# Concurrent & Distributed Systems

## Distributed Systems 4

### Inter-process communications

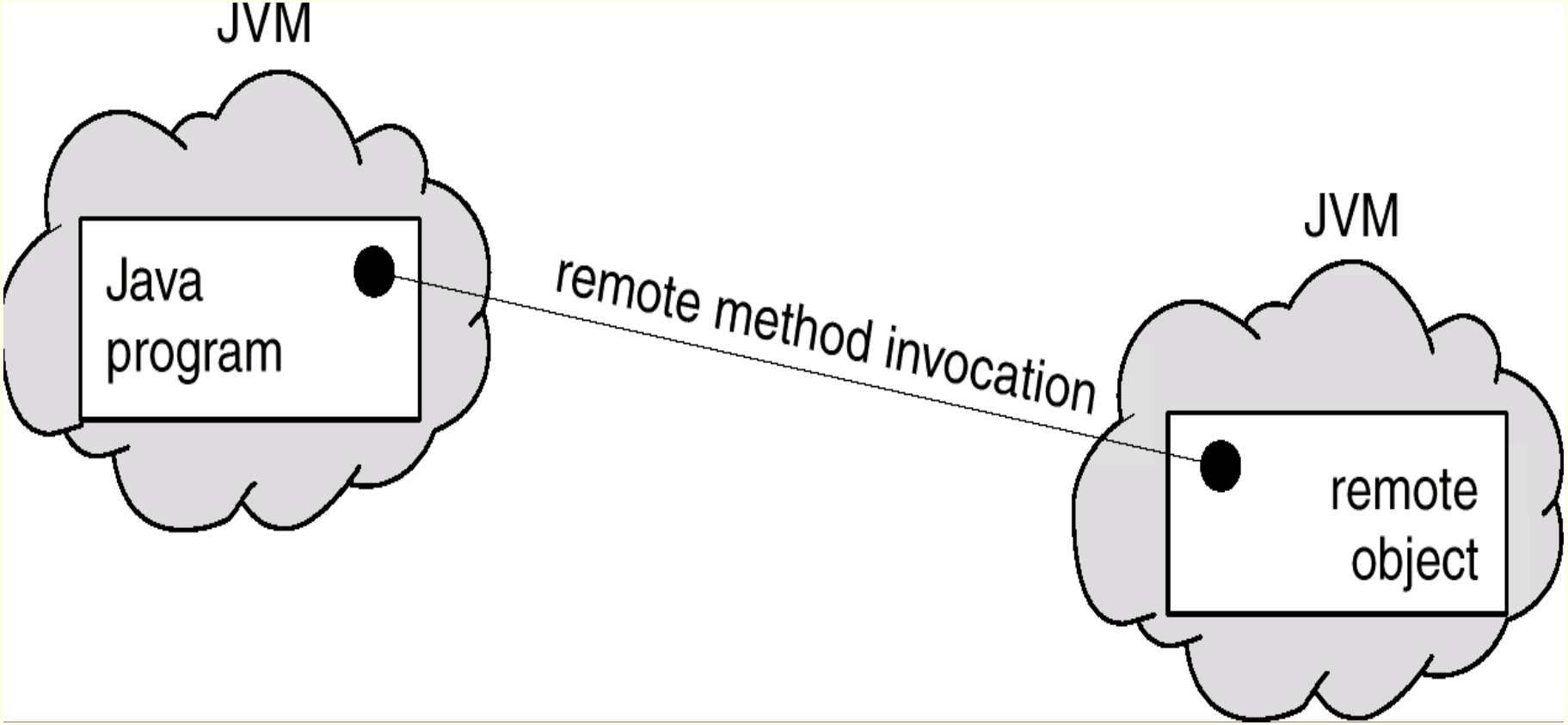
# inter-process communication

- files
- sockets
- message passing
- remote procedure calls
- ***remote method invocation***
- **(CORBA)**

# remote method invocation (rmi)

- shares many aspects of RPCs
  - Java's version of RPCs
  - marshalling and un-marshalling of data
  - stubs & skeletons --> server-side stub called a skeleton in old versions of java
  - remote methods - rather than procedures
  - a thread may invoke a Method on a Remote Object
  - an Object is considered "remote" if it resides in a separate Java Virtual Machine.
- References
  - "Applied Operating System Concepts", Silberschatz, Galvin & Gagne, Wiley, §15.3.

# Remote Method Invocation



# RPC versus RMI

- RPCs support Procedural Programming Style
- RMI supports Object-Oriented Programming Style

- parameters to RPCs are Ordinary Data Structures
- parameters to RMI are Objects

# Stubs and Skeletons

- "Stub" is a Proxy for the Remote Object – resides on Client.
- The Stub *marshalls* the parameters and sends them to the Server.

- "Skeleton" is on Server Side.
- Skeleton *un-marshalls* the parameters and delivers them to the Server.

Skeletons are no longer required for remote method calls in the Java 2 platform v1.2 and greater.

Since version 5.0 of J2SE; support for dynamically generated stub files has been added, and ***rmic*** is only provided for backwards compatibility with earlier runtimes.

https://docs.oracle.com/javase/7/docs/technotes/tools/windows/rmic.html

# rmic - The Java RMI Compiler

**rmic** generates stub, skeleton, and tie classes for remote objects using either the JRMP or IIOP protocols. Also generates OMG IDL.

## SYNOPSIS

rmic [ options ] *package-qualified-class-name(s)*

## DESCRIPTION

The **rmic** compiler generates **stub** and **skeleton class files** (JRMP protocol) and **stub** and **tie class files** (IIOP protocol) for remote objects. These classes files are generated **from compiled Java** programming language classes that are remote object implementation classes. A **remote implementation class** is a class that implements the interface **java.rmi.Remote**. The class names in the **rmic** command must be for classes that have been compiled successfully with the **javac** command and must be fully package qualified. For example, running **rmic** on the class file name HelloImpl as

```
rmic hello.HelloImpl
```

creates the HelloImpl_Stub.class file in the hello subdirectory (named for the class's package).

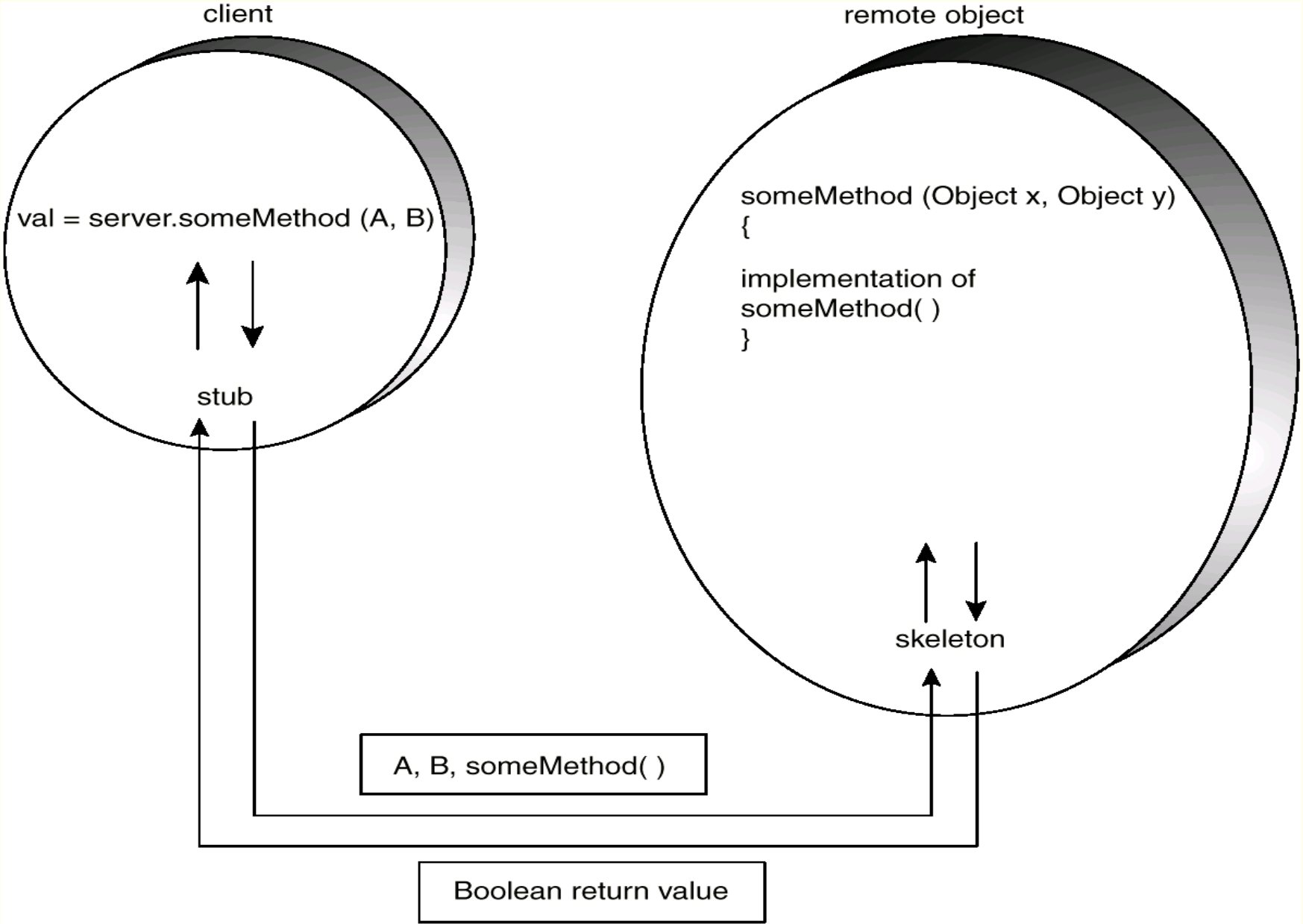
https://docs.oracle.com/javase/7/docs/technotes/tools/windows/rmic.html

# rmic - The Java RMI Compiler

**rmic** generates stub, skeleton, and tie classes for remote objects using either the JRMP or IIOP protocols. Also generates OMG IDL.

- A ***skeleton*** for a remote object is a JRMP protocol server-side entity that has a method that dispatches calls to the actual remote object implementation.
- A ***tie*** for a remote object is a server-side entity similar to a skeleton, but which communicates with the client using the IIOP protocol.
- A ***stub*** is a client-side proxy for a remote object which is responsible for communicating method invocations on remote objects to the server where the actual remote object implementation resides. A client's reference to a remote object, therefore, is actually a reference to a local stub.

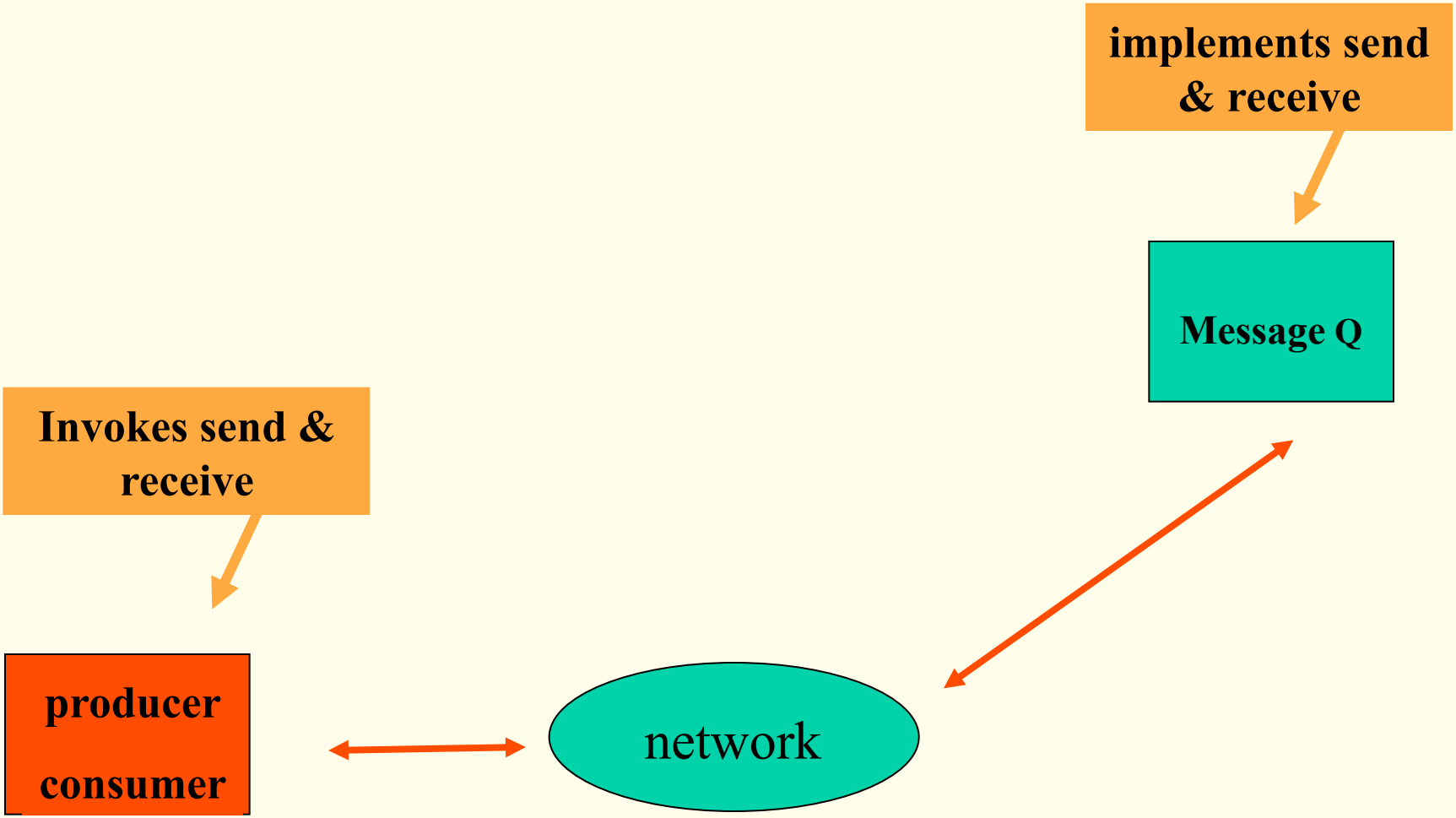
# marshalling parameters



# parameters

- Local (Non-Remote) Objects are Passed-by-Copy using Object Serialization
  - local object must implement interface java.io.Serializable
  - for example, the class *Date*
  - serialization allows the state to be passed as a byte stream
  - object “re-constituted” in receiver as copies
- Remote Objects are Passed by Reference
  - the receiver may then - invoke methods on the remote object
- consider the following consumer-producer problem
  - producers and consumers share a remote queue object Q
  - the remote object Q will be passed by reference to he consumer & producer threads …
  - … which can then invoke the send & receive methods in this object remotely

# MessageQueue example



# Remote Objects

Message Q

- remote objects are declared by specifying an interface that extends `java.rmi.Remote`
- every Method must throw `java.rmi.RemoteException`

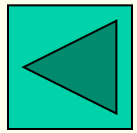
```
public interface MessageQueue extends java.rmi.Remote
{
   public void send(Object item)
        throws java.rmi.RemoteException;

   public Object receive()
        throws java.rmi.RemoteException;
}
```

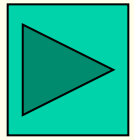
# MessageQueue implementation

Message Q

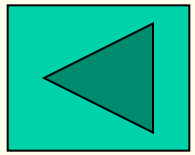
- Class must extend `java.rmi.server.UnicastRemoteObject`
- again, every Method must throw `java.rmi.RemoteException`

```
public class MessageQueueIMPL
   extends java.rmi.server.UnicastRemoteObject
   implements MessageQueue
{
   public void send(Object item)
       throws java.rmi.RemoteException
   { /* implementation */ }
   public Object receive()
       throws java.rmi.RemoteException
   { /* implementation */ }
}
```





```
 // This implements a non-blocking send

public synchronized void send(Object item)
                        throws RemoteException {

   queue.addElement(item);
   System.out.println("Producer entered " + item +
                        " size = " + queue.size());
}
```



```
 // This implements a non-blocking receive

public synchronized Object receive()
                        throws RemoteException {
   Object item;

   if (queue.size() == 0)
       return null;
   else {
       item = queue.firstElement();
       queue.removeElementAt(0);
       System.out.println("Consumer removed "+ item
                           + " size =" + queue.size());
       return item;
   }
}
```

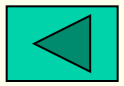
# MessageQueue implementation

Message Q

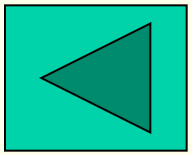
- Object also includes a main() method
- has a security manager to allow access to network files
- notice that the name must be bound in

RMISecurityManager() is deprecated in version 8. Use SecurityManager()

```
public static void main(String args[]){
      System.setSecurityManager(new RMISecurityManager())

  try {
      MessageQueue server = new MessageQueueIMPL();
      Naming.rebind("//127.0.0.1/MessageServer", server);
      System.out.println("Server Bound"); }
  catch(Exception e) {  System.err.println(e); }
}
```



# The Client



RMISecurityManager() is deprecated in version 8. Use SecurityManager()

- the Client Must

(1) install a Security Manager (as done by the server!):

```
System.setSecurityManager(new RMISecurityManager());
```

(2) get a Reference to the Remote Object

```
MessageQueue mb;
mb = (MessageQueue)Naming.lookup(
                    "rmi://127.0.0.1/MessageServer");
```

we know the name !!!



# running the Producer-Consumer using RMI

- Compile all source files
- generate Stub and Skeleton

```
rmic MessageQueueImpl
```

- start the Registry Service

start `rmiregistry`

- create the Remote Object

```
java -Djava.security.policy=java.policy
   MessageQueueImpl
```

- start the Client

```
java -Djava.security.policy=java.policy
   Factory
```

Since version 5.0 of J2SE; support for dynamically generated stub files has been added, and *rmic* is only provided for backwards compatibility with earlier runtimes.

# policy file

- New with Java 2
  - jdk1.2 onwards

```
grant {
     permission java.net.SocketPermission "*:1024-65535",
                "connect,accept,resolve,listen";
};
```