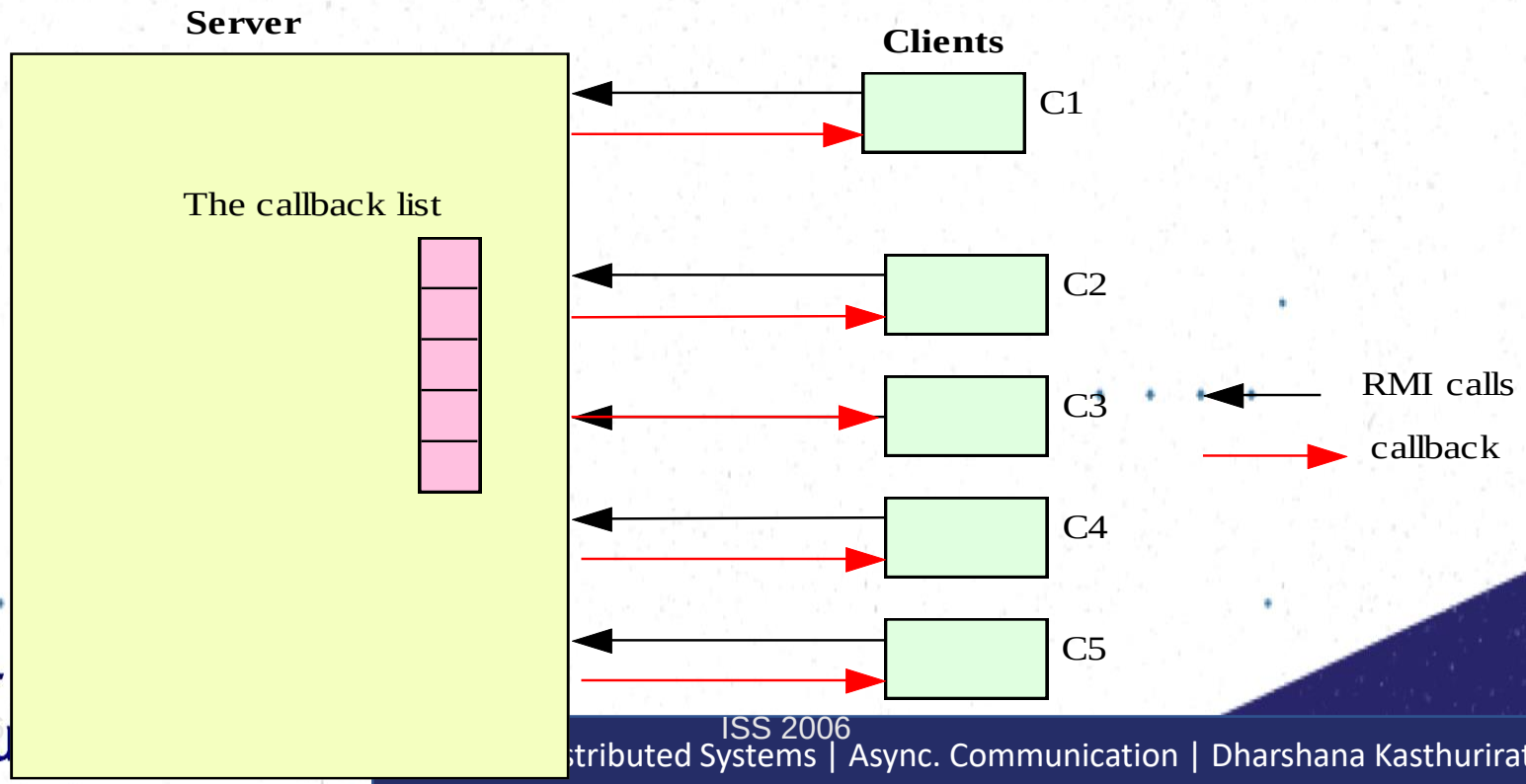
- Some applications require that both sides may initiate IPC.
- Using sockets, duplex communication can be achieved by using two sockets on either side.
- With connection-oriented sockets, each side acts as both a client and a server.



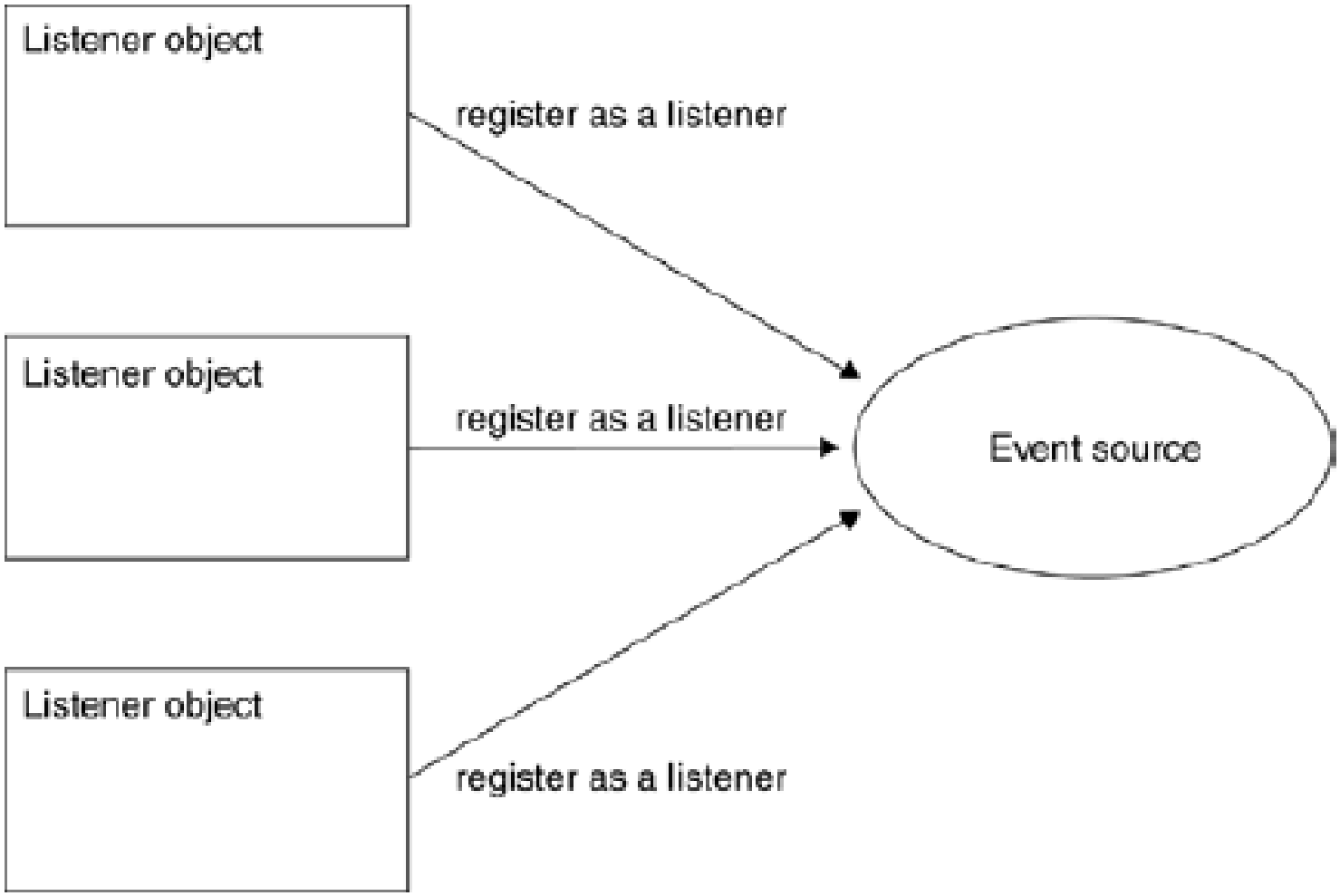
RMI Callbacks

RMI Callbacks

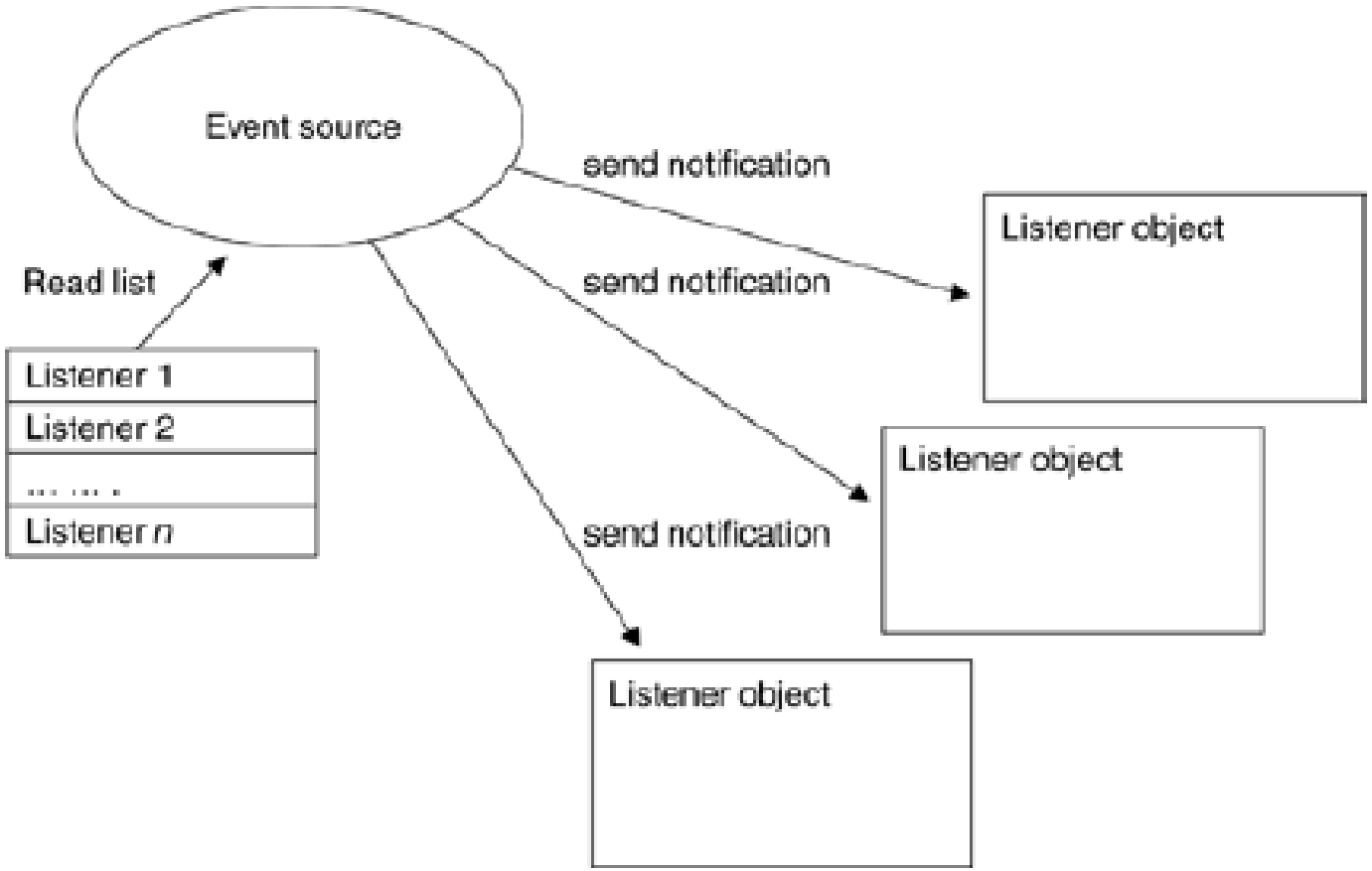
- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.



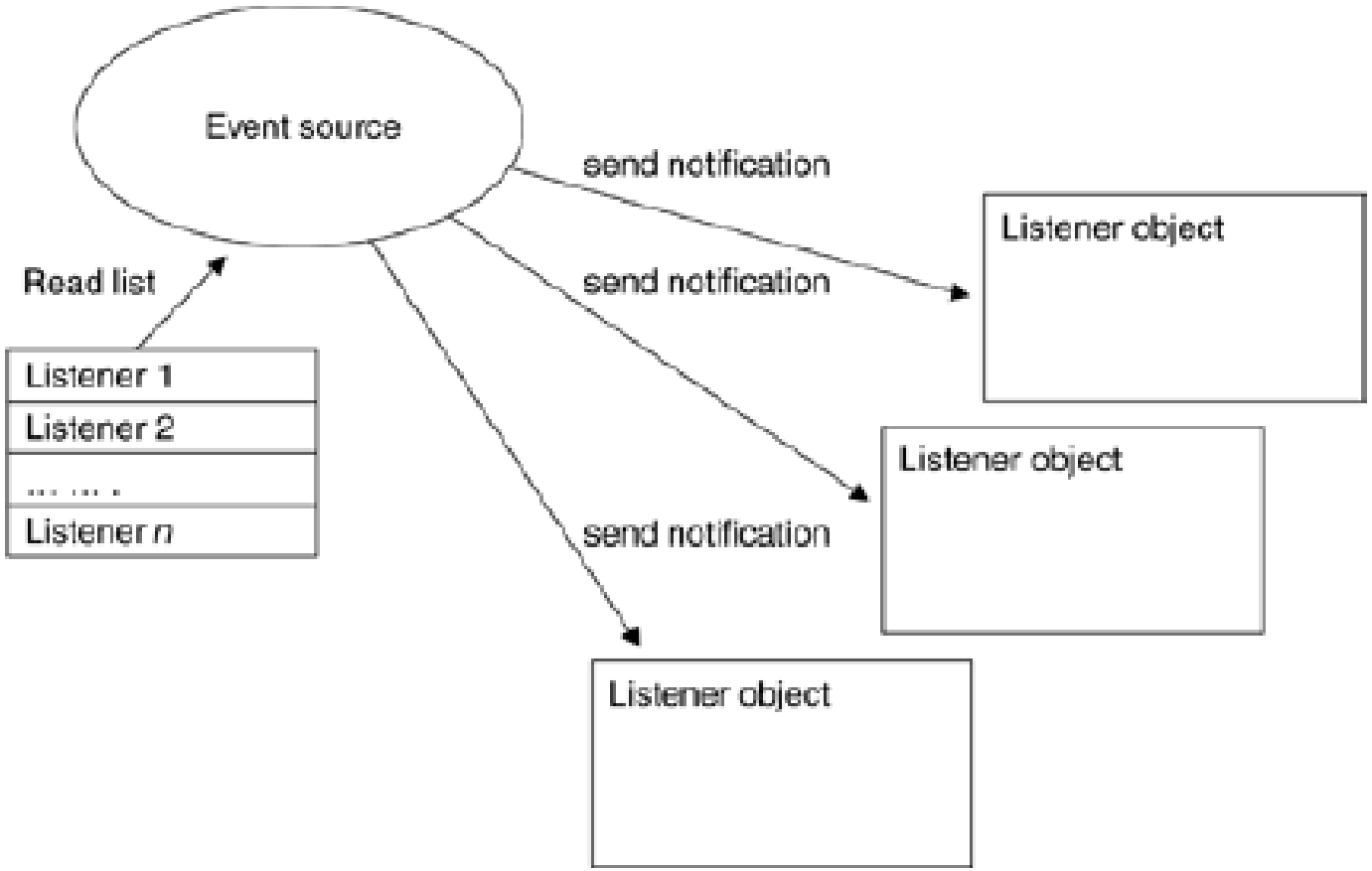
Multiple listeners



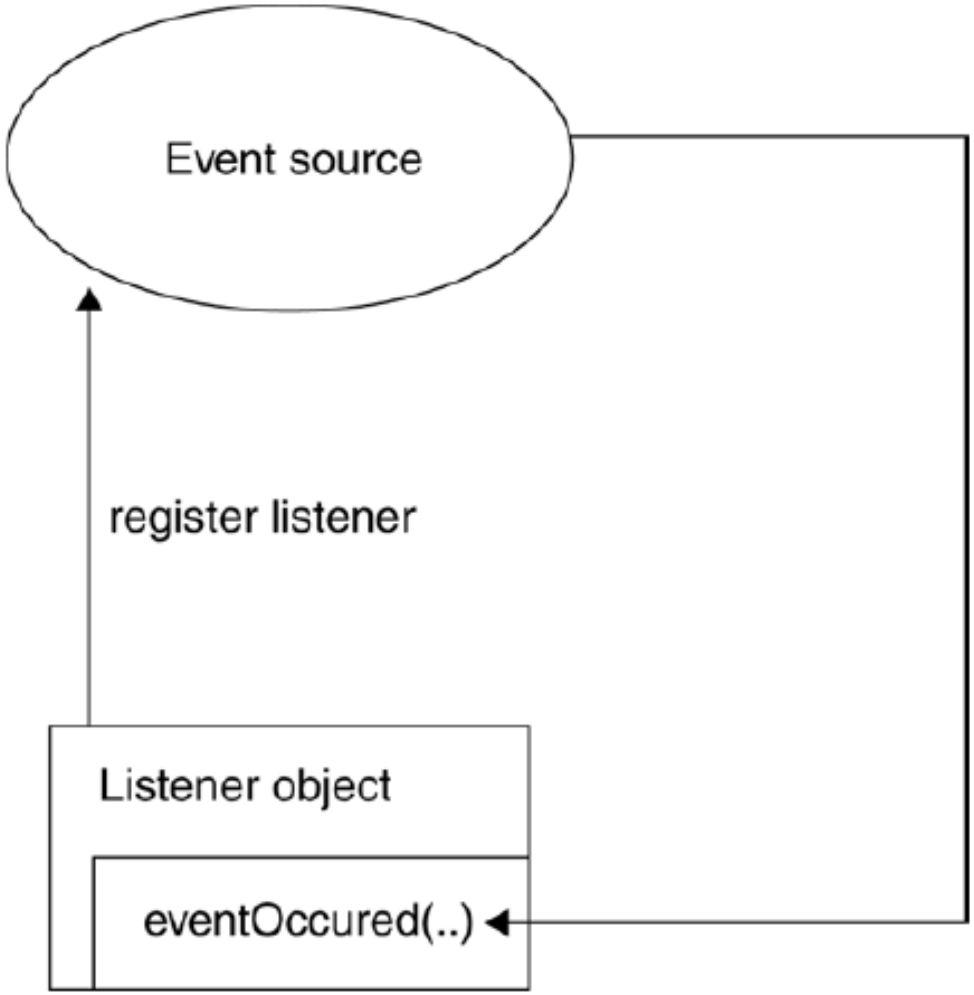
Callback notification to every registered listener



Callback notification to every registered listener



Callback implemented by invoking a method on a listening object



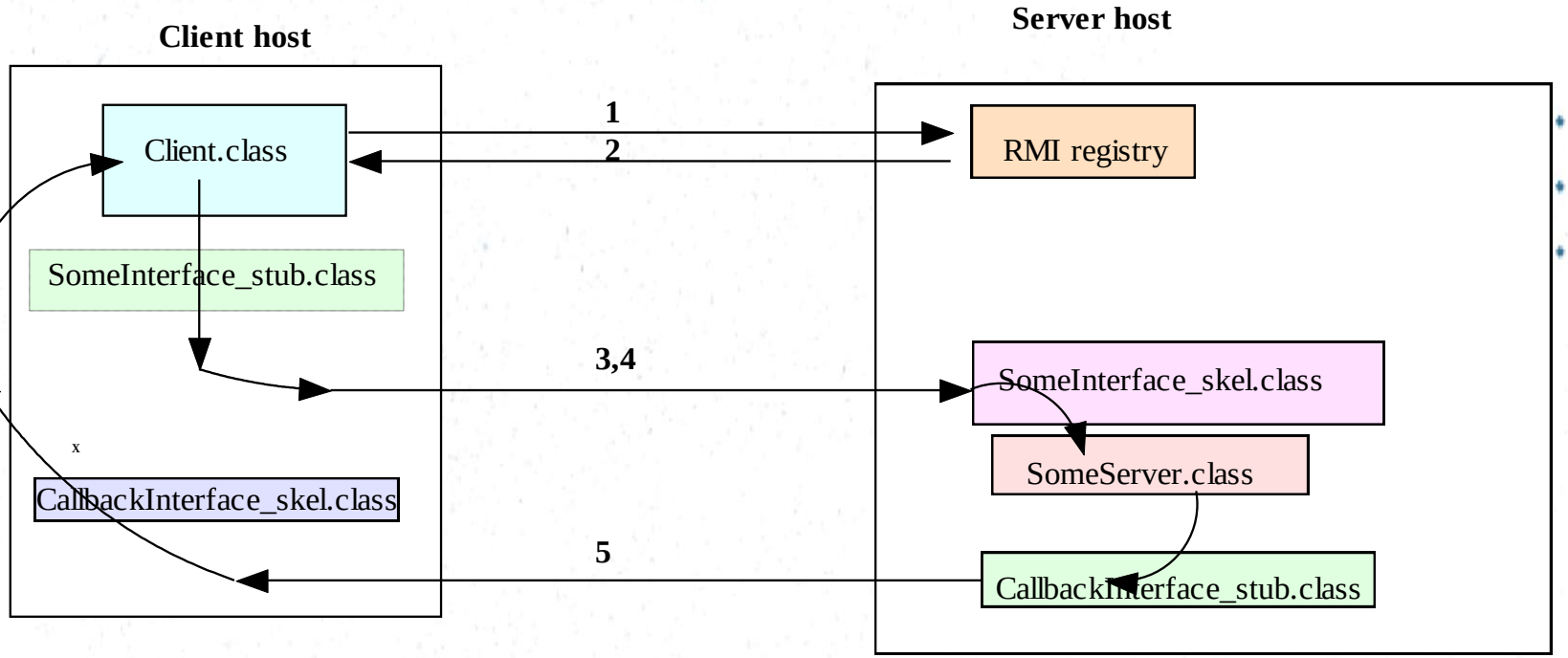
Client callback

- 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

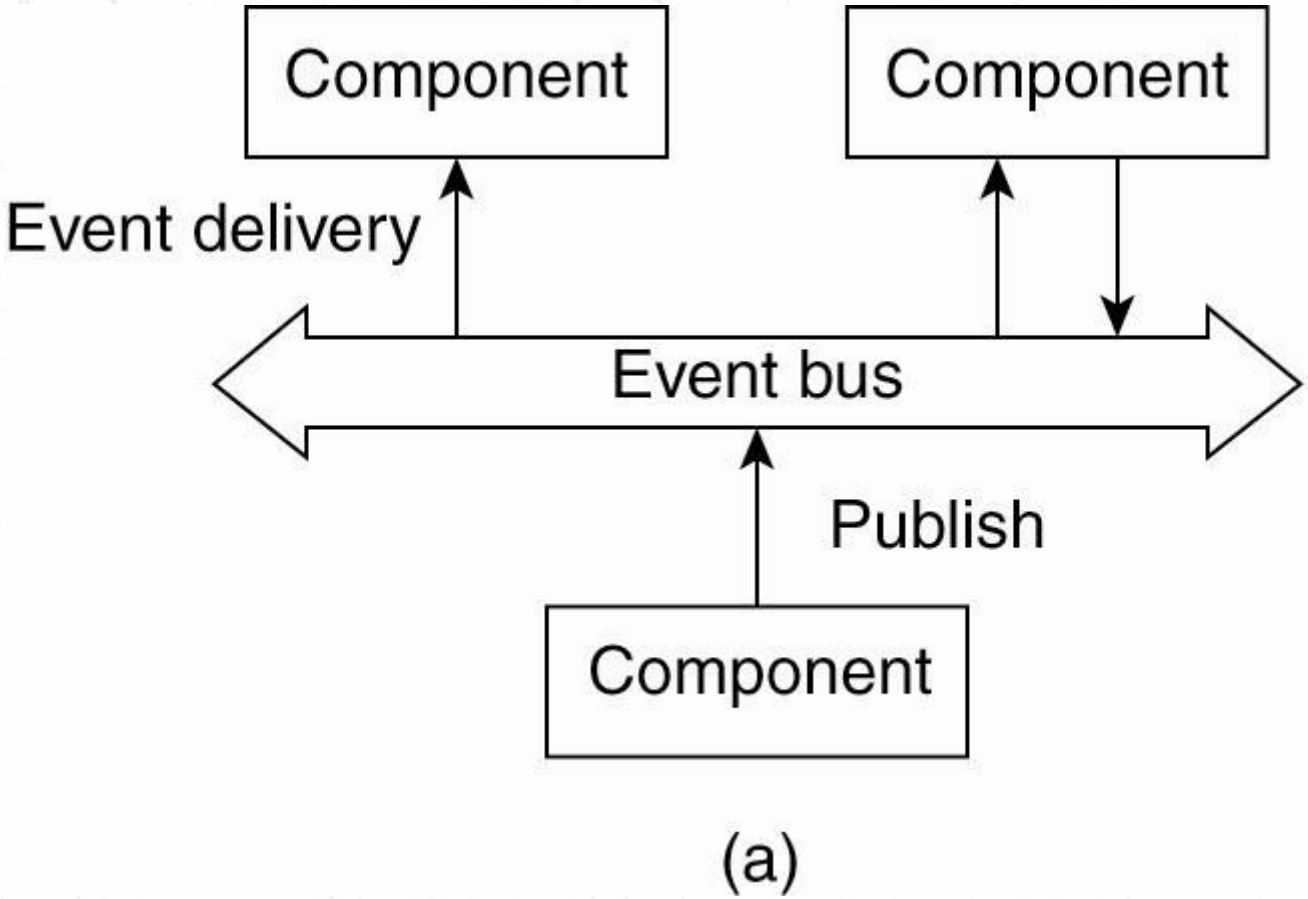
1. Compile the applications and generate stub/skeleton files for both
2. TemperatureSensorServer and TemperatureSensorMonitor.
3. Run the rmiregistry application.
4. Run the TemperatureSensorServer.
5. Run the TemperatureSensorMonitor.

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

Asynchronous Messaging services

Event based Architectures



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

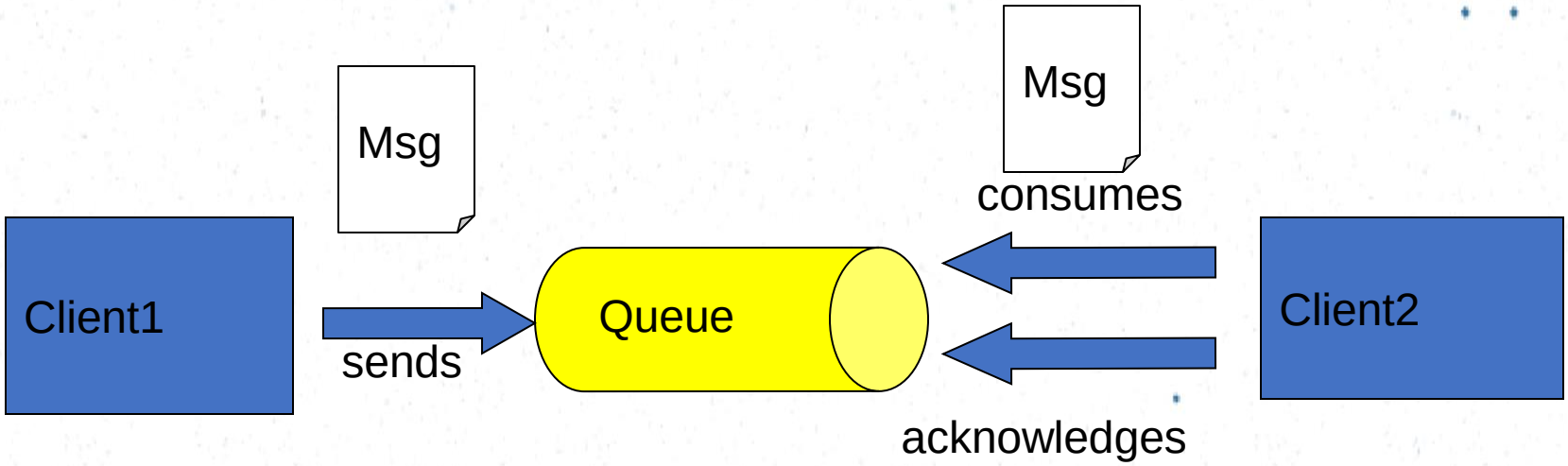
A JMS Application

- JMS Clients
  - Java programs that send/receive messages
- Messages
- Administered Objects
  - preconfigured JMS objects created by an admin for the use of clients
  - ConnectionFactory, Destination (queue or topic)
- JMS Provider
  - messaging system that implements JMS and administrative functionality

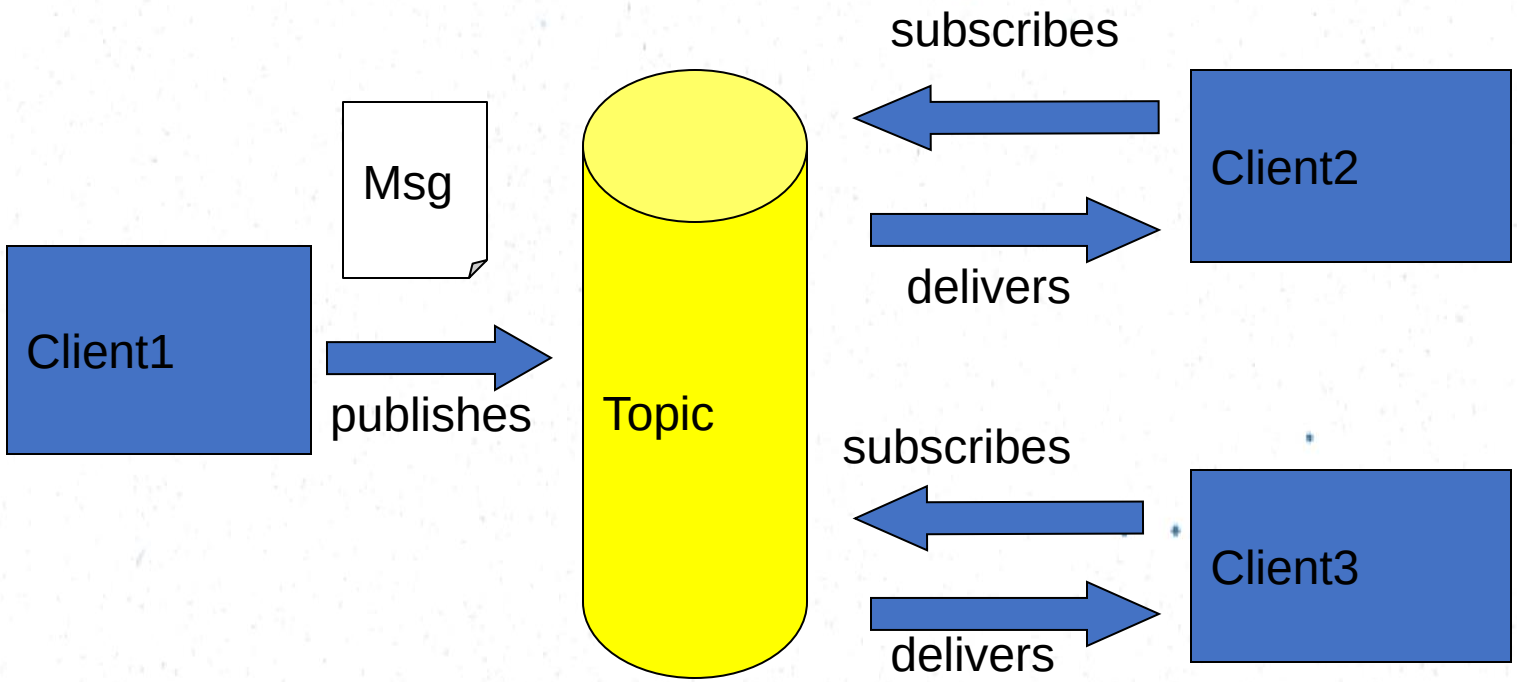
JMS Messaging Domains

- Point-to-Point (PTP)
  - built around the concept of message queues
  - each message has only one consumer
- Publish-Subscribe systems
  - uses a “topic” to send and receive messages
  - each message has multiple consumers

Point-to-Point Messaging



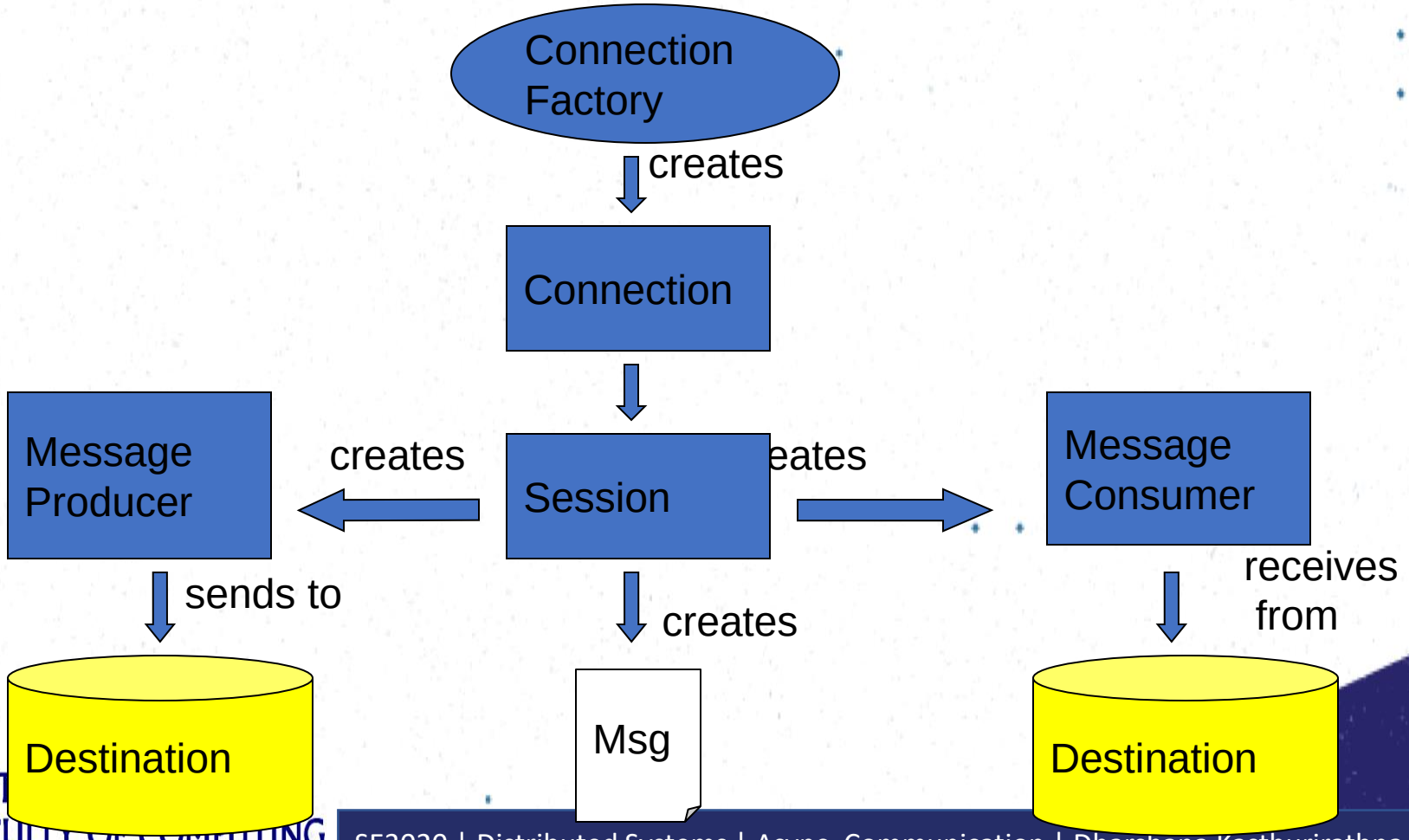
Publish/Subscribe Messaging



Message Consumptions

- 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.
- 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.

JMS API Programming Model



JMS Client Example

- Setting up a connection and creating a session

```
InitialContext jndiContext=new InitialContext();
//look up for the connection factory
ConnectionFactory
  cf=jndiContext.lookup(connectionfactoryname);
//create a connection
Connection connection=cf.createConnection();
//create a session
Session
  session=connection.createSession(false,Session.AUTO_ACKNOWL
  EDGE);
//create a destination object
Destination dest1=(Queue)
  jndiContext.lookup("/jms/myQueue"); //for PointToPoint
Destination dest2=(Topic)jndiContext.lookup("/jms/myTopic");
  //for publish-subscribe
```

Producer Sample

- Setup connection and create a session
- Creating producer

```
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();
```

Consumer Sample (Synchronous)

- Setup connection and create a session
- Creating consumer

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

- Start receiving messages

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

Consumer Sample (Asynchronous)

- 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
}
```

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 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)
- 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()
  - producer=session.createProducer(msg.getJMSReplyTo());
    reply= session.createTextMessage("reply");
    reply.setJMSCorrelationID(msg.getJMSMessageID);
    producer.send(reply);

More JMS Features

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