COMP4057
Distributed and Cloud Computing
COMP7940
Cloud Computing

Chapter 03

Communication Paradigms in Distributed Systems

Notes in green: programming related; for COMP 4057 only
Notes in red: newly added materials, for both COMP 4057 and COMP 7940
Optional: for reference only

Learning Outcomes

- Understand the importance of external data representation
- Be able to describe the characteristics of the three major communication paradigms in DS:
  - Interprocess communication
  - Remote invocation
  - Indirect communication
- Give examples of the different communication paradigms, such as
  - MPI
  - Java RMI
  - publish-subscribe systems

Outline

- External data representation
  - Java Serialization
- Interprocess communication
  - MPI
- Remote invocation
  - Java RMI
- Indirect communication
  - Publish-subscribe systems

External Data Representation and Marshalling

- Different systems may store the same information in different ways.
  - Big-endian system vs. little-endian system
    - Big-endian: most significant byte first
    - Little-endian: least significant byte first
  - Different formats for floating-point numbers
    - Size of mantissa and exponent
  - ASCII code vs. Unicode
    - ASCII: 1 byte per character
    - Unicode: 2 bytes per character, can use characters in different languages

- **External data representation** is an agreed standard for the representation of data structures and primitive values.
  - It enables any two systems to exchange binary data values.
  - ***Marshalling***: the process of taking a collection of data items and assembling them into a form suitable for transmission in a message
  - ***Unmarshalling***: the process of disassembling a message to produce an equivalent collection of data items at the destination

Example: Java Object Serialization

- In Java, an object is an instance of a Java class.

```
private Point3D pt1=new Point3D(1,2,3);
// The variable pt1 now holds an instance of
Point3D.
```

- Java serialization can flatten an object or a set of objects into a serial form that is suitable for storing on disk or transmitting in a message.
  - By stating that a class implements the ***Serializable*** interface (provided in ***java.io*** package), its instances can be serialized.
  - E.g., ***public class Employee implements Serializable { … }***
  - ***ObjectOutputStream.writeObject(X)*** is used to serialize an object ***X***
- Java deserialization can restore the state of an object or a set of objects from their serialized form.
  - ***X = ObjectInputStream.readObject()*** is used to deserialize an object and store into ***X***.
- See ObjectRefTest.java for an example to write an object into a file and later read it back.

Interprocess Communication (IPC)

- Interprocess communication provides low-level support for communication between ***processes*** (or ***threads***) in DSs, such as
  - Socket programming
  - Message Passing Interface (MPI)

- Message passing between a pair of processes can be supported by two operations: ***send*** and ***receive***
  - One process sends a message to a destination.
  - Another process at the destination receives the message.
  - A queue is associated with each message destination.
    - A sending process causes a message to be added to the remote queue, and the receiving process removes messages from its local queue.

Characteristics of IPC

- Synchronous and asynchronous
  - In synchronous communication, the sending and receiving processes synchronize at every message.
    - Both send and receive are blocking operations.
    - Whenever a send is issued, the sending process is blocked until the corresponding receive is issued.
    - Whenever a receive is issued by a process, it blocks until a message arrives at its local queue. (No blocking if the local queue is not empty.)
  - In asynchronous communication, the send operation is non-blocking, while the receive operation can be blocking or non-blocking.
    - The sending process can proceed as soon as the message has been copied to a local buffer.
    - The transmission of the message proceeds in parallel while the sending process can do other work.
    - In a system environment such as Java, which supports multiple threads in a single process, the blocking receive has no disadvantage, because other threads may continue to do work.
    - A non-blocking receiving process involves extra complexity due to notification of message arrival.
      - Today's systems do not generally provide the non-blocking form of receive.
- Note that the concept of synchronous / asynchronous communication is different from the concept of synchronous / asynchronous distributed system.

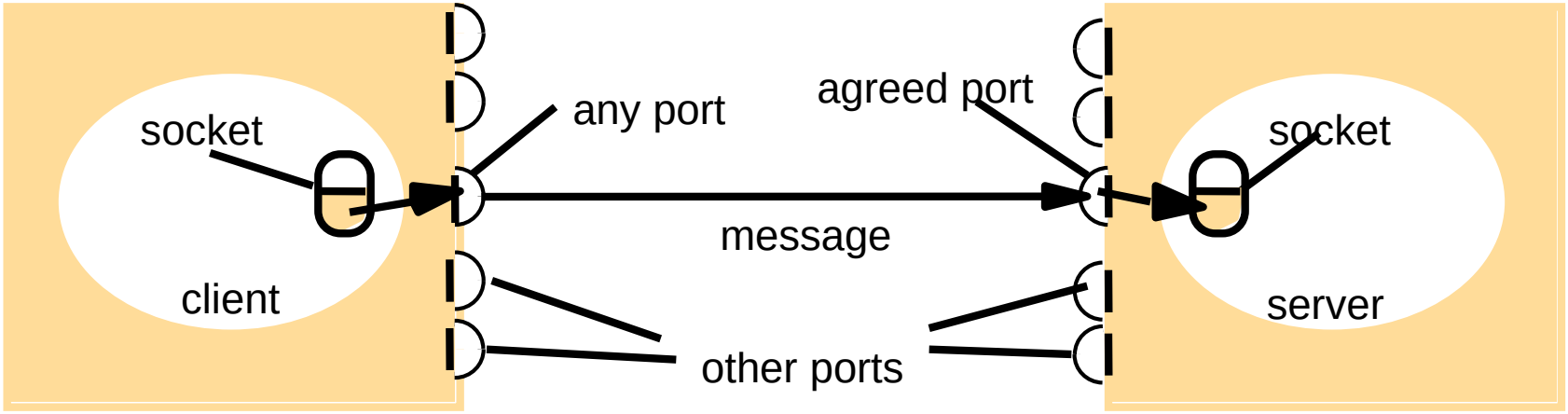
Characteristics of IPC (Cont.)

- Reliability: validity and integrity
  - Validity: a message service is reliable if messages are guaranteed to be delivered, despite some packets being dropped or lost.
  - Integrity: messages must arrive uncorrupted and without duplication.

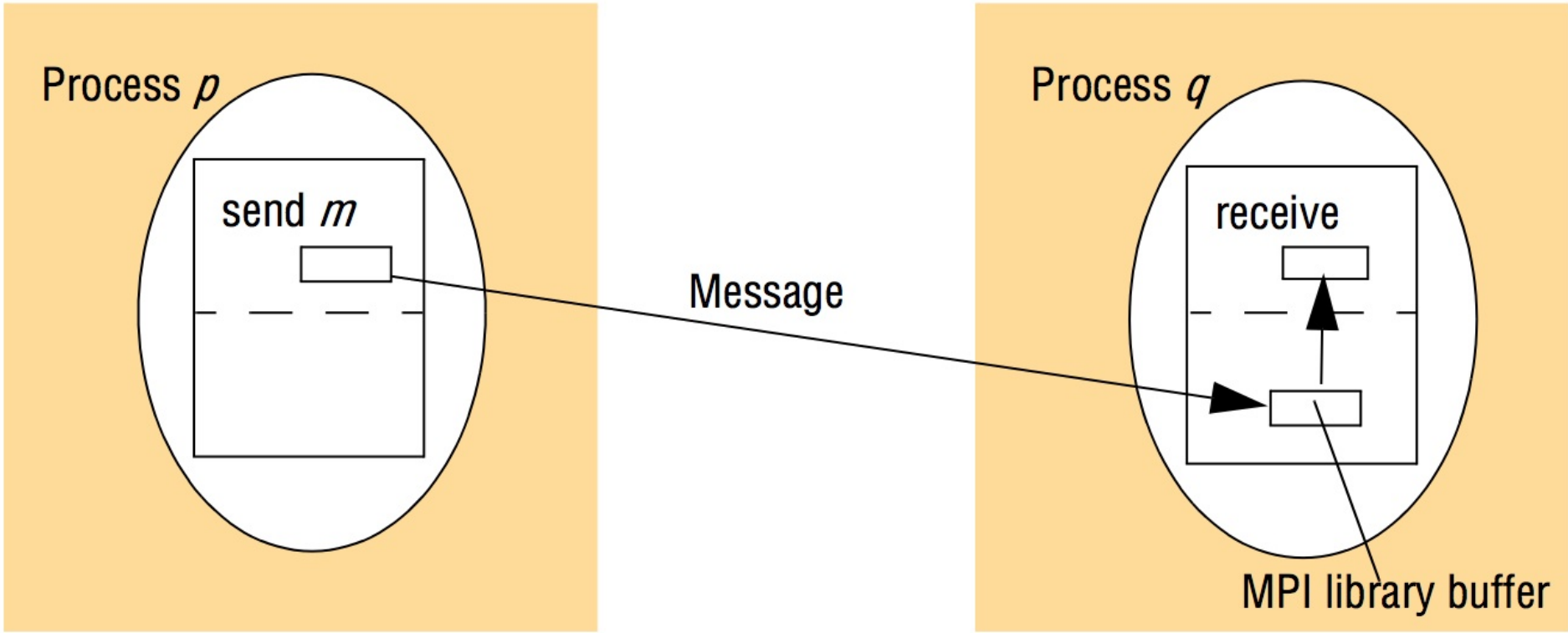
- Ordering: some applications require that messages be delivered in sender's order.
  - E.g., file transfer
  - Some other applications may tolerate occasional packet loss, e.g., real-time voice call.

IPC Example 1: Java Socket

- Java Socket API supports
  - UDP datagrams
  - TCP stream communication
    - See sample UDP and TCP client-server programs.
  - IP multicast
    - See sample multicast socket program.



IPC Example 2: MPI



What is Message Passing Interface (MPI)?

- MPI is a communication protocol for programming parallel computers, e.g., clusters.
  - See description in Wikipedia: https://en.wikipedia.org/wiki/Message_Passing_Interface
- MPI is a widely used standard for writing message-passing programs, especially for high performance computers.
  - http://www.mpi-forum.org
  - It's a specification, not an implementation.
  - Goal: to retain the inherent simplicity, practicality and efficiency of the message passing approach, but to enhance portability through a standardized interface independent of the OS or programming language-specific socket interface.

- It is implemented as a library, not a programming language
  - Examples include MPICH, Open MPI, Microsoft MPI (MS-MPI), Intel MPI
  - MPI has been supported by C, C++, Fortran, Java, Python, etc.

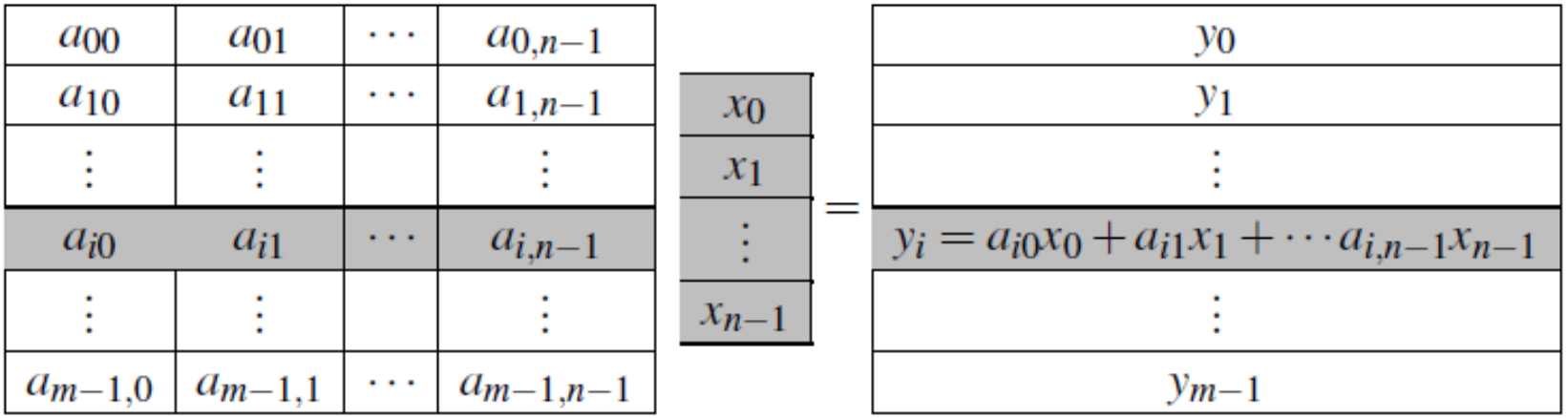
MPI Process and Message Passing

- An MPI program consists of many processes.
  - These processes are executed on a set of physical processors which exchange data (by internal bus or a network).

- The processes executing in parallel have separate address spaces.
  - Assume your program has a statement “y = a + b”.
  - When process A and process B both execute the above statement, each process has its own set of variables {a, b, y}.

- Message-passing: a portion of one process’s address space can be copied into another process’s address space.
  - “message” means “data”
  - “message-passing” means “data transfer”
  - It’s usually done by ***send*** operation and ***receive*** operation.

Example: Matrix-Vector Multiplication

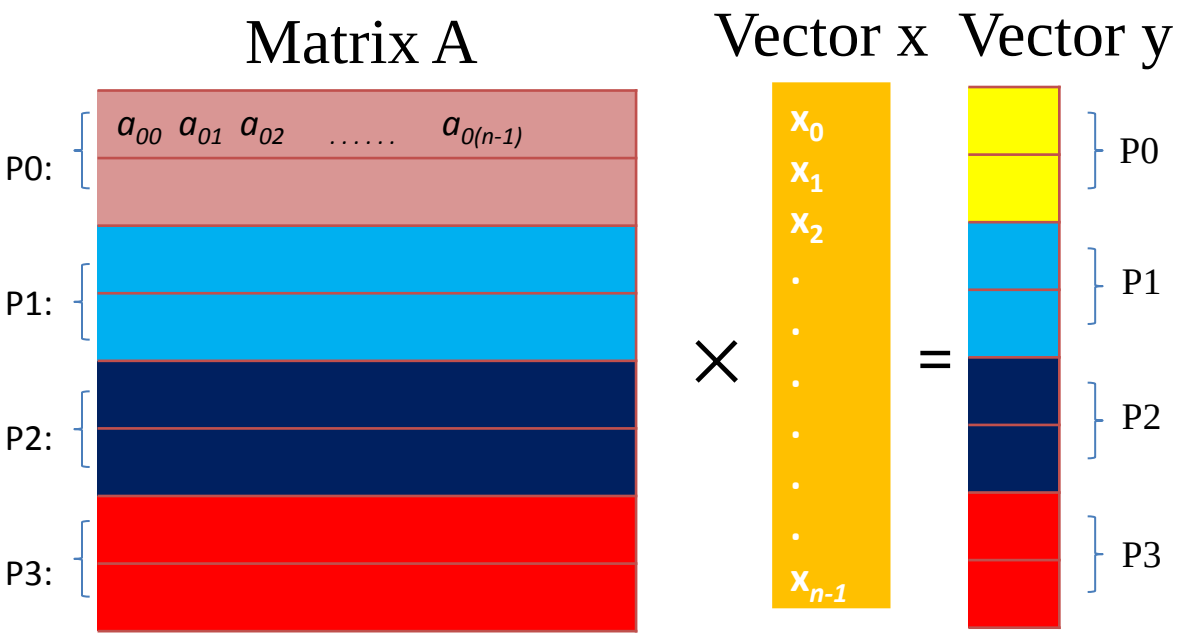



```
/* For each row of A */
for (i = 0; i < m; i++) {
    /* Form dot product of ith row with x */
    y[i] = 0.0;

    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Rowwise 1-D Partitioning

- Given *p* processes, Matrix A (*m* x *n*) is partitioned into *p* smaller matrices, each with dimension (*m*/*p* x *n*).
  - For simplicity, we assume *p* divides *m* (or, *m* is divisible by *p*).



Matrix-Vector Multiplication by MPI

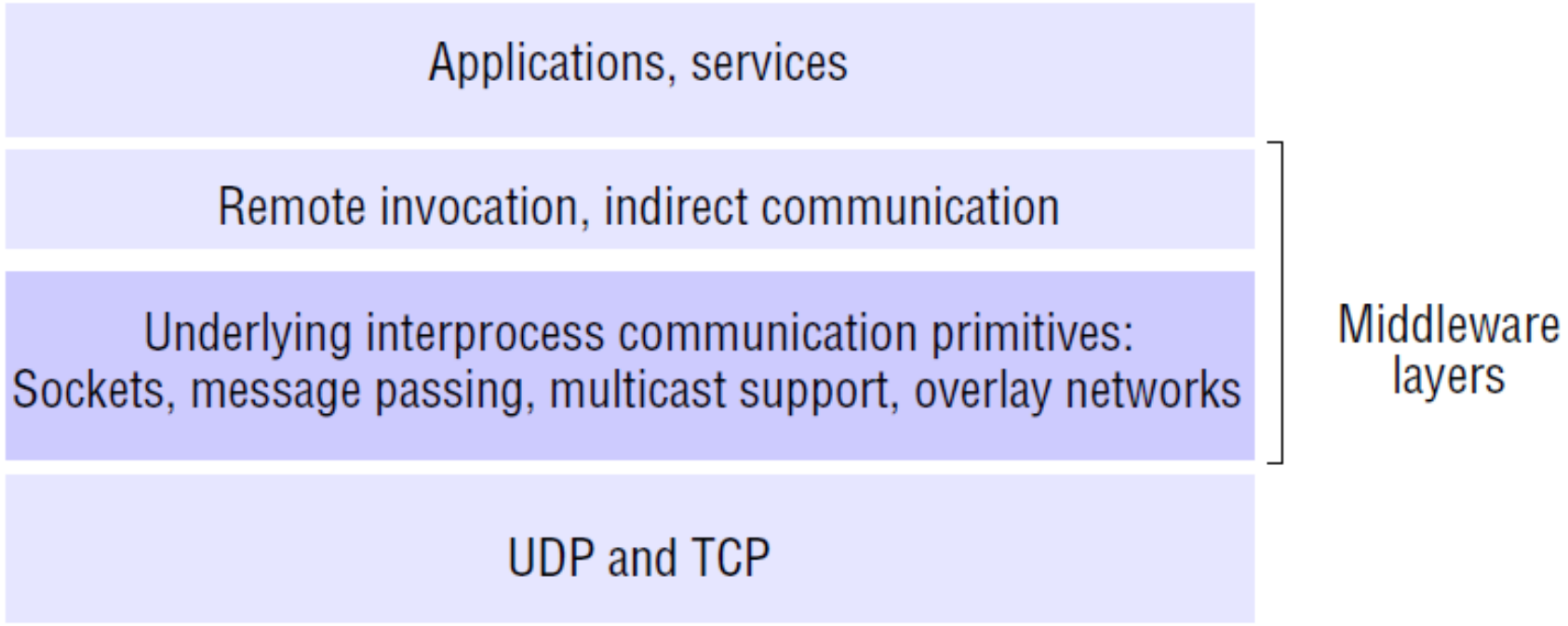
- Assumptions
  - A total of *p* processes
  - Matrix *A* (*m* x *n*) and vector *x* (*n* x 1) are created at **process *0***
    - called “master process” because it coordinates the work of other processes (i.e., “slave processes”)

1. Message passing:
   - Process *0* will send (*p*-1) sub-matrices to corresponding processes
   - Process *0* will send vector *x* to all other *p*-1 processes
2. Calculations:
   - Each process carries out its own matrix-vector multiplication
3. Message passing:
   - Processes 1 to (*p*-1) send the results (i.e., part of vector *y*) back to process *0*

Remote Invocation

- Remote invocation is the most common communication paradigm in DSs.
  - Request-reply protocols: a pattern of two-way message exchange on top of message passing
    - Such as **HTTP**
  - Remote procedure call (RPC): a client program can call procedures in a server program transparently.
    - Such as **Sun RPC**
    - Not further discussed
  - Remote method invocation (RMI): extends the concept of RPC to object-oriented programming model.
    - Such as **Java RMI**

Middleware Layers



Request-reply Protocols

- Based on message passing (send and receive)
- Normally via synchronous communication – client blocks until reply arrives from server.
- The client makes a request. The server (may) perform some computation and (may) return a reply.
- Styles of exchange protocols
  - Request (R) protocol
    - Example usage: a sensor reports the temperature.
  - Request-reply (RR) protocol
    - Example usage: a browser requests a web page from a server.
  - Request-reply-acknowledge (RRA) protocol
    - Acknowledgment allows server to discard history record.

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Request-reply Protocols (cont.)

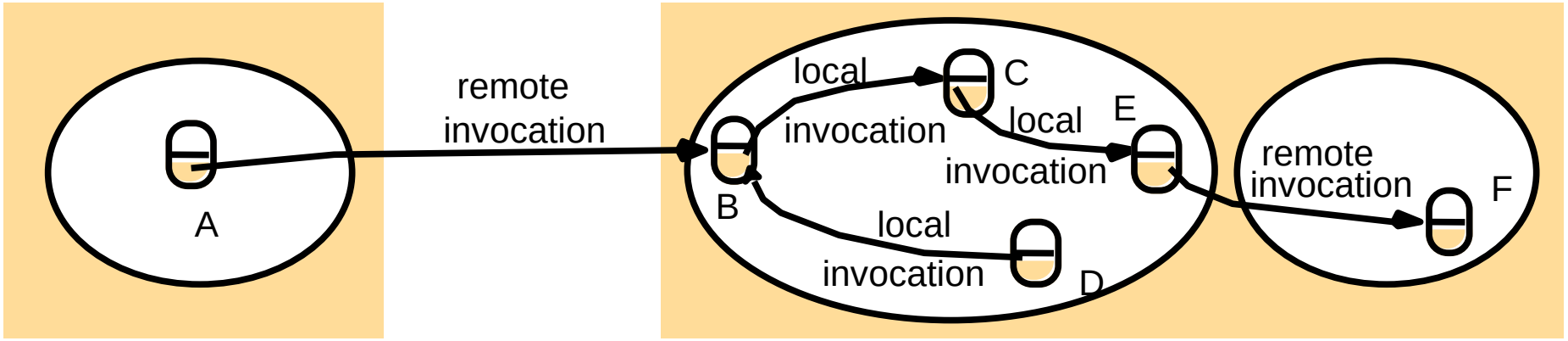
- Failure models of Request-reply Protocols
  - Omission failure – message (request or reply) lost in transit
  - Messages not guaranteed to be delivered in sender's order
  - Duplicated message possible
- Each message has a unique requestID.
  - Client includes requestID in request message.
  - Server uses requestID to discard duplicate request messages.
  - RequestID included in server's reply for client to ensure a correct reply of previous request.
- Timeouts
- History
  - In RRA, if the server does not get an acknowledgment, the server retransmits the reply.
  - The term 'History' refers to a structure that contains records of replies that have been transmitted.
  - A record is deleted only when the acknowledgment is received.

Traditional Object Model

- An object encapsulates a set of data and a set of methods.
- An object-oriented program consists of a collection of interacting objects.
- An object communicates with other objects by invoking their methods, passing arguments and receiving results.
- Interfaces: an interface provides a definition of the signatures of a set of methods without specifying their implementations.
  - We say “an object provides a particular interface” if its class contains code that implements the methods of that interface.
- Actions: an action is initiated by an object invoking a method in another object.
  - As an invocation can lead to further invocations of methods in other objects, an action can be a chain of related invocations.

Distributed Objects

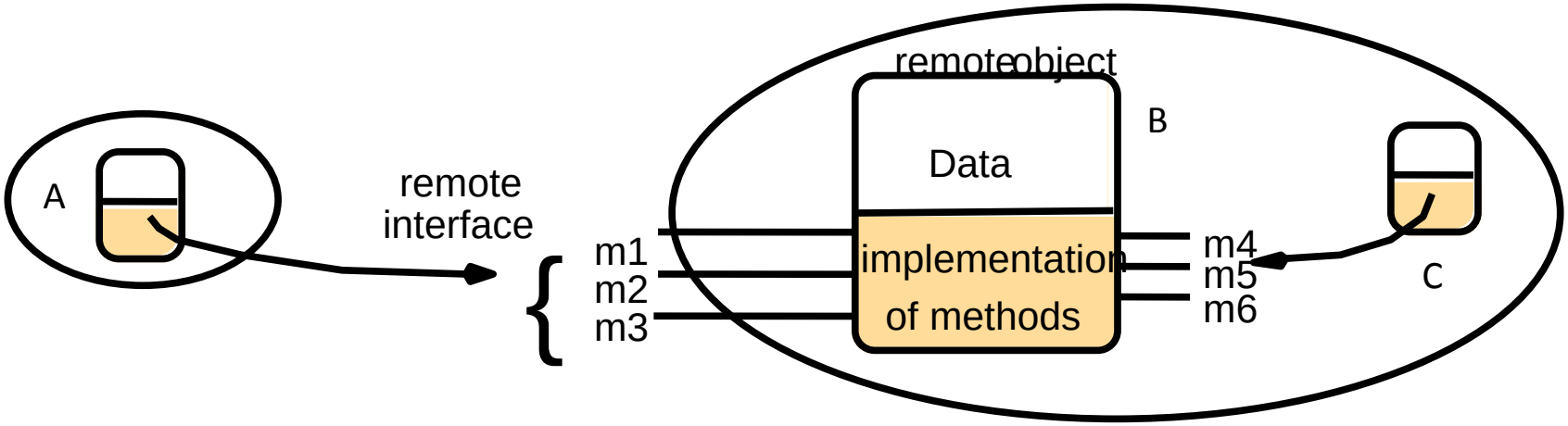
- In distributed systems, objects can be physically distributed into different processes or computers.
- Remote objects: objects that can receive remote invocations.
  - An object in one computer may be invoked by two or more remote objects at the same time: concurrency and synchronization issues.
  - Some objects can receive both local and remote invocations, whereas some other objects can receive only local invocations.
    - Objects B and F below are remote objects.



REF1 Fig. 5.12: Remote and local invocations

Distributed Objects (Cont.)

- Two fundamental concepts
  - **Remote object references**: an identifier that can be used to refer to a remote object. Object references can be passed as parameters and returned as results.
  - **Remote interfaces**: every remote object has a remote interface that specifies which of its methods can be invoked remotely.
    - In the figure below, object A can remotely invoke m1, m2, and m3.
    - Local object C can invoke m1, m2, m3, m4, m5, and m6.



REF1 Fig. 5.13: A remote object and its remote interface

Remote Object Reference

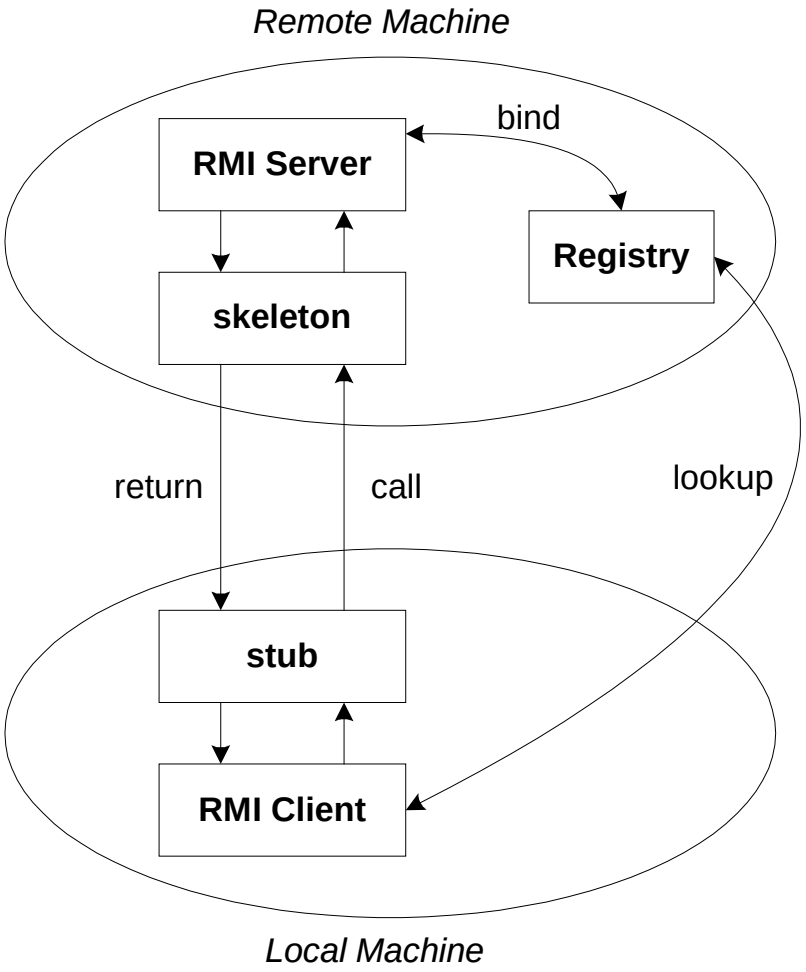
Representation of a remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

- The internet address and port number uniquely identifies a remote application.
- The creation time helps to identify the process.
  - Removes the ambiguity that an application is closed and rerun again.

The General RMI Architecture

- The server must first bind its name to the registry.
- The client looks up the server name in the registry to establish remote references.
- The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.



Client: How to Obtain Remote Object References?

- Binder service:
  - The system runs a binder service that maps a textual name to an object reference.
    - E.g., type "**rmiregistry &**" in your Linux terminal to start the binder service.
  - The server assigns a URL-style "name" to each remote object, and makes a registration.
    - Name format: //computerName:port/objectName
  - The client can obtain a remote object reference by looking up the name of the remote object.

RMI Example 1: HelloWorld

- A simple RMI "Hello World!" program is given in:
  - https://www.cs.ucsb.edu/~cappello/lectures/rmi/helloworld.shtml
- Download the files Hello.java, HelloImpl.java, HelloClient.java, security.policy to a subdirectory in cslinux1.
  - Hello.java contains the interface Hello.
  - HelloImpl.java is the server program.
  - HelloClient.java is the client program.
  - security.policy contains the security.policy of the server.
- In line 1 of security.policy, change "/home/compstaff/tam/comp4057/rmihelloworld/" to your subdirectory name.
- Compile the java files by

  javac Hello.java HelloImpl.java HelloClient.java
- In your subdirectory in cslinux1, run the RMI registry in the background by:

  rmiregistry &
- Now run the HelloWorld server in cslinux1 by:

  java HelloImpl
- Run the client in cslinux2 by:

  java –Djava.security.policy=security.policy HelloClient
- This displays the String "Hello World!", which comes from running the sayHello() method in the HelloImpl remote object in the server in cslinux1.

RMI Example 2: Whiteboard (optional)

- Java RMI extends the Java object model to provide support for distributed objects.
- It allows objects to invoke methods on remote objects using the same syntax as for local invocations.
- Example application: shared whiteboard
  - A group of users share a common view of a drawing surface containing graphical objects (circles, rectangles, lines, etc.)
    - ***GraphicalObject*** is the class that holds the state of a graphical object.
  - Users can draw graphical objects on the shared whiteboard from different machines.
  - The server maintains the current state of a drawing and provides users the latest shapes drawn by other users.
    - Each graphical object is identified by a unique integer called "***version***".
    - ***Java.util.Vector*** is used to manage the set of graphical objects.

GraphicalObject must implement *Serializable* interface (optional)

```
import java.awt.Rectangle; import java.awt.Color; import java.io.Serializable;

public class GraphicalObject implements Serializable{
   public String type;   public Rectangle enclosing;   public Color line;   public Color fill;  public boolean isFilled;

   //  constructors
   public GraphicalObject() { }

   public GraphicalObject(String aType, Rectangle anEnclosing, Color aLine,Color aFill, boolean anIsFilled) {
            type = aType;
            enclosing = anEnclosing;
            line = aLine;
            fill = aFill;
            isFilled = anIsFilled;
   }

   public void print(){
            System.out.print(type);
            System.out.print(enclosing.x + " , " + enclosing.y + " , " + enclosing.width + " , "  + enclosing.height);
            if(isFilled) System.out.println("- filled");else System.out.println("not filled");
   }
}
```

A Typical Java RMI Application (optional)

- A typical Java RMI application may include the following programs:
  - Interface program: define the remote interface
  - Servant program: define and implement the remote object
  - Server program: maintain the remote objects
  - Client program: look up remote object references and make remote invocations

Example Code of Remote Interfaces in Java RMI (optional)

```
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;
}

public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Remote Interfaces in Java RMI (optional)

- Remote interfaces are defined by extending an interface called ***Remote*** in ***java.rmi*** package.

- The example defines two remote interfaces:
  - ***Shape***:
    - getVersion(): returns the version of a graphical object
    - getAllState(): returns an instance of the class GraphicalObject
  - ***ShapeList***:
    - newShape(): passes an instance of GraphicalObject and returns a remote object
    - allShapes(): returns the vector of instances of GraphicalObject
    - getVersion(): returns the number of graphical objects

ShapeServant: Implement the *Shape* Interface (optional)

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeServant implements Shape {
    int myVersion;
    GraphicalObject theG;
    public ShapeServant(GraphicalObject g, int version)throws RemoteException{
        theG = g;
        myVersion = version;
    }
    public int getVersion() throws RemoteException {
         return myVersion;
    }
    public GraphicalObject  getAllState() throws RemoteException{
        return theG;
   }
}
```

ShapeListServant: Implement the *ShapeList* Interface (optional)

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant implements ShapeList {
      private Vector theList;                    // contains the list of Shapes
      private int version; // the number of Shapes
     public ShapeListServant() throws RemoteException {
         theList = new Vector();
         version = 0;
     }
     public Shape newShape(GraphicalObject g) throws RemoteException {
           version++;
           Shape s = new ShapeServant( g, version);
           theList.addElement(s);
           return s;
     }
     public  Vector allShapes()throws RemoteException { return theList; }
     public int getVersion() throws RemoteException { return version; }
}
```

Example of Server Program (optional)

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();
            ShapeList stub = (ShapeList)
                UnicastRemoteObject.exportObject(aShapeList,0);

            Naming.rebind("ShapeList", aShapeList );
            System.out.println("ShapeList server ready");
        } catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());
        }
    }
}
```

1. *exportObject()* makes the *aShapeList* object available to receive incoming invocations.
2. *rebind()* binds the *aShapeList* object to name "ShapeList" in the registry.

Example of Client Program (optional)

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

public class ShapeListClient{
    public static void main(String args[]){
      System.setSecurityManager(new RMISecurityManager());
      ShapeList aShapeList = null;
      try{
            aShapeList  = (ShapeList) Naming.lookup("//xxx.xxx.xxx/ShapeList");
            Vector sList = aShapeList.allShapes();
      } catch(RemoteException e) {System.out.println(e.getMessage());
      }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

A more complete code can be found at:
https://github.com/alvinhkh/rmi-whiteboard

Indirect Communication

- Both IPC and RMI are based on direct communication.
  - Direct coupling between the sender and receiver.
  - It may not well handle the scenario that clients or servers are temporally disconnected from the distributed system.

- Indirect communication: communication between entities in a DS through an intermediary with no direct coupling between the sender and the receiver(s). Two key properties are:
  - Space uncoupling: the sender does not know or need to know the identity of the receiver(s).
  - Time uncoupling: the send and receiver(s) can have independent lifetimes.

Indirect Communication

Figure 6.1 Space and time coupling in distributed systems

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: One-to-one email
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter Publish-subscribe systems

Pros and Cons of Indirect Communication

- Advantages:
  - With space uncoupling, system developer has more degree of freedom in dealing with system changes such as failure, replacement, upgrade, replication, or migration of system participants (senders or receivers).
  - With time uncoupling, sender and receiver(s) do not need to exist at the same time to communicate. This is good for volatile environments where senders and receivers may come and go.
    - Example: mobile environments
- Disadvantages:
  - Performance overhead by the added level of indirection
  - The system becomes more difficult to manage precisely.

Publish-subscribe Systems

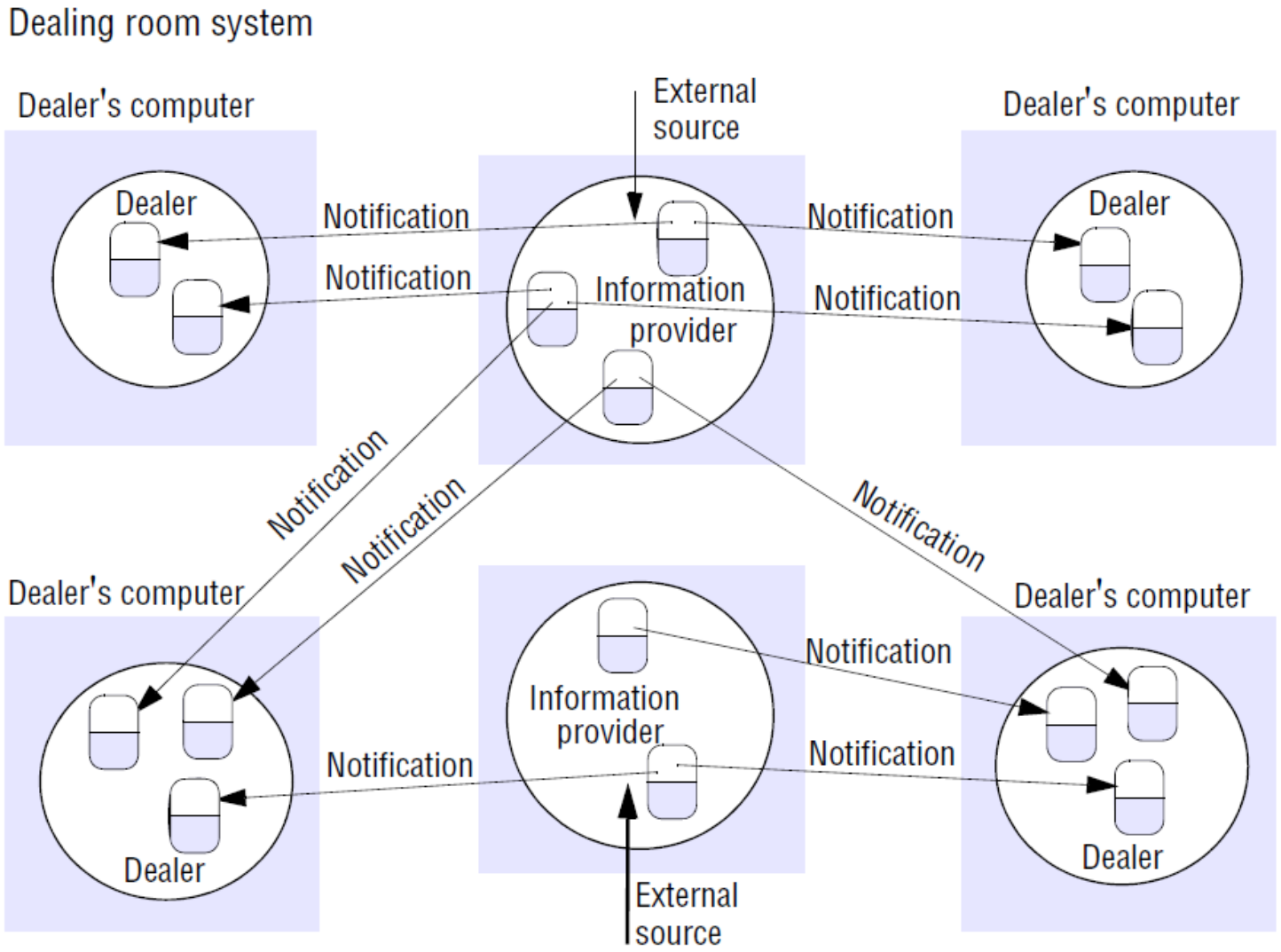
- A widely used indirect communication technique
  - A one-to-many indirect communication paradigm
  - Also called distributed event-based system
- ***Publishers*** publish structured events to an event service.
- ***Subscribers*** express interest in particular events through subscriptions.
- The publish-subscribe system matches subscriptions against published events and ensure the correct delivery of event notifications.
- Applications include:
  - Financial information systems
  - Areas with live feeds of real-time data
  - Support for cooperative working, where a number of participants need to be informed of events of shared interest
  - Support for ubiquitous computing, e.g., location events
  - Monitoring applications such as network monitoring, oil pipe monitoring, etc.

Example: A Dealing Room System

- A dealing room system: to allow dealers see the latest information about the market prices of the stocks they deal in.
  - The market price for a single stock is represented by an associated object.
  - Market prices come from many information providers in the form of updates to objects representing stocks.
  - A dealer is only interested in his own specialist stocks.

- The system can be implemented by two types of processes:
  - Information provider process
  - Dealer process

Example:
A Dealing Room System



Example: A Dealing Room System

- Information provider process
  - Continuously receives new trading information from a single external source.
    - Examples of sources: Reuter, Bloomberg, CNN Business News.
  - Each update is regarded as an event.
  - Publishes such events to the publish-subscribe system for delivery to all dealers who have expressed interest in the corresponding stock.

- Dealer process
  - Creates a subscription representing each named stock that the dealer is interested in.
  - Each subscription expresses an interest in events related to a stock.
  - Receives all the information sent to it and displays it to the dealer.

Characteristics of Publish-Subscribe Systems

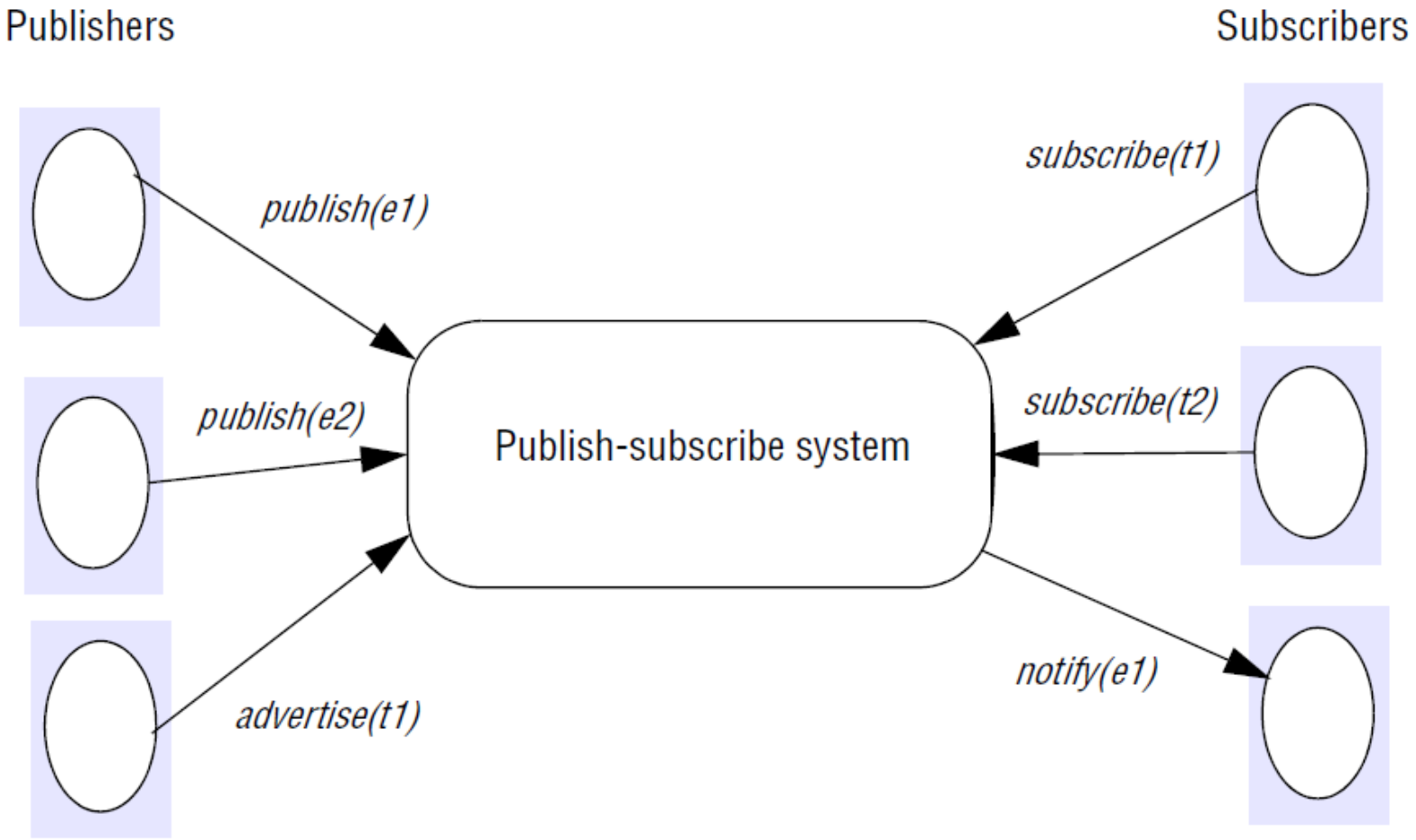
- Heterogeneity:
  - Heterogeneous entities in a distributed system can work together with event notifications as a means of communication.
- Asynchronicity:
  - Notifications are sent asynchronously by event-generating publishers to all subscribers.
- Applications may choose different delivery guarantees for notifications. Examples:
  - To deliver the latest state of a player in an Internet game, notifications need not be guaranteed to arrive, because the next update is likely to get through.
  - For the dealing room application, to be fair to all dealers, guarantee of notification arrival is needed.

Programming Model of Publish-Subscribe Systems

- Publishers disseminate an event ***e*** through a ***publish(e)*** operation.
- Subscribers express an interest in a set of events through ***subscribe(f)*** operation, where ***f*** is a filter, i.e., a pattern that expresses the subscriber's interest.
  - ***unsubscribe(f)*** is used to revoke this interest.
- When events arrive at a subscriber, the events are delivered using ***notify(e)*** operation.
- Some systems introduce the concept of advertisements.
  - Publishers have the option of declaring the nature of future events through an ***advertise(f)*** operation.
  - ***f*** takes the same form as filters.
  - Advertisements can be revoked by calling ***unadvertise(f)***.

Programming Model of Publish-Subscribe Systems

The publish-subscribe paradigm



Subscription Model of Publish-Subscribe Systems

- How to express a subscriber's interest (filter models)?
  - **Channel-based**: publishers publish events to named channels, and subscribers subscribe to one of the named channels to receive all events in that channel.
    - E.g., news from Bloomberg, Reuter, Yahoo Finance
  - **Topic-based** (or subject-based): each notification includes a field that denotes the "topic"; and subscriptions are defined by the "topic" of interest. A "topic" can be hierarchical, such as "sport/basketball/NBA", "sport/soccer/PremierLeague".
  - **Content-based**: each notification has many attributes; a content-based filter is a query defined in terms of compositions of constraints over the values of event attributes.
    - E.g., "stock=AAPL" and "price<100"
  - **Type-based**: linked with object-based approaches where objects have a specified type. Subscriptions are defined in terms of types of events and matching is defined in terms of types or subtypes of the given filter.
    - E.g., news of "Apple Inc."

More Reading Materials

- MPI:
  - https://computing.llnl.gov/tutorials/mpi/
- Java RMI:
  - https://docs.oracle.com/javase/tutorial/rmi/
- Publish-Subscribe System:
  - Y. Liu and B. Plate, “Survey of Publish Subscribe Event Systems,” TR574, Indiana University.
  - ftp://www.cs.indiana.edu/pub/techreports/TR574.pdf