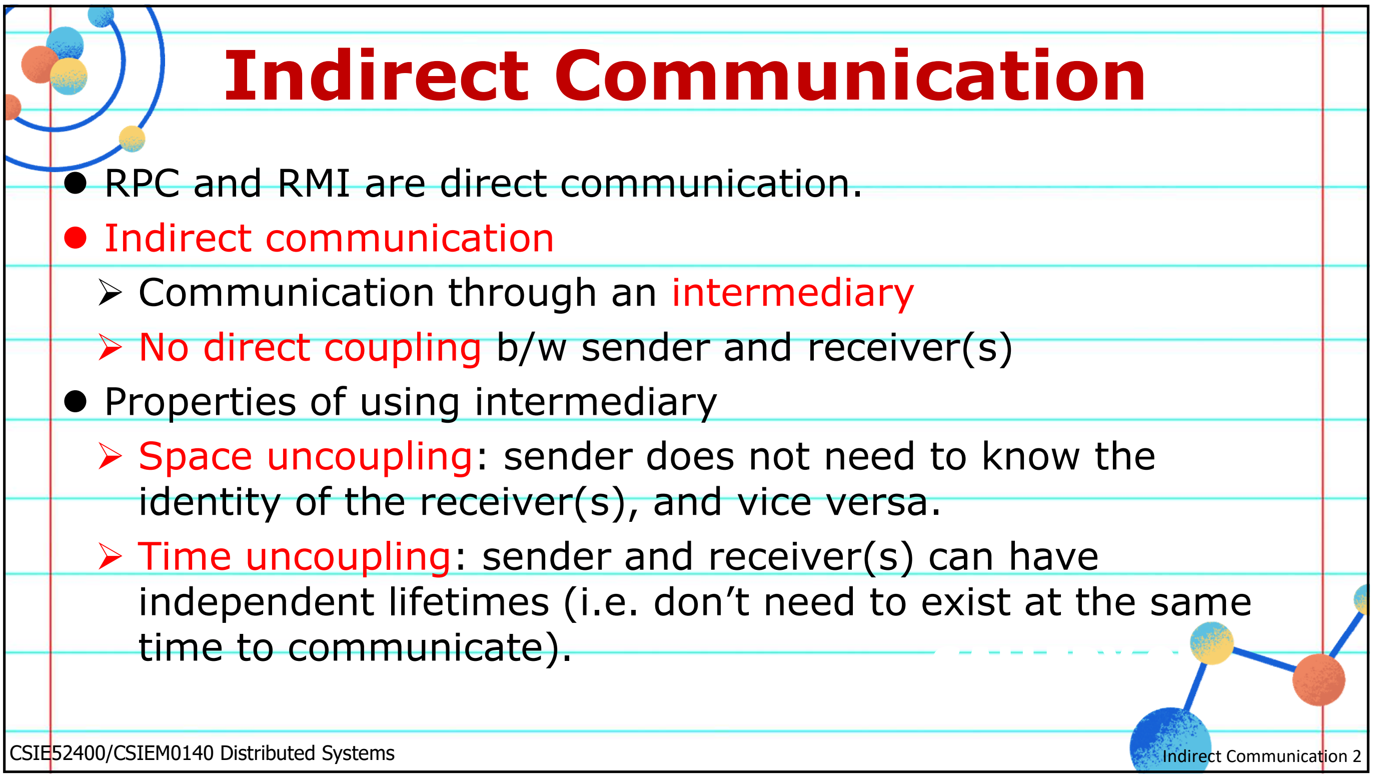
CSIE52400/CSIEM0140 Distributed Systems

Lecture 08: Indirect Communication

Shiow-yang Wu (吳秀陽)

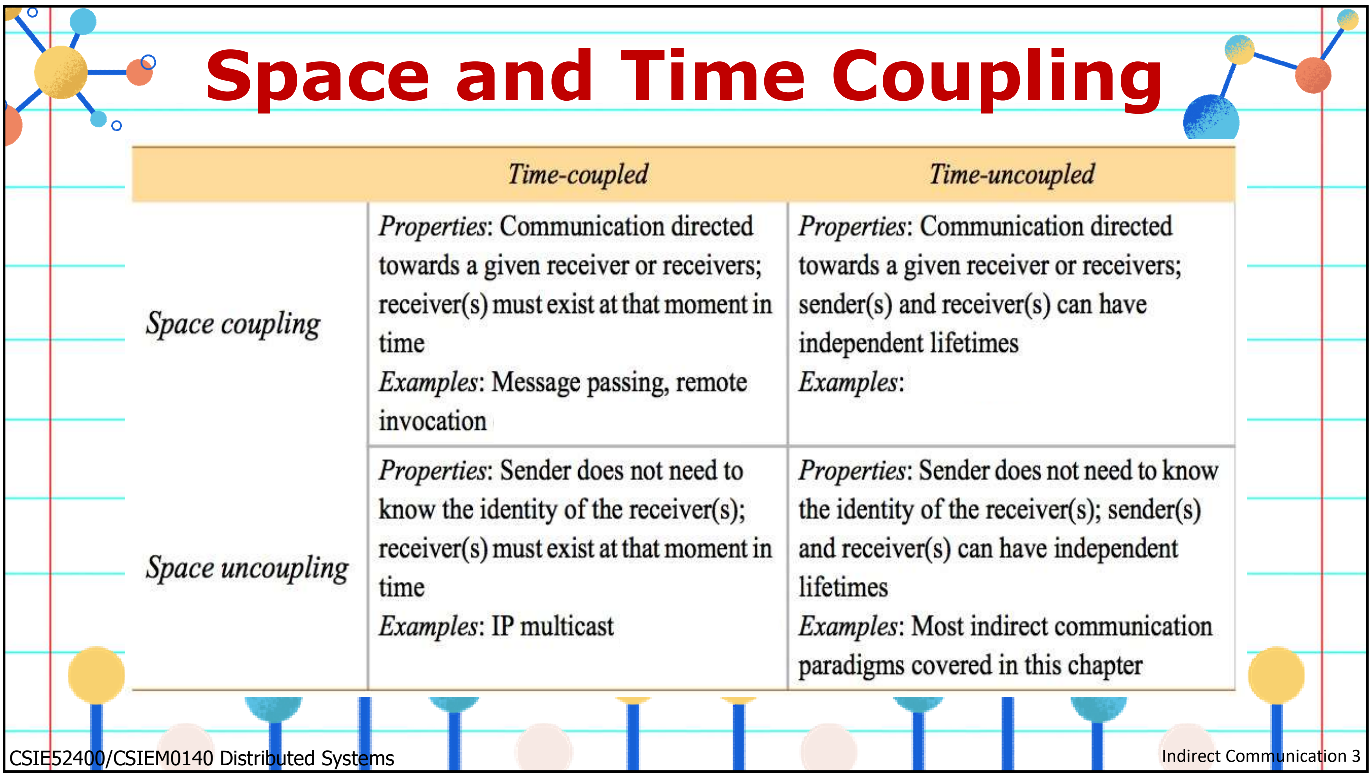
Department of Computer Science and Information Engineering

National Dong Hwa University



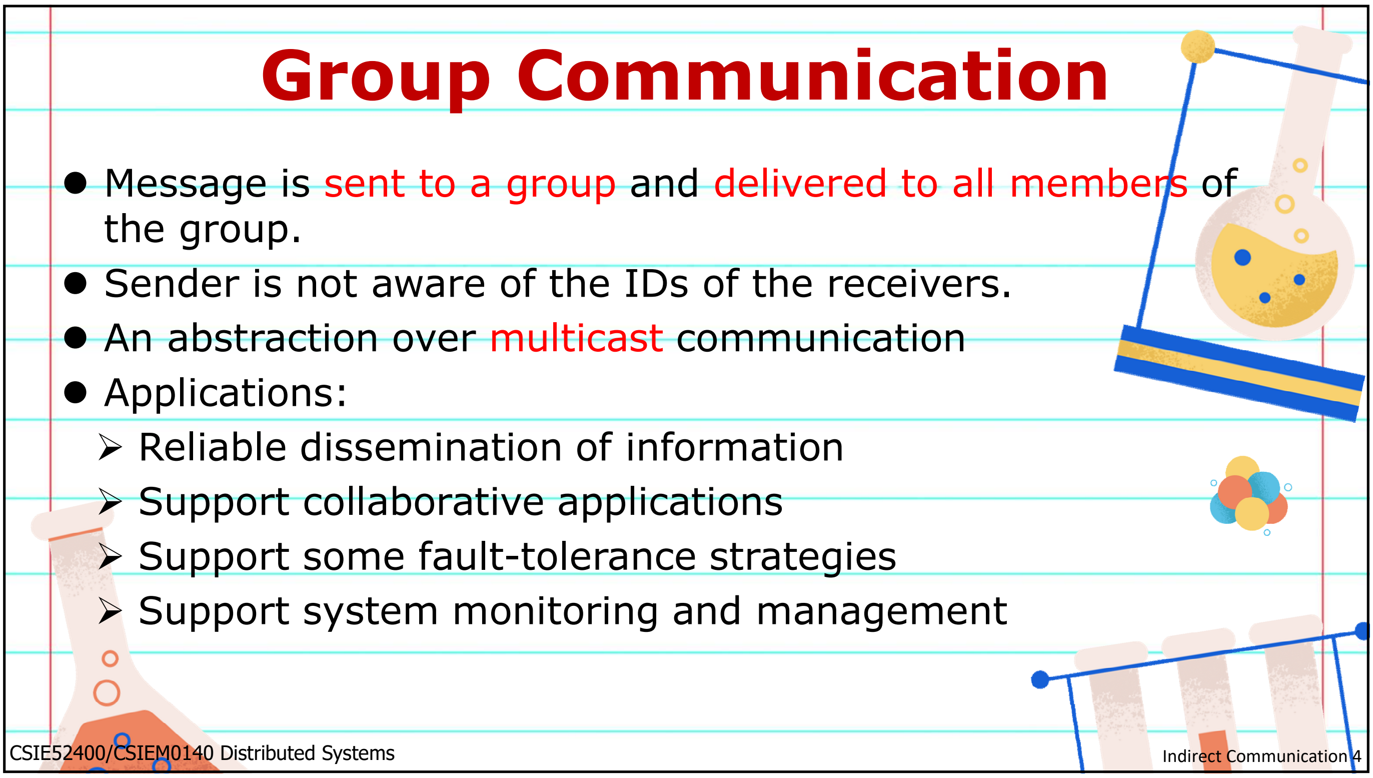
Indirect Communication

- RPC and RMI are direct communication.
- Indirect communication
  - Communication through an intermediary
  - No direct coupling b/w sender and receiver(s)
- Properties of using intermediary
  - Space uncoupling: sender does not need to know the identity of the receiver(s), and vice versa.
  - Time uncoupling: sender and receiver(s) can have independent lifetimes (i.e. don’t need to exist at the same time to communicate).



Space and Time Coupling

	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	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*:
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	*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

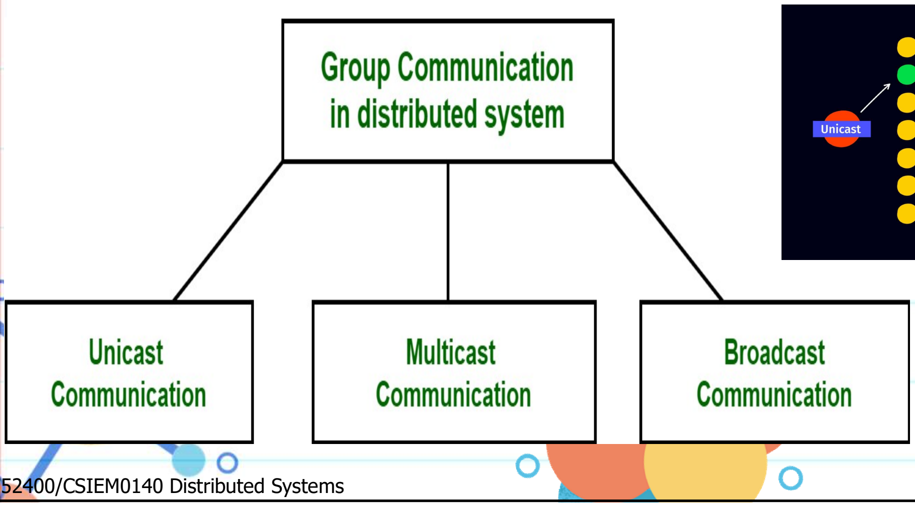


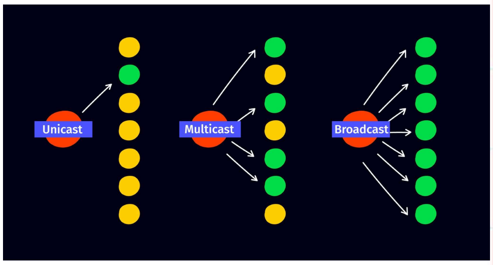
Group Communication

- Message is sent to a group and delivered to all members of the group.
- Sender is not aware of the IDs of the receivers.
- An abstraction over multicast communication
- Applications:
  - Reliable dissemination of information
  - Support collaborative applications
  - Support some fault-tolerance strategies
  - Support system monitoring and management

Types of Group Communication

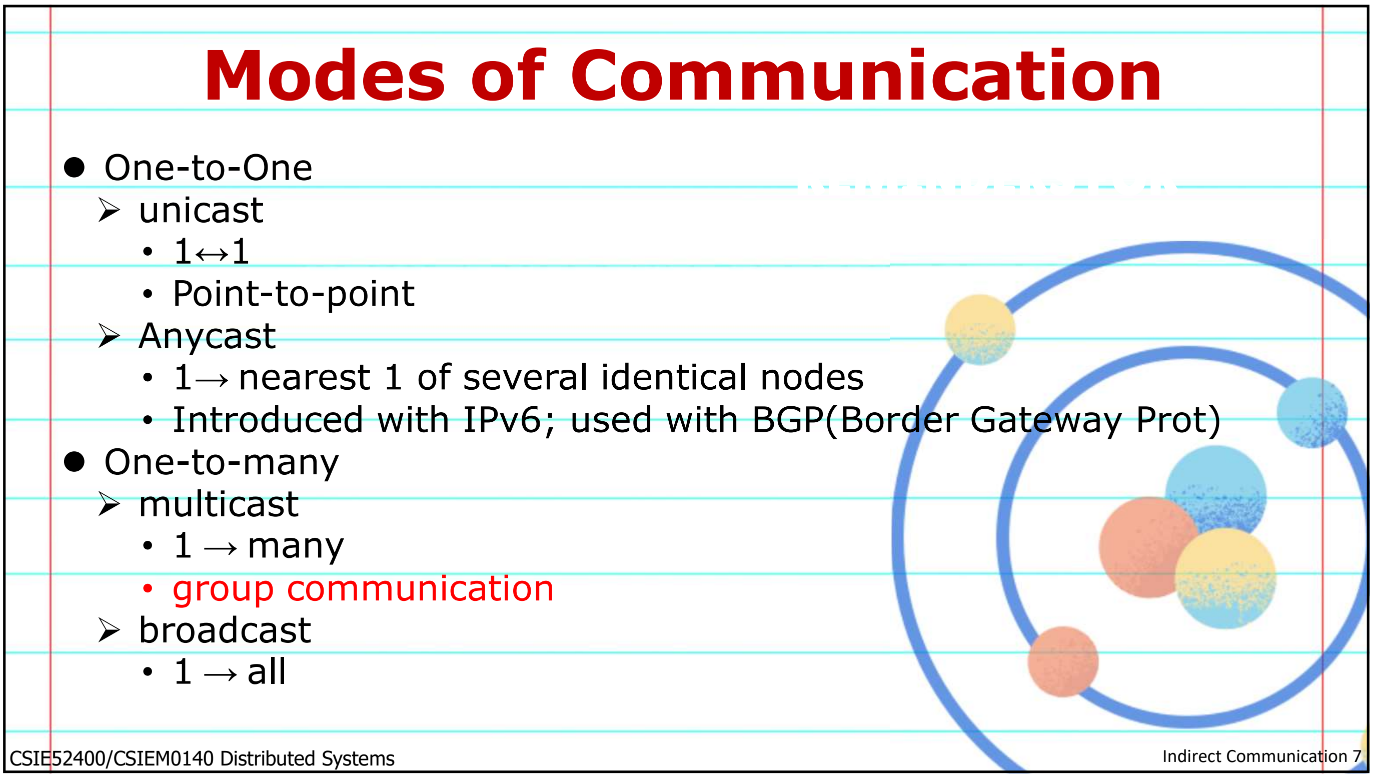
- In general, three types of communication can all be considered as group communication.





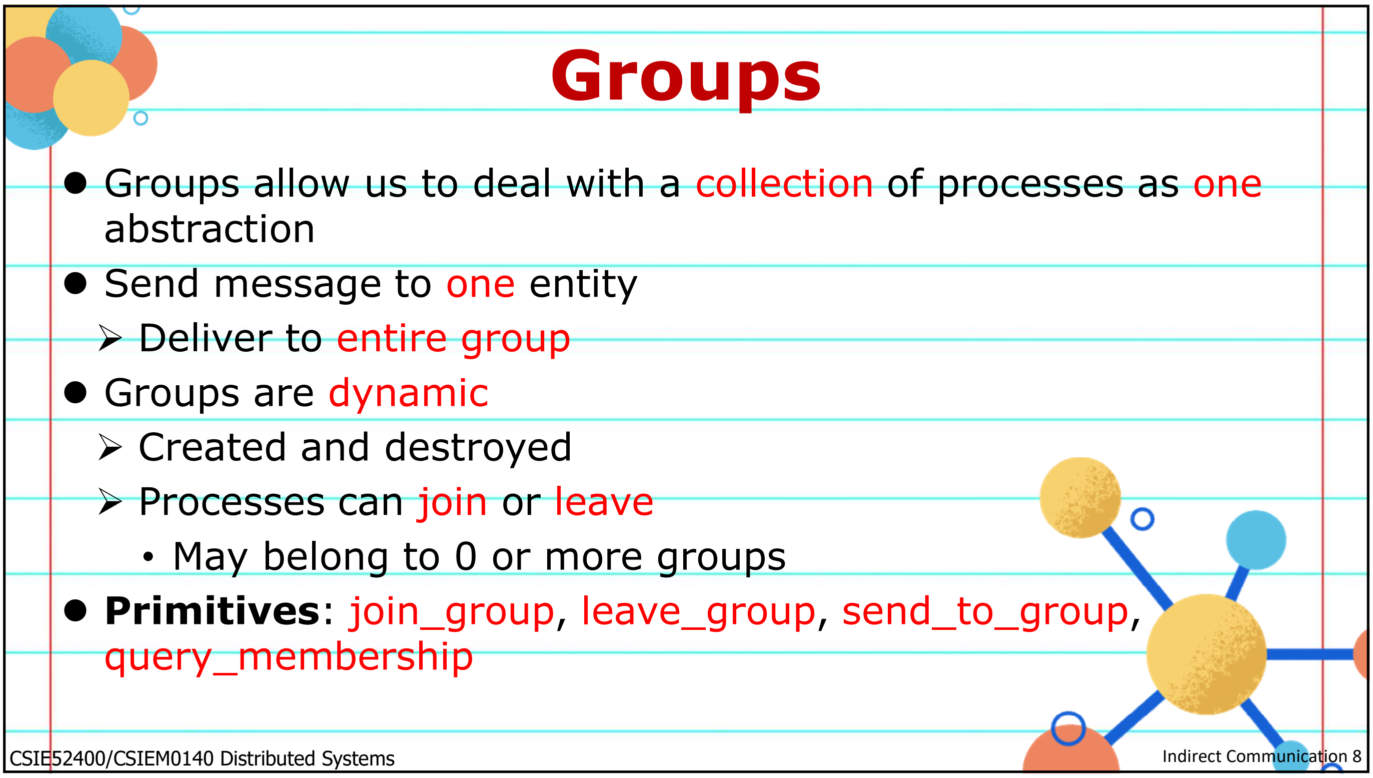
Group Communication System

- Services provided by group comm systems:
  - ➢ Abstraction of a Group
  - ➢ Multicast of messages to a Group
  - ➢ Membership of a Group
  - ➢ Reliable messaging
  - ➢ Ordering of messages sent to a Group
  - ➢ Failure detection of members of the Group
  - ➢ Semantic model of how messages are handled when changes to the Group membership occur



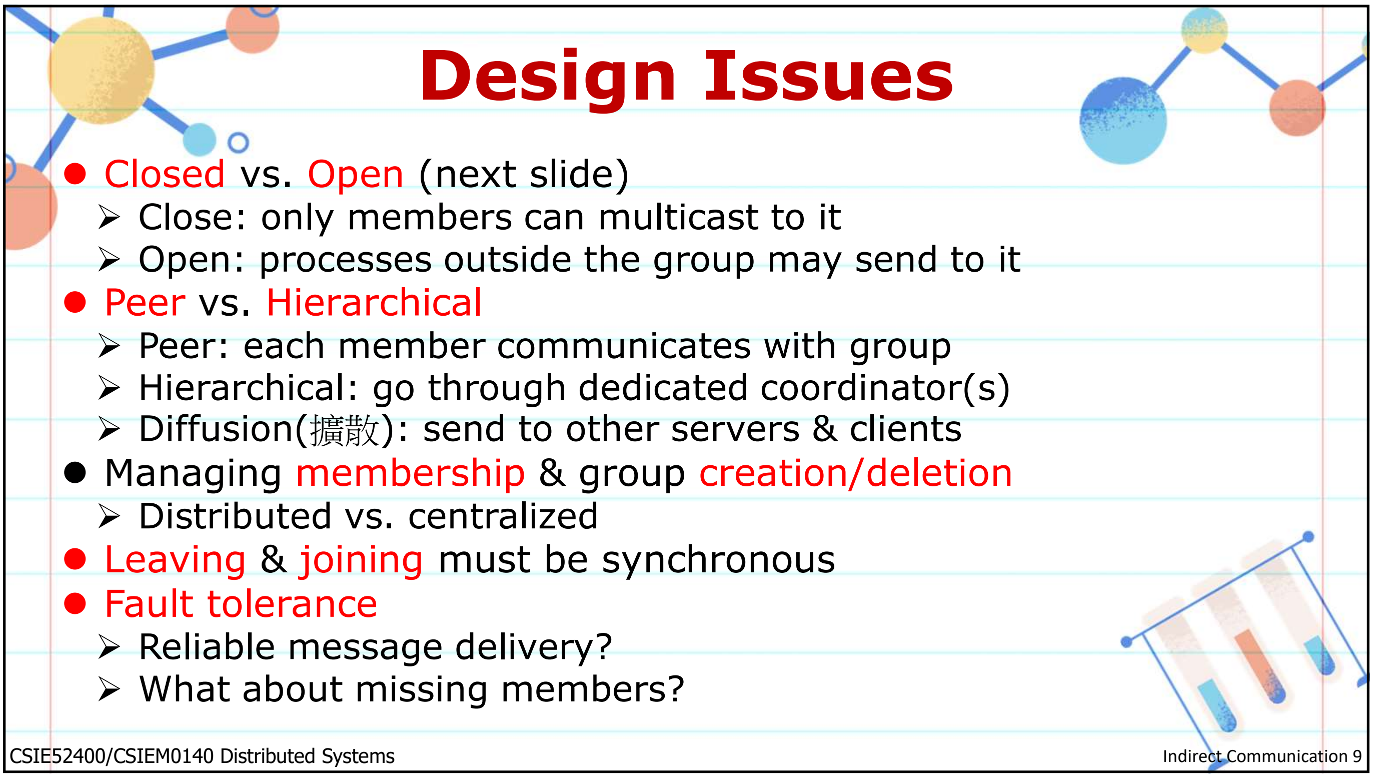
Modes of Communication

- One-to-One
  - ➢ unicast
    - 1↔1
    - Point-to-point
  - ➢ Anycast
    - 1→ nearest 1 of several identical nodes
    - Introduced with IPv6; used with BGP(Border Gateway Prot)
- One-to-many
  - ➢ multicast
    - 1 → many
    - group communication
  - ➢ broadcast
    - 1 → all



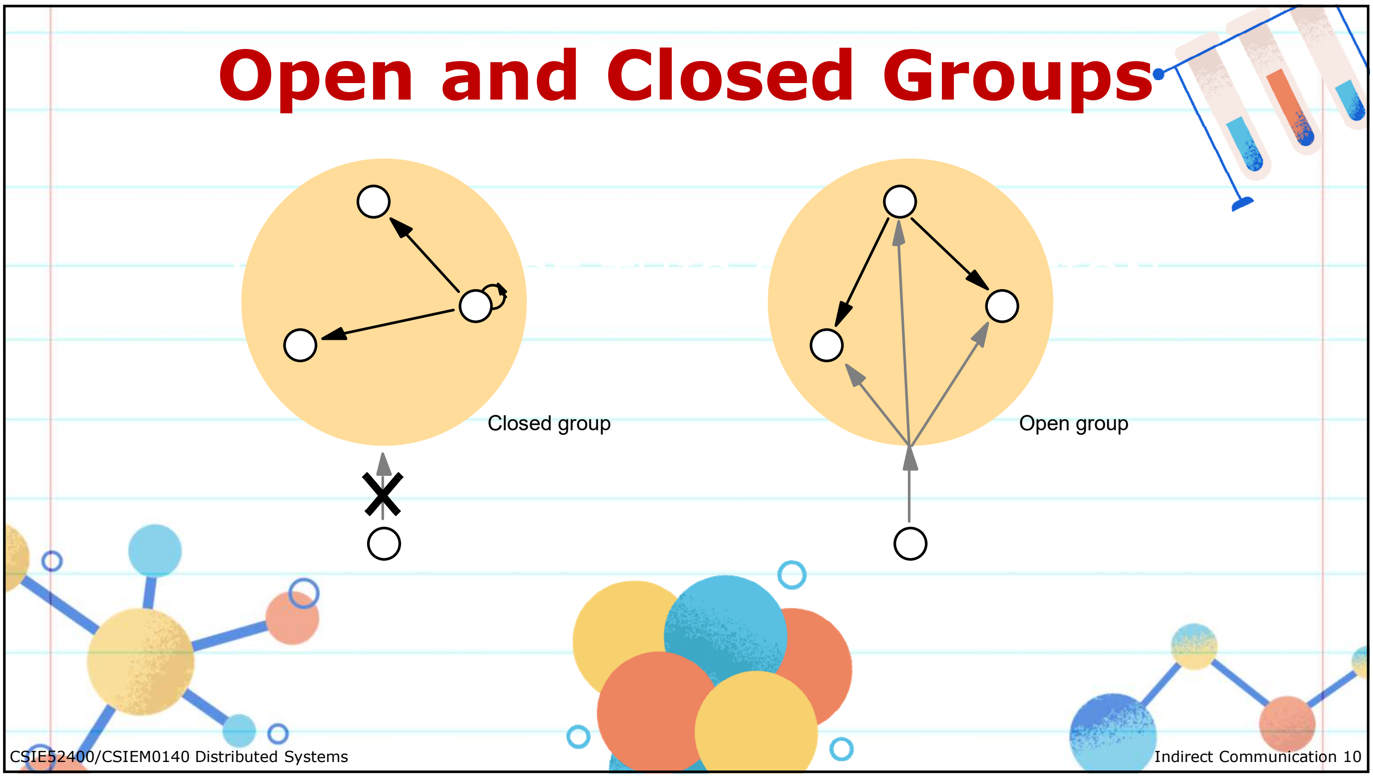
Groups

- Groups allow us to deal with a collection of processes as one abstraction
- Send message to one entity
  - ➢ Deliver to entire group
- Groups are dynamic
  - ➢ Created and destroyed
  - ➢ Processes can join or leave
    - May belong to 0 or more groups
- **Primitives**: join_group, leave_group, send_to_group, query_membership



Design Issues

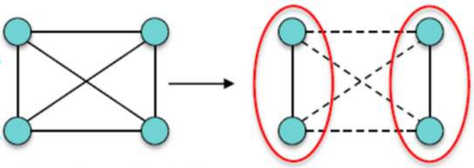
- Closed vs. Open (next slide)
  - Close: only members can multicast to it
  - Open: processes outside the group may send to it
- Peer vs. Hierarchical
  - Peer: each member communicates with group
  - Hierarchical: go through dedicated coordinator(s)
  - Diffusion(擴散): send to other servers & clients
- Managing membership & group creation/deletion
  - Distributed vs. centralized
- Leaving & joining must be synchronous
- Fault tolerance
  - Reliable message delivery?
  - What about missing members?



Open and Closed Groups

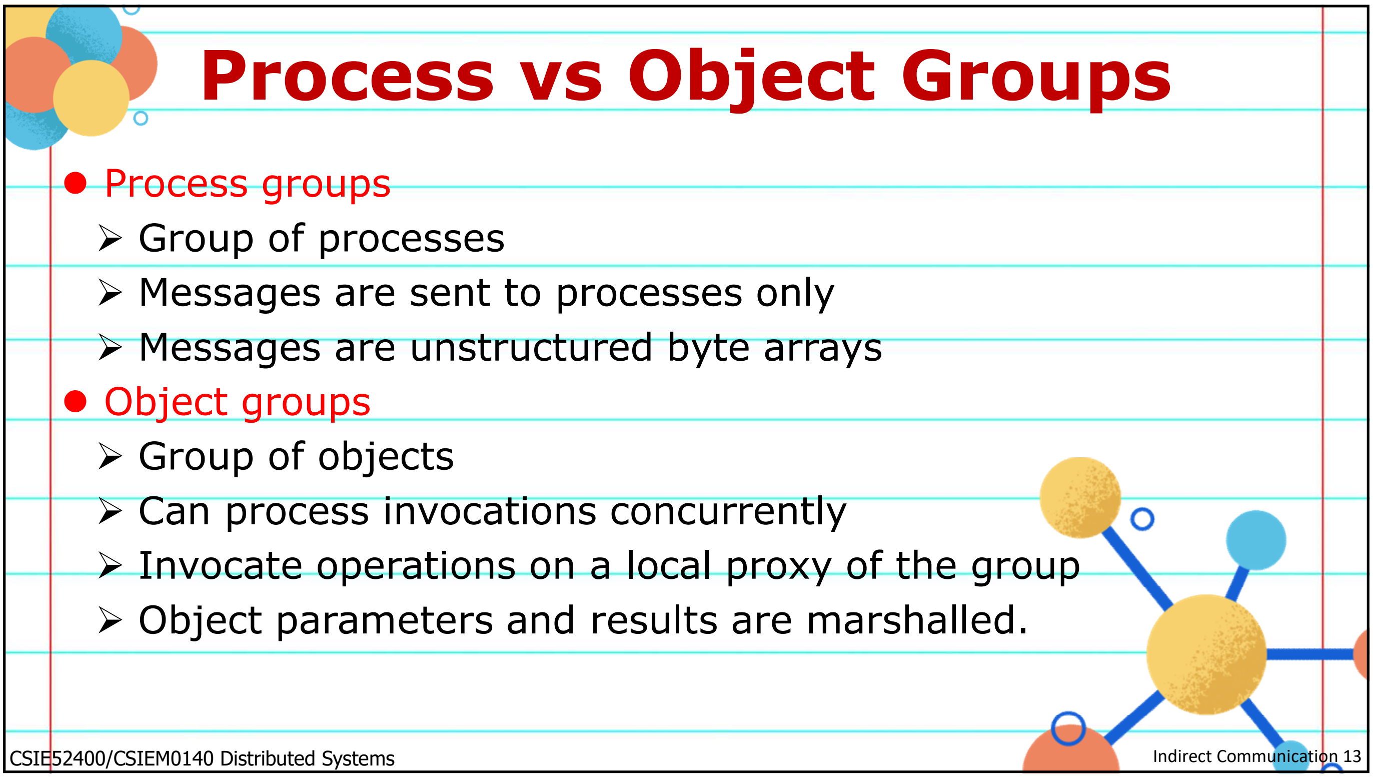
Failure Considerations

- Crash failure
  - Process stops communicating
- Omission failure (typically due to network)
  - Send omission: A process fails to send messages
  - Receive omission: A process fails to receive messages
- Byzantine failure
  - A message is faulty
- Partition failure
  - The network may get segmented, dividing the group into two or more unreachable sub-groups



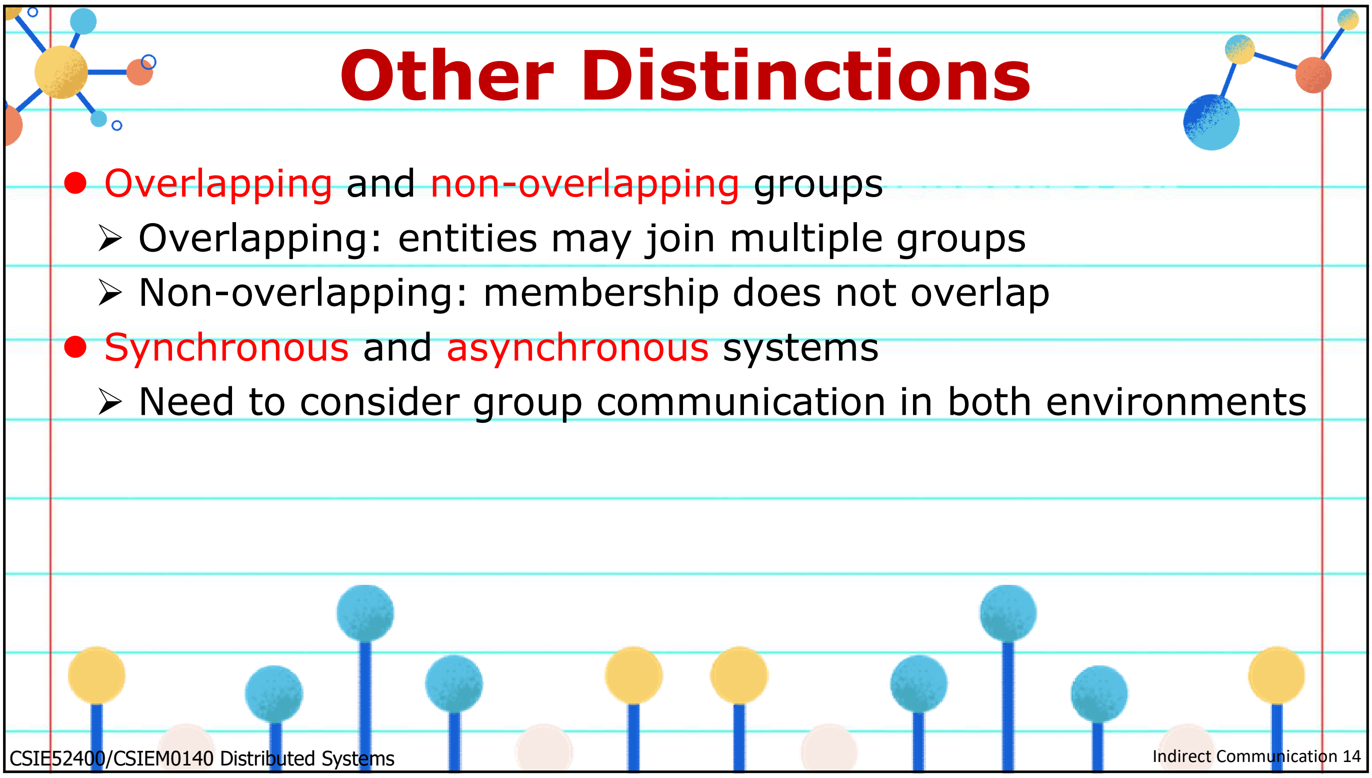
Programming Model

- Group, group membership
- Processes may join or leave the group.
- A single multicast operation such as aGroup.send(aMessage) is enough to send to each member of a group.
- **Advantages**:
  - Convenience for the programmers
  - Efficient utilization of bandwidth
  - Minimize total time to deliver the message to all



Process vs Object Groups

- Process groups
  - Group of processes
  - Messages are sent to processes only
  - Messages are unstructured byte arrays
- Object groups
  - Group of objects
  - Can process invocations concurrently
  - Invocate operations on a local proxy of the group
  - Object parameters and results are marshalled.



Other Distinctions

- Overlapping and non-overlapping groups
  - Overlapping: entities may join multiple groups
  - Non-overlapping: membership does not overlap
- Synchronous and asynchronous systems
  - Need to consider group communication in both environments

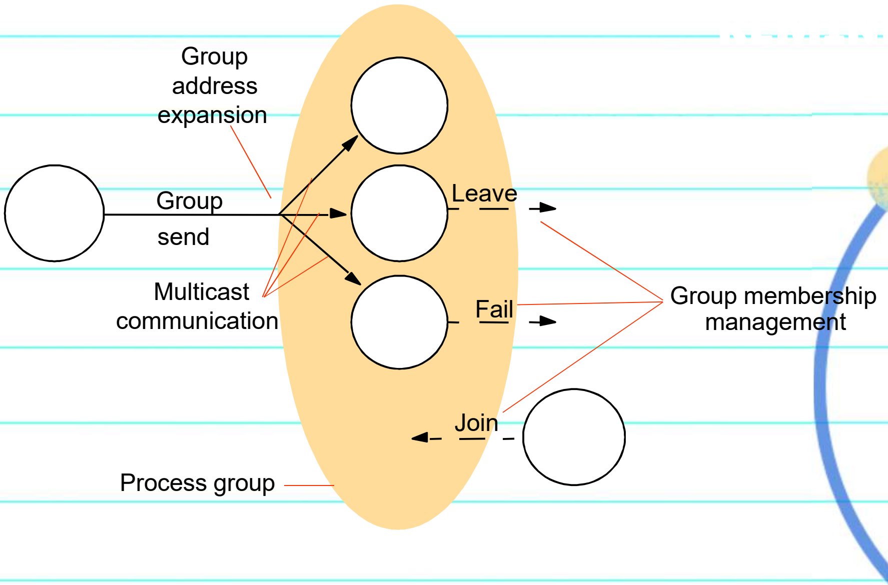
Implementation Issues

- **Reliability** (reliable multicast)
  - Integrity: message delivered intact w no duplicate
  - Validity: message sent is eventually delivered
  - Agreement: if a message is received by one, it must be delivered to all.
- **Ordering** (ordered multicast) – one or more
  - FIFO ordering: msgs delivered in sending order
  - Causal ordering: if a msg happens before another, it is delivered in that order
  - Total ordering: same ordering across all processes

Group Membership Mgnt

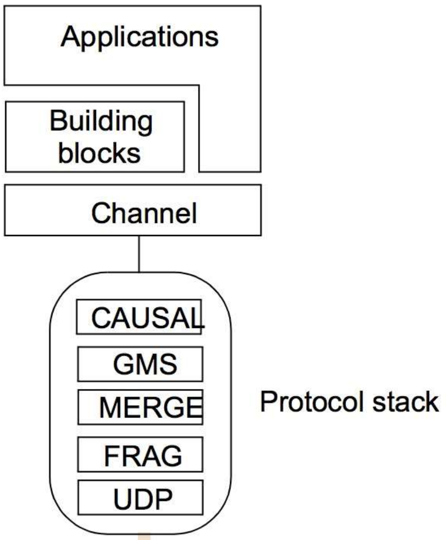
- Group membership services
  - Interface for group membership update (create/destroy groups, join/leave group)
  - Failure detection: to monitor the reachability of each member
  - Notification of membership changes: notify members about group changes
  - Group address expansion: a msg sent to a group identifier is expanded to the addresses of the members
- Called view-synchronous group communication

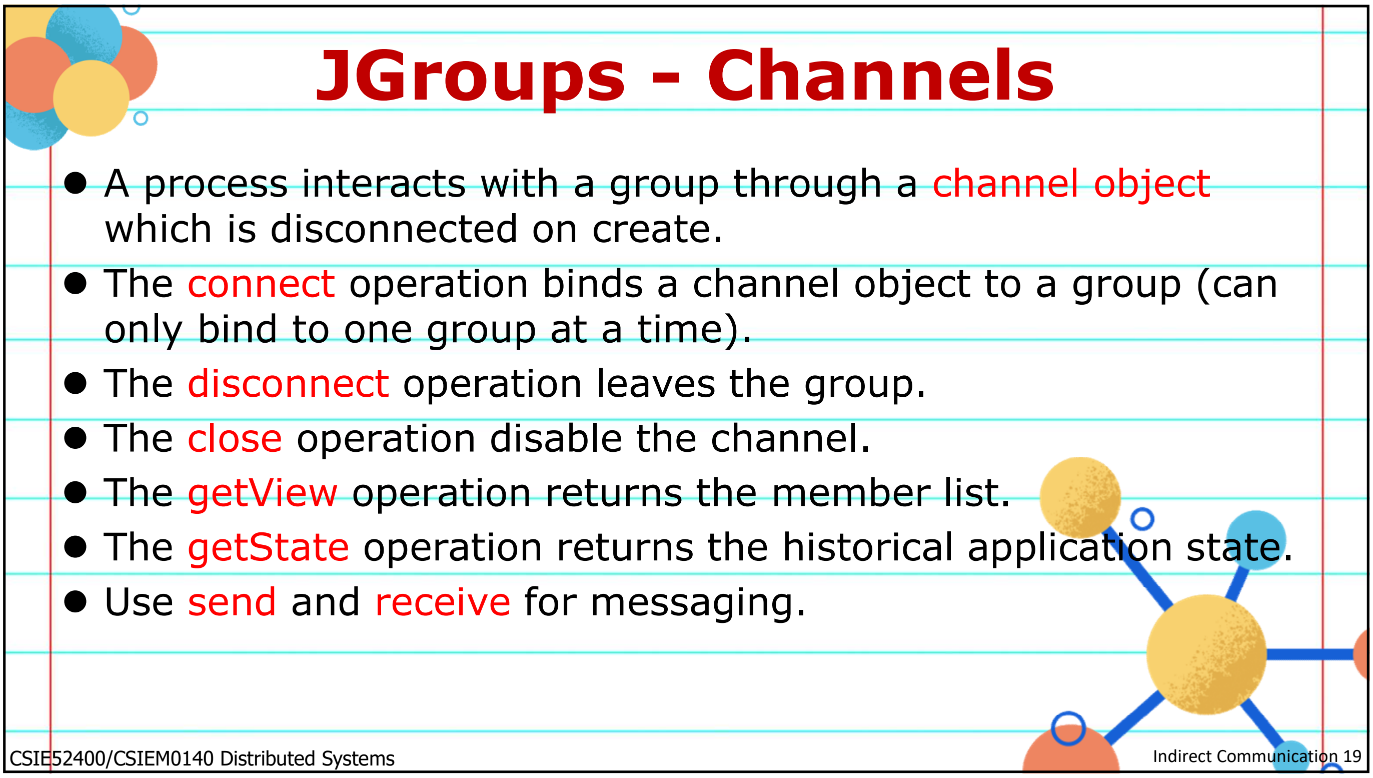
Group Membership Mgnt



JGroups Toolkit

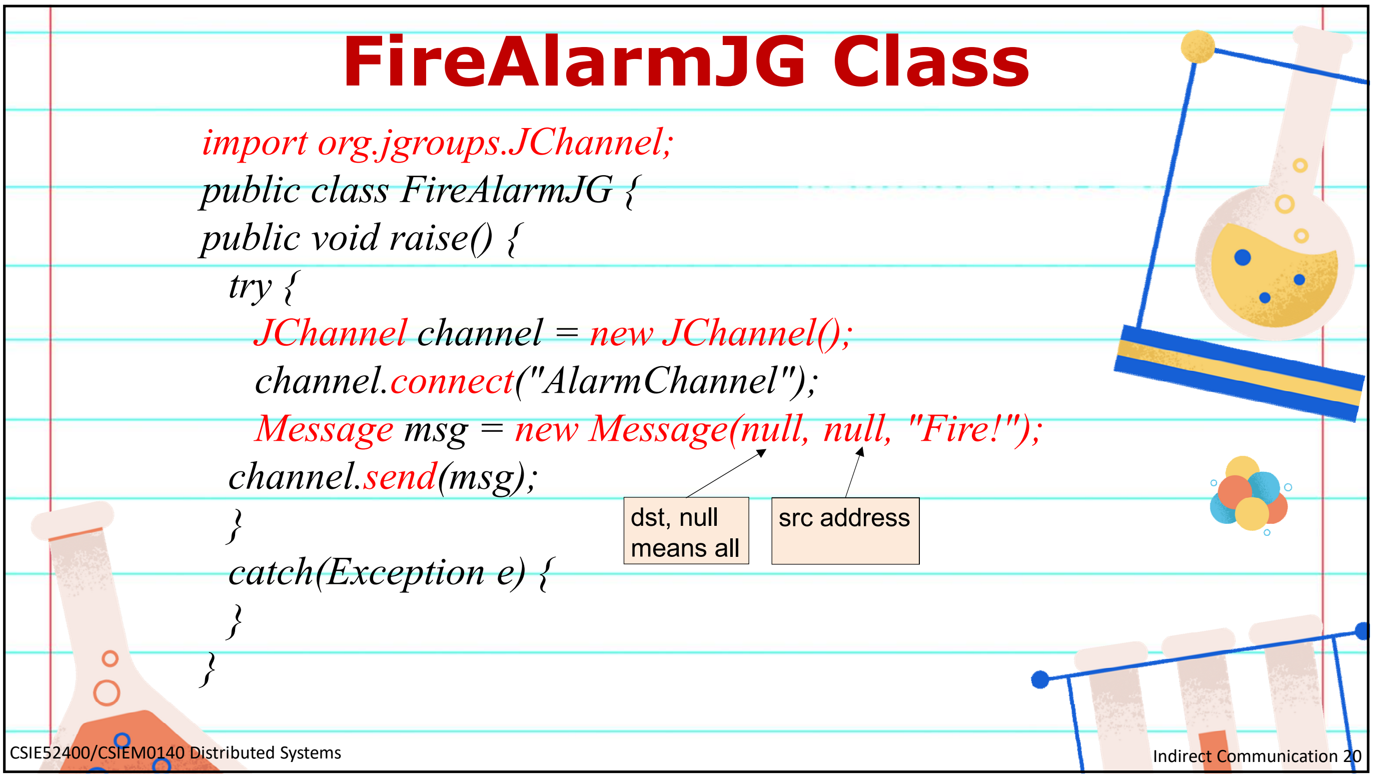
- Java toolkit for reliable group communication.
- Channels: primitive interface for joining, leaving, sending, receiving
- Building blocks: higher-level abstraction
- Protocol stack: underlying communication protocol





JGroups - Channels

- A process interacts with a group through a channel object which is disconnected on create.
- The connect operation binds a channel object to a group (can only bind to one group at a time).
- The disconnect operation leaves the group.
- The close operation disable the channel.
- The getView operation returns the member list.
- The getState operation returns the historical application state.
- Use send and receive for messaging.

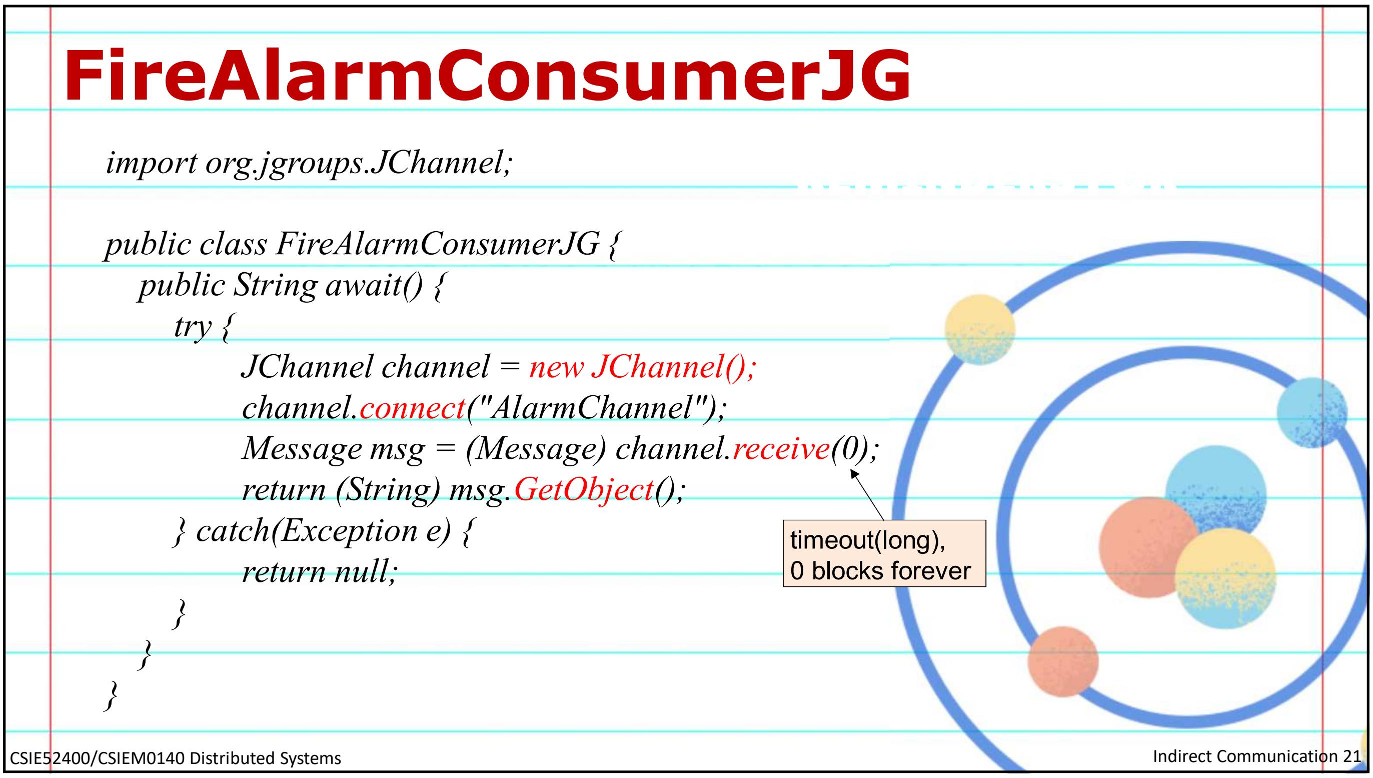


FireAlarmJG Class

```
import org.jgroups.JChannel;
public class FireAlarmJG {
public void raise() {
  try {
    JChannel channel = new JChannel();
    channel.connect("AlarmChannel");
    Message msg = new Message(null, null, "Fire!");
  channel.send(msg);
  }
  catch(Exception e) {
  }
}
```

dst, null means all

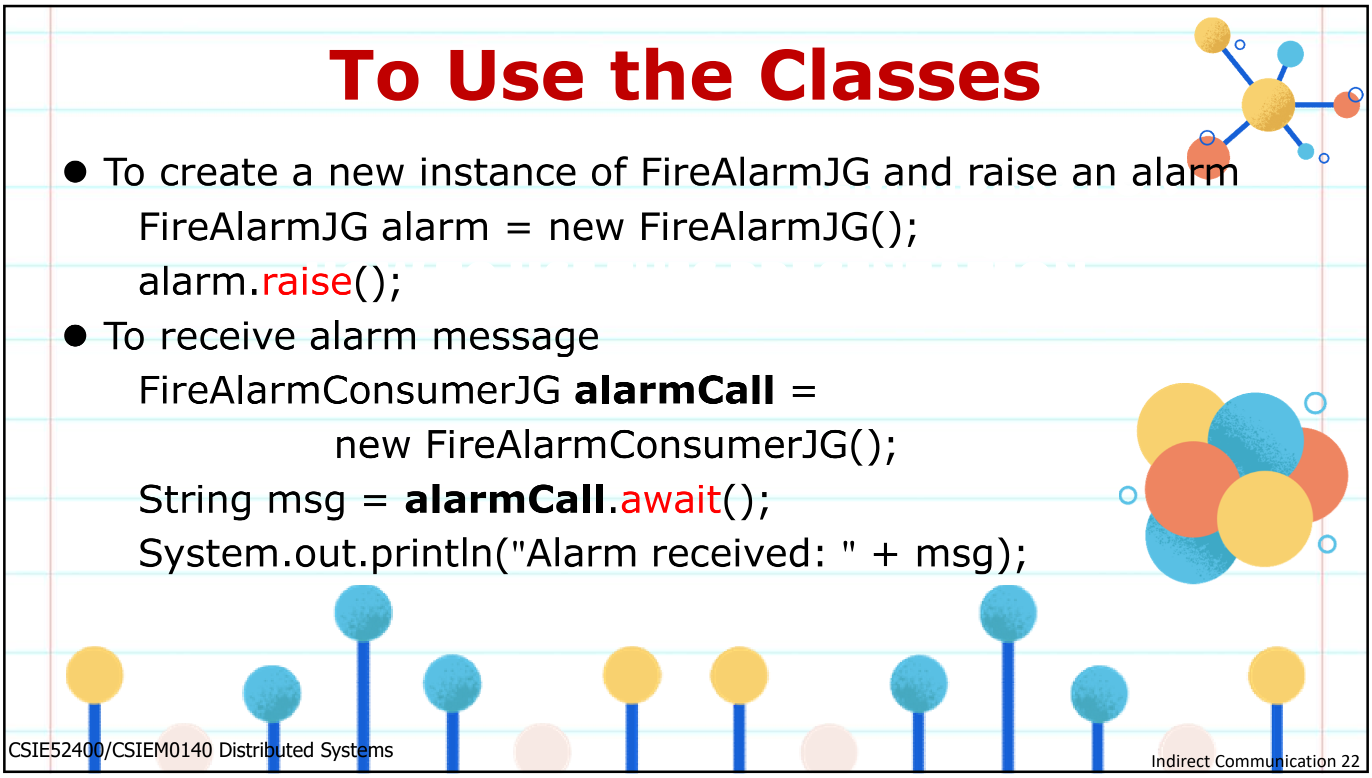
src address



FireAlarmConsumerJG

```
import org.jgroups.JChannel;

public class FireAlarmConsumerJG {
   public String await() {
      try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = (Message) channel.receive(0);
            return (String) msg.GetObject();
      } catch(Exception e) {
            return null;
      }
   }
}
```

timeout(long),
0 blocks forever



To Use the Classes

- To create a new instance of FireAlarmJG and raise an alarm

  FireAlarmJG alarm = new FireAlarmJG();
  alarm.raise();

- To receive alarm message

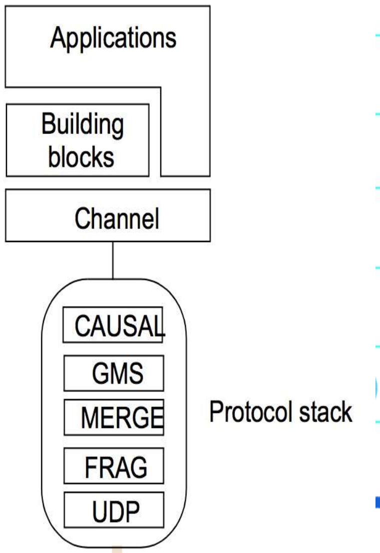
  FireAlarmConsumerJG **alarmCall** =
  new FireAlarmConsumerJG();
  String msg = **alarmCall**.await();
  System.out.println("Alarm received: " + msg);

Building Blocks

- Higher-level abstraction on top of channels.
- MessageDispatcher provides synchronous and asynchronous message sending, as well as conditional receiving.
- RpcDispatcher invokes remote methods on all objects of a group.
- NotificationBus provides notification sending and handling capability.
- Read the online JGroups API for more details.

Protocol Stacks

- UDP is the common transport layer in JGroups.
- FRAG implements message packetization.
- MERGE deals with unexpected network partitioning and the subsequent merging of subgroups.
- GMS implements a group membership protocol.
- CAUSAL implements causal ordering.

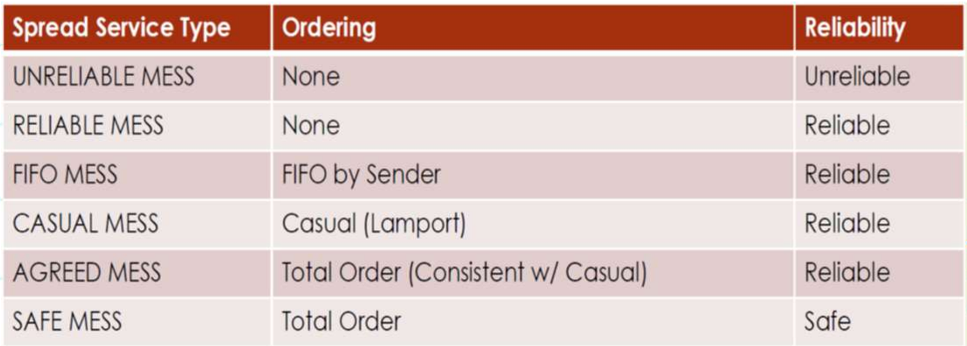


Spread Toolkit

- An open source toolkit providing high performance group communication system resilient to faults across networks.
  - Reliable and scalable messaging and group communication.
  - A powerful but simple API.
  - Easy to use, deploy and maintain.
  - Highly scalable from one LAN to complex wide area networks.
  - Supports thousands of groups with different sets of members.
  - Reliable messaging in the presence of machine failures, process crashes and recoveries, and network partitions and merges.
  - Provides a range of reliability, ordering and stability guarantees.
  - Emphasis on robustness and high performance.
  - Completely distributed algorithms with no central point of failure.
  - Interfaces with C/C++, Java, Perl, Python, Ruby, PHP, …

Spread Level of Service

- When an application sends a Spread message, it chooses a **level of service for that message.**
- It controls what kind of ordering and reliability are provided to that message.



Spread Service Type	Ordering	Reliability
UNRELIABLE MESS	None	Unreliable
RELIABLE MESS	None	Reliable
FIFO MESS	FIFO by Sender	Reliable
CASUAL MESS	Casual (Lamport)	Reliable
AGREED MESS	Total Order (Consistent w/ Casual)	Reliable
SAFE MESS	Total Order	Safe

Spread Architecture

- SPREAD is composed of 2 entities – daemons and client processes.
- Daemon(s) maintain information about all processes connected to it and the presence of other daemon(s), if any.
- Client processes join/leave the group or send/receive messages using the SPREAD primitives.

Spread Architecture

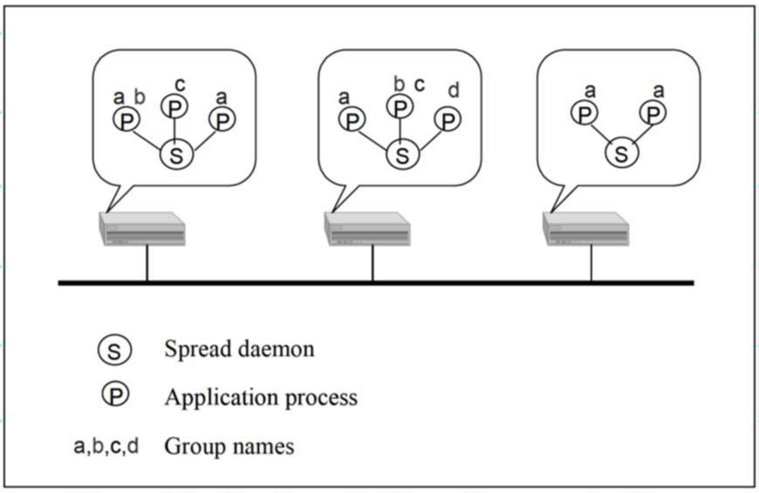
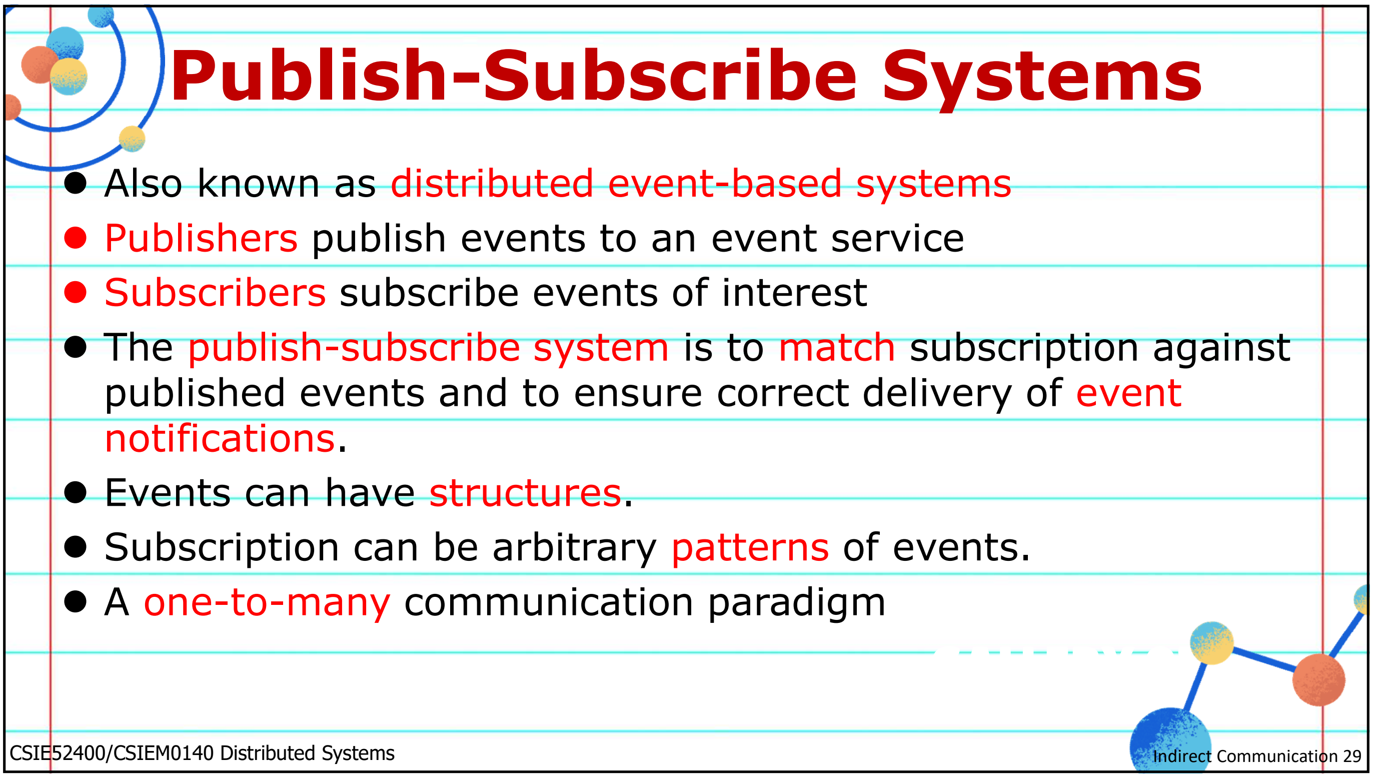
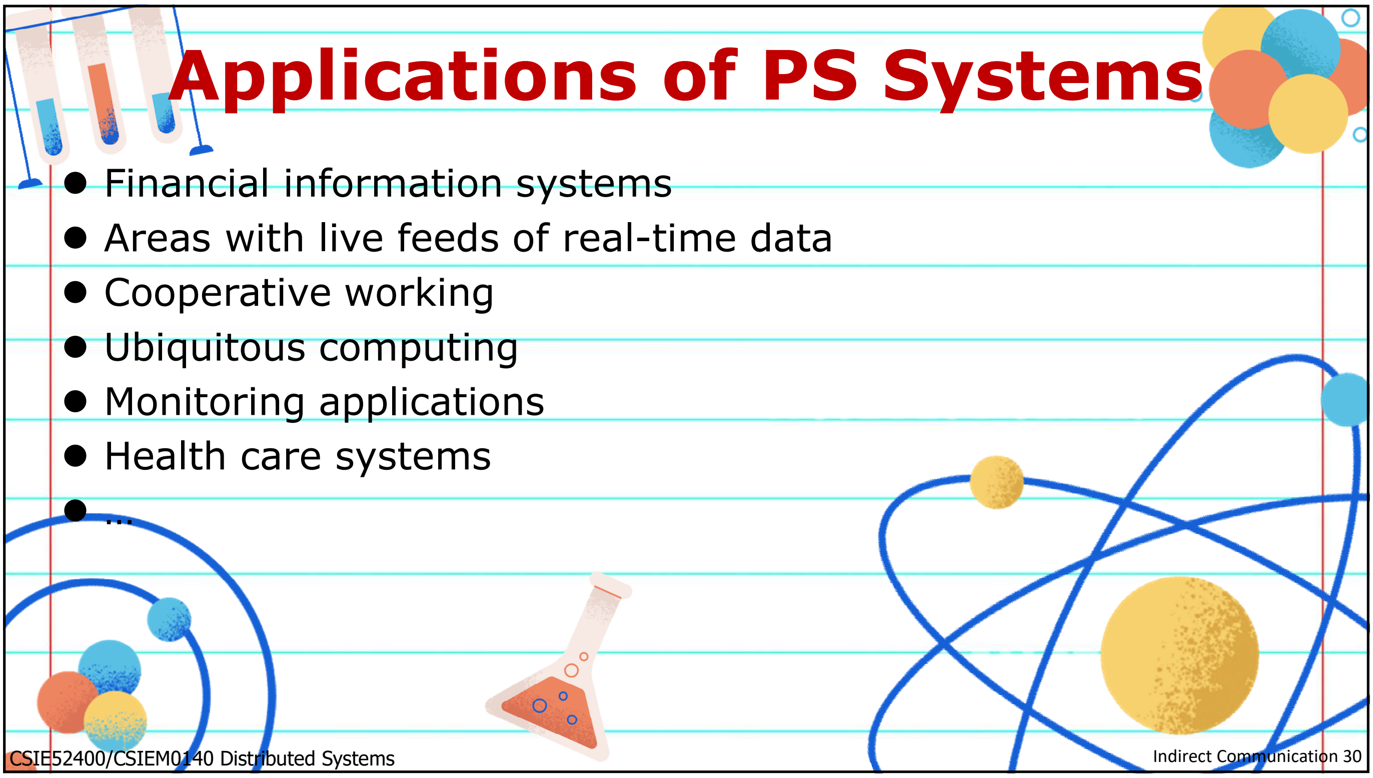


Figure 1.1: The Spread Client-Daemon Architecture



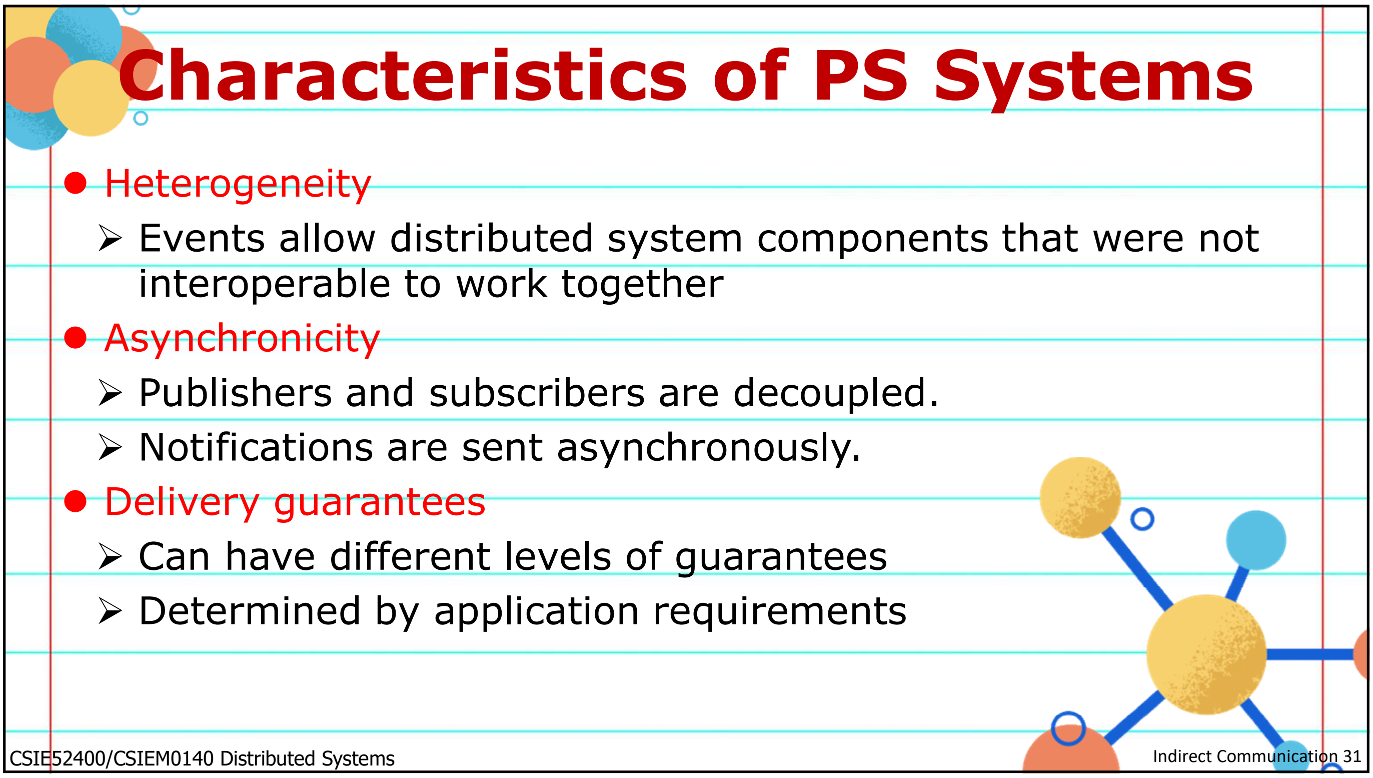
Publish-Subscribe Systems

- Also known as distributed event-based systems
- Publishers publish events to an event service
- Subscribers subscribe events of interest
- The publish-subscribe system is to match subscription against published events and to ensure correct delivery of event notifications.
- Events can have structures.
- Subscription can be arbitrary patterns of events.
- A one-to-many communication paradigm



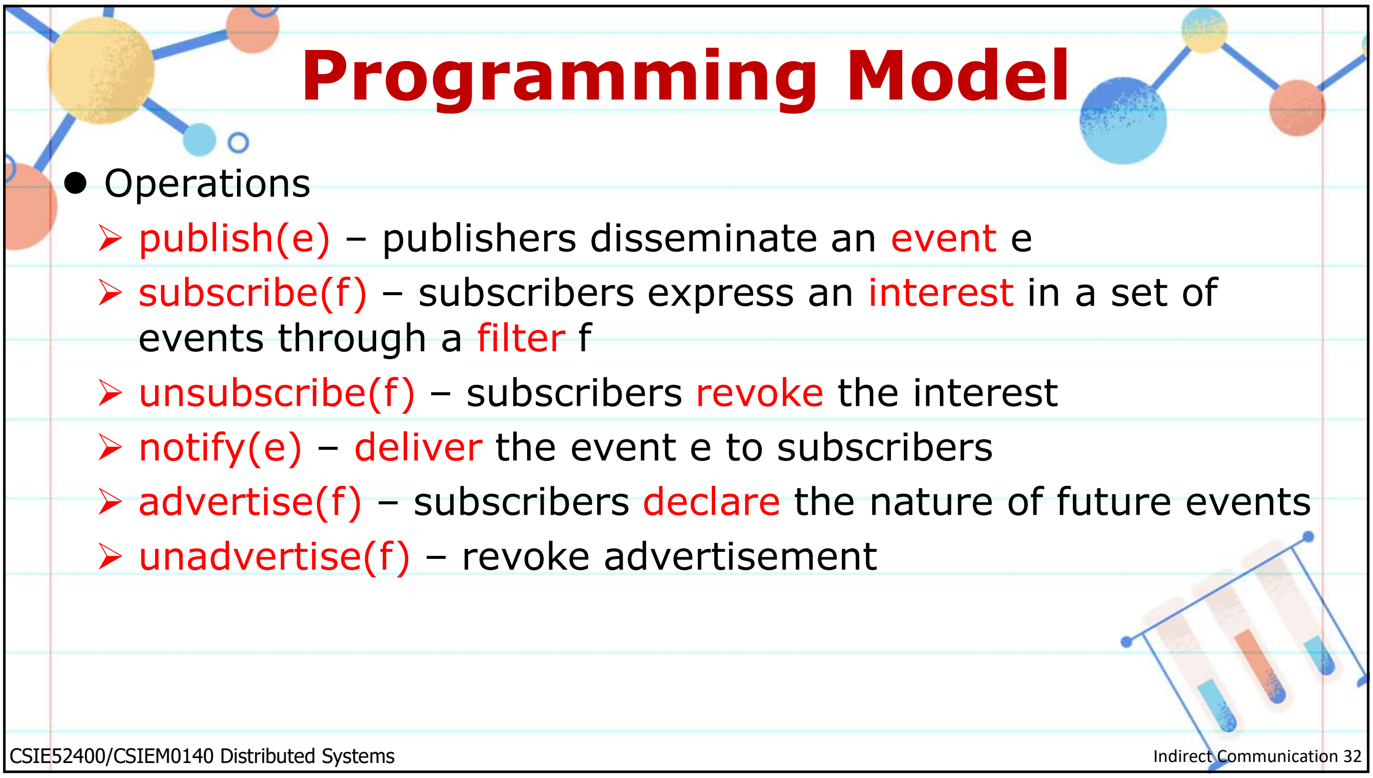
Applications of PS Systems

- Financial information systems
- Areas with live feeds of real-time data
- Cooperative working
- Ubiquitous computing
- Monitoring applications
- Health care systems
- …



Characteristics of PS Systems

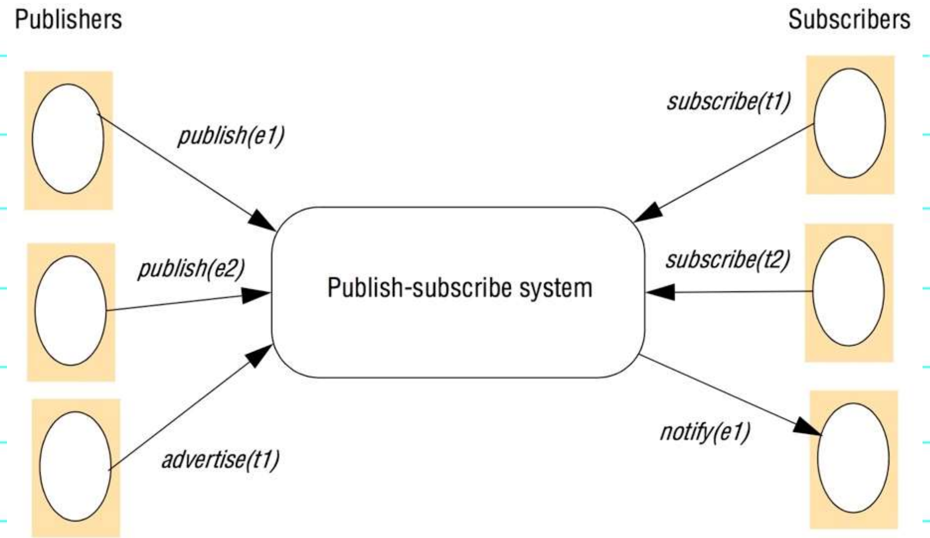
- Heterogeneity
  - Events allow distributed system components that were not interoperable to work together
- Asynchronicity
  - Publishers and subscribers are decoupled.
  - Notifications are sent asynchronously.
- Delivery guarantees
  - Can have different levels of guarantees
  - Determined by application requirements



Programming Model

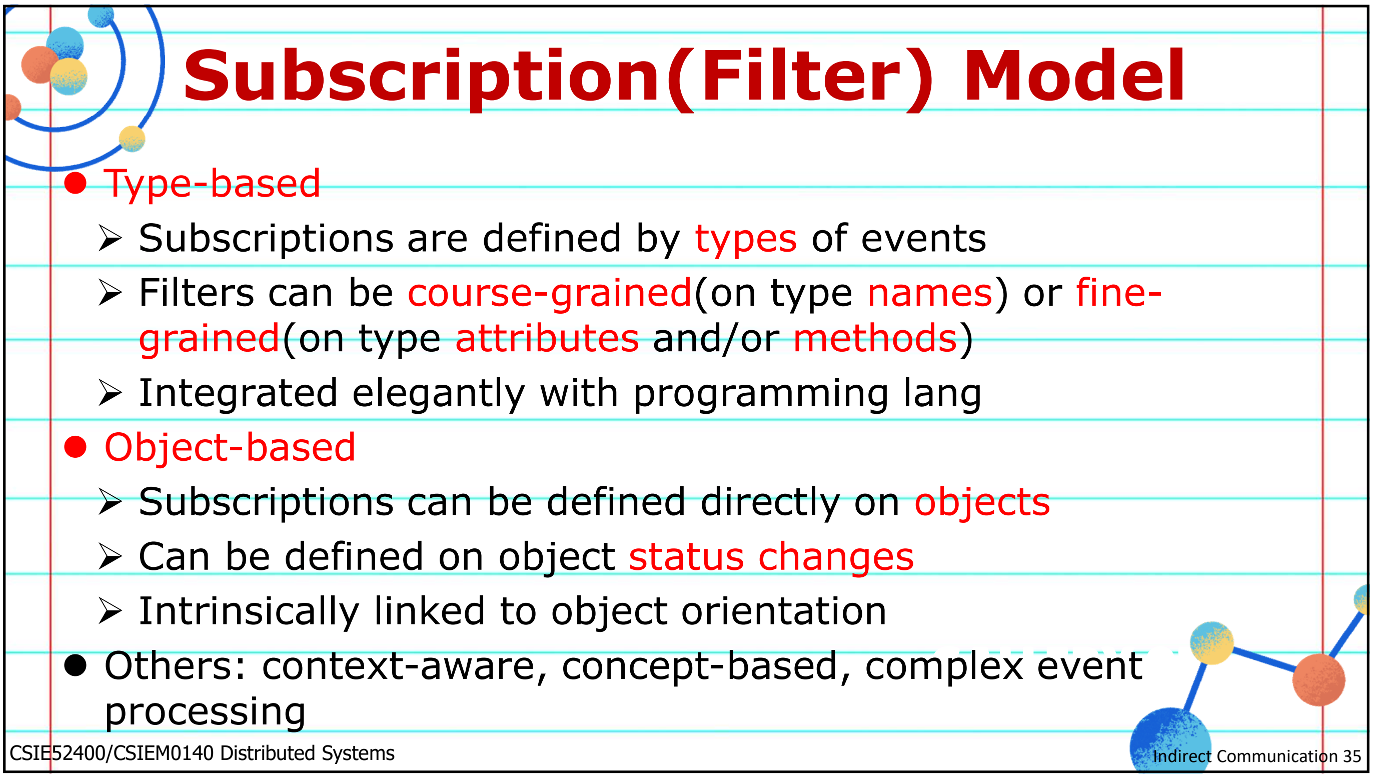
- Operations
  - publish(e) – publishers disseminate an event e
  - subscribe(f) – subscribers express an interest in a set of events through a filter f
  - unsubscribe(f) – subscribers revoke the interest
  - notify(e) – deliver the event e to subscribers
  - advertise(f) – subscribers declare the nature of future events
  - unadvertise(f) – revoke advertisement

Publish-Subscribe Paradigm



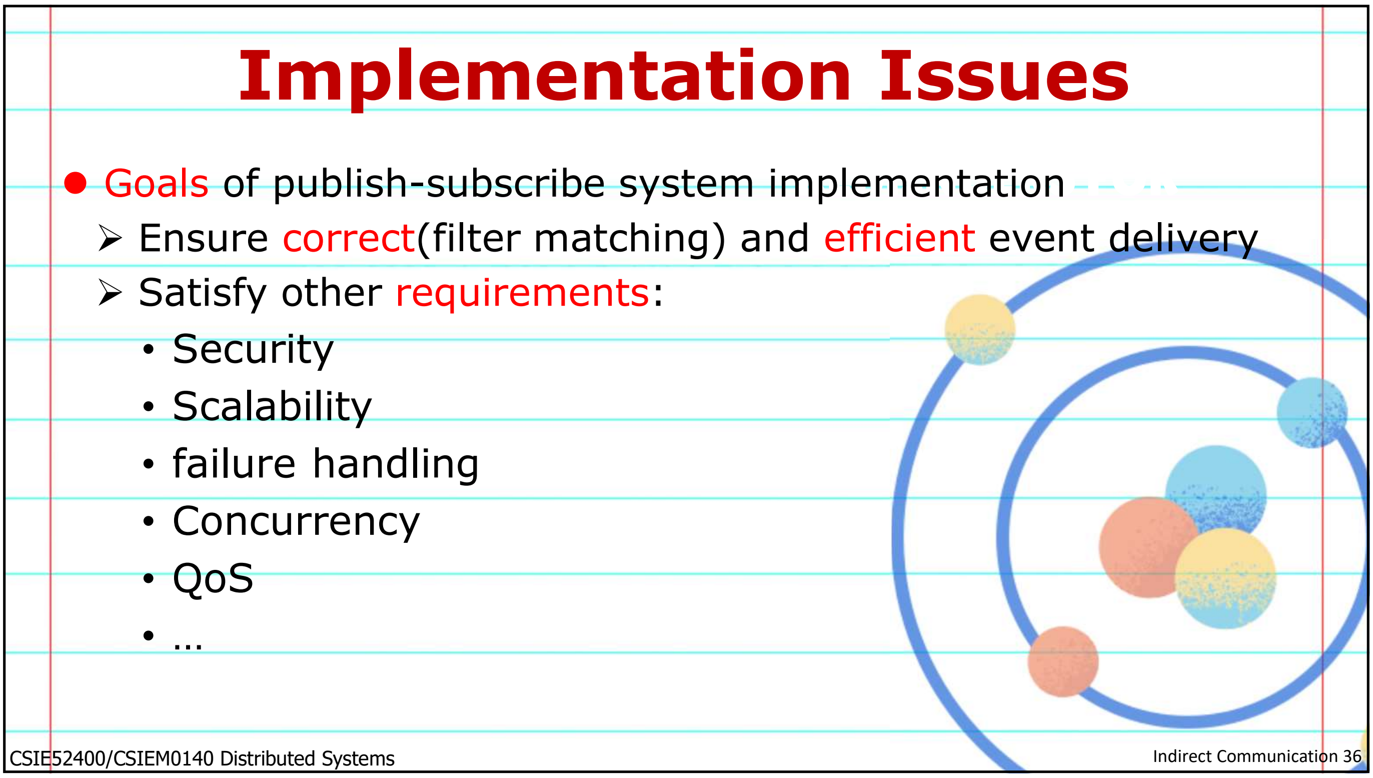
Subscription(Filter) Model

- Channel-based
  - Use named channels to publish/subscribe
  - Primitive scheme with physical channel
- Topic-based (Subject-based)
  - Notification is expressed by topic and other fields
  - Subscriptions are defined by topic(s) of interest
  - Topic can be hierarchically organized
- Content-based
  - The filter is a query defined on event attributes
  - Much more expressive then two approaches above



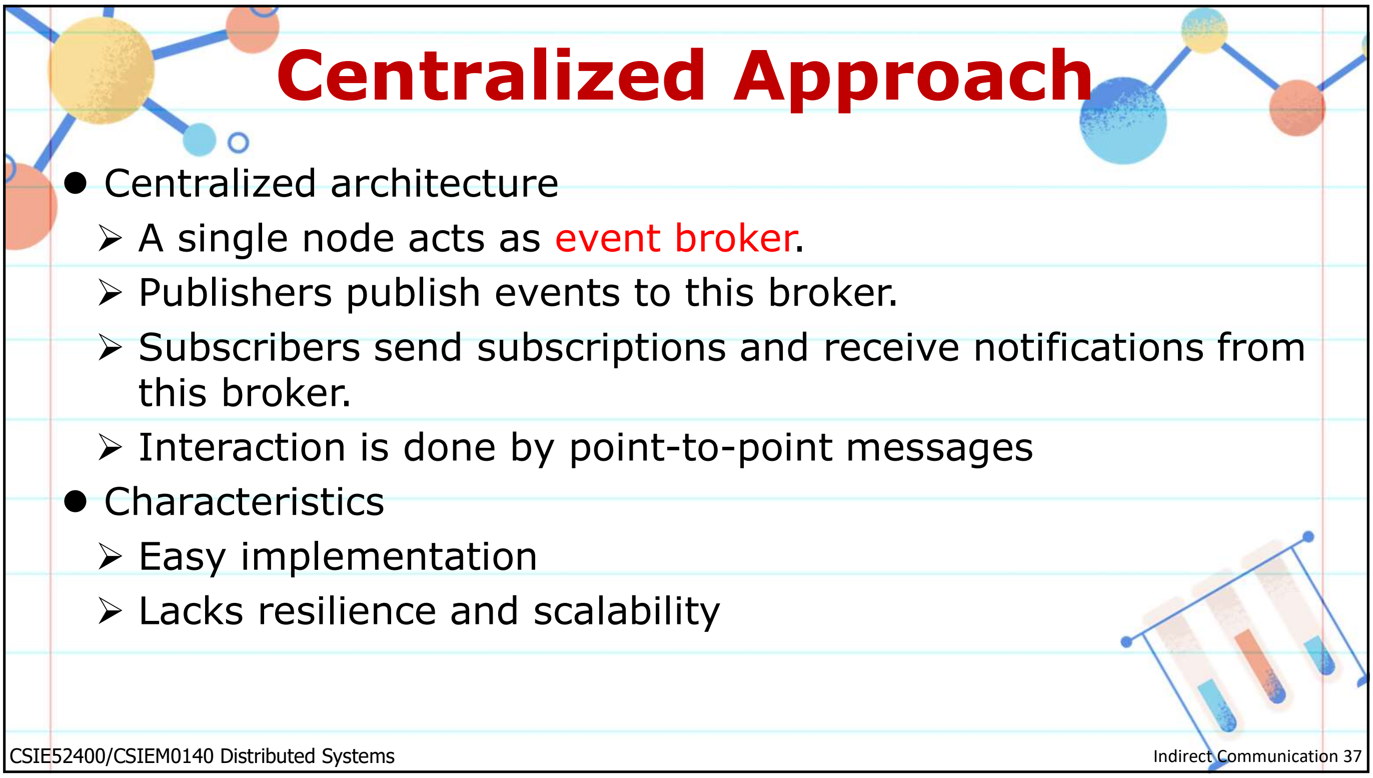
Subscription(Filter) Model

- Type-based
  - Subscriptions are defined by types of events
  - Filters can be course-grained(on type names) or fine-grained(on type attributes and/or methods)
  - Integrated elegantly with programming lang
- Object-based
  - Subscriptions can be defined directly on objects
  - Can be defined on object status changes
  - Intrinsically linked to object orientation
- Others: context-aware, concept-based, complex event processing



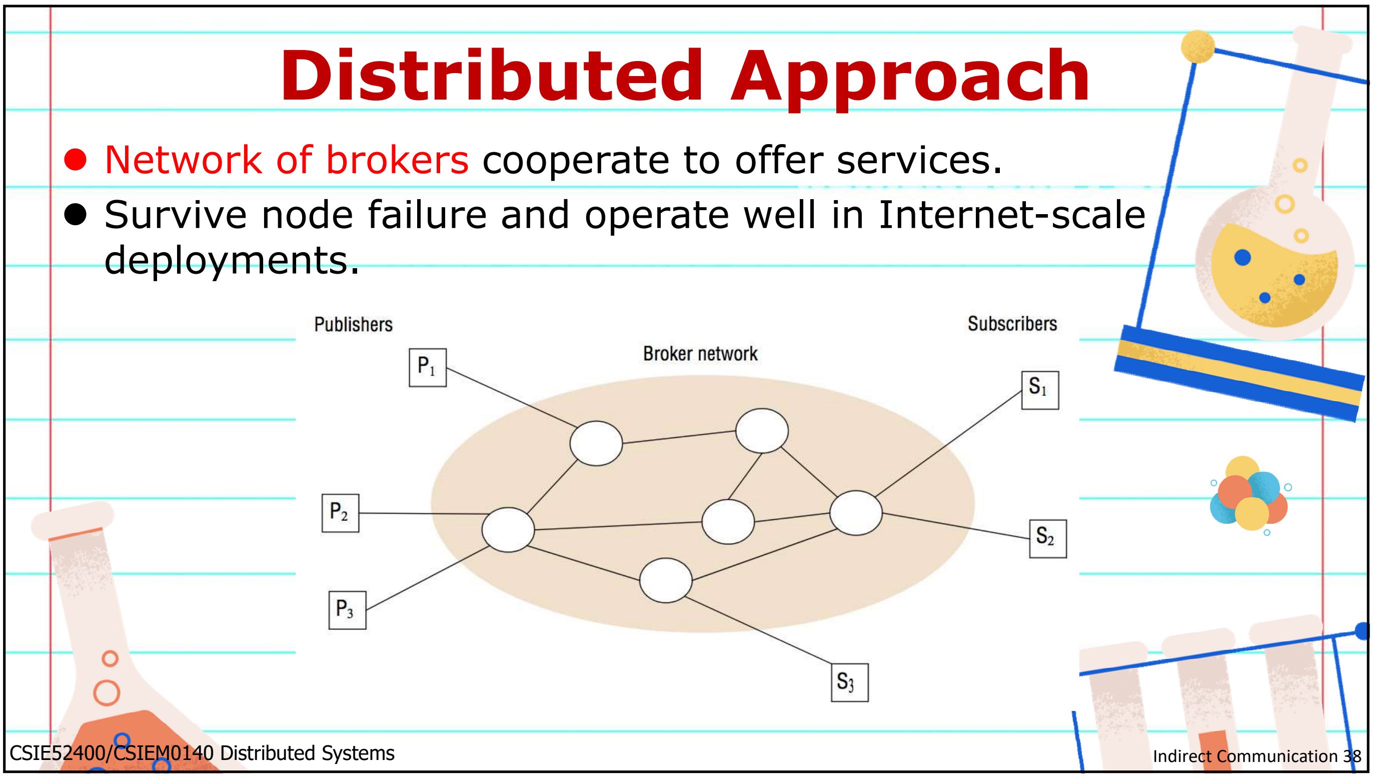
Implementation Issues

- Goals of publish-subscribe system implementation
  - Ensure correct(filter matching) and efficient event delivery
  - Satisfy other requirements:
    - Security
    - Scalability
    - failure handling
    - Concurrency
    - QoS
    - …



Centralized Approach

- Centralized architecture
  - A single node acts as event broker.
  - Publishers publish events to this broker.
  - Subscribers send subscriptions and receive notifications from this broker.
  - Interaction is done by point-to-point messages
- Characteristics
  - Easy implementation
  - Lacks resilience and scalability



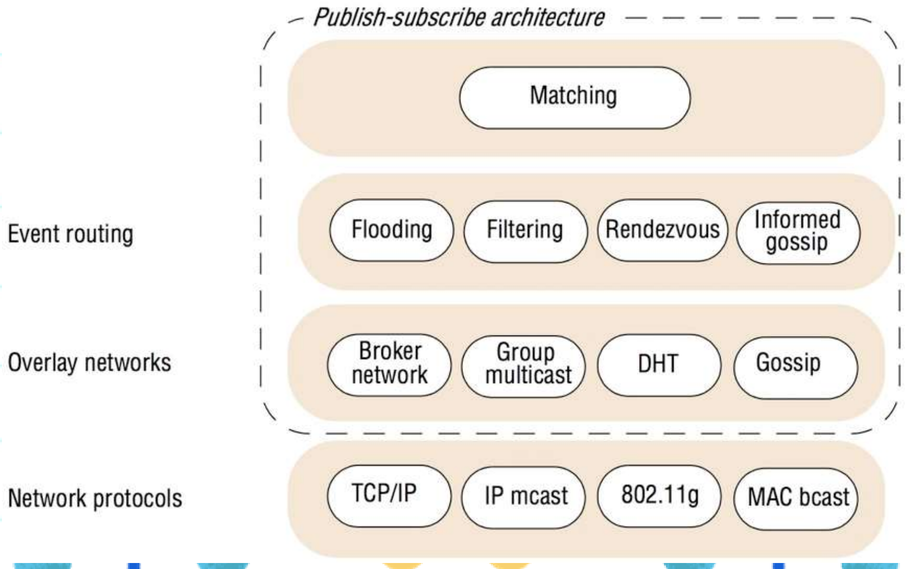
Distributed Approach

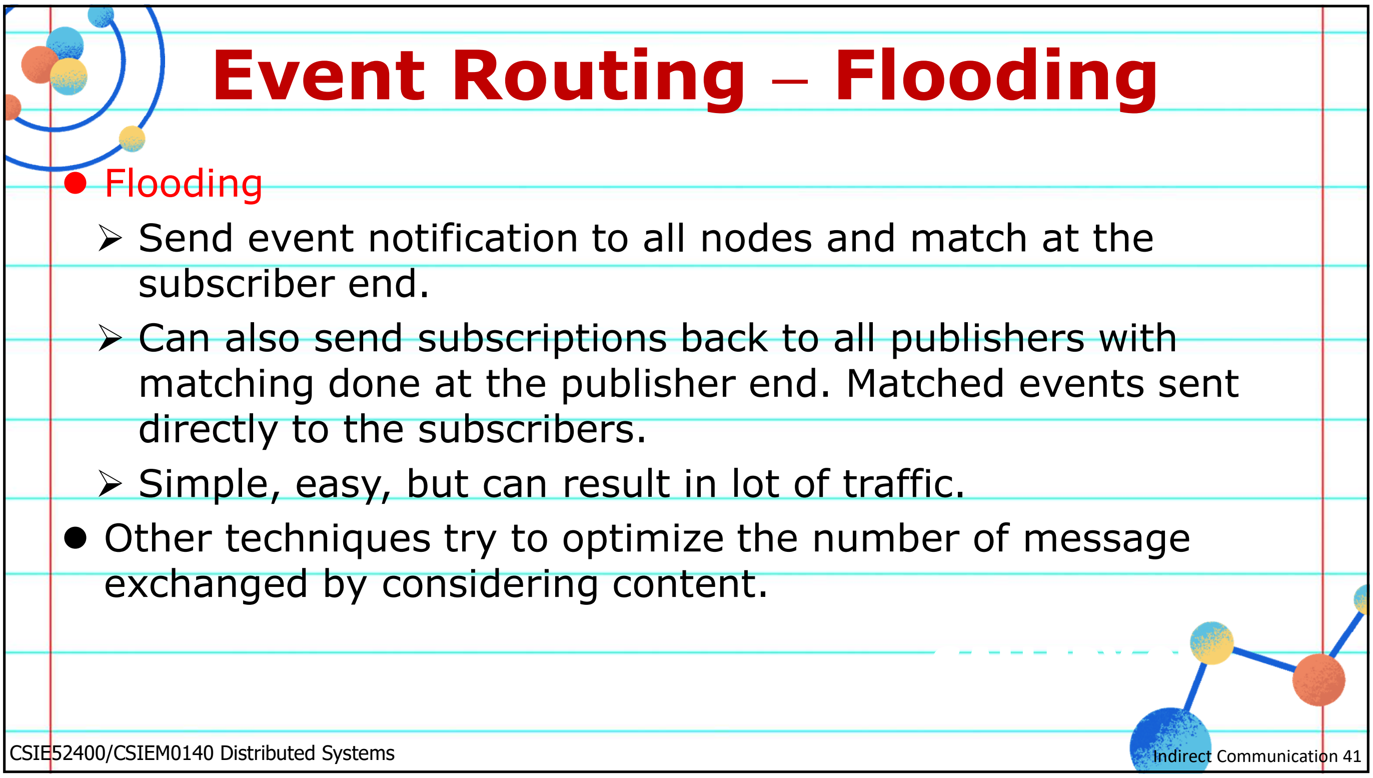
- Network of brokers cooperate to offer services.
- Survive node failure and operate well in Internet-scale deployments.

Peer-to-peer Approach

- No distinction between publishers, subscribers and brokers.
- All nodes act as brokers, cooperatively implementing the desired functionalities.
- Very popular for recent systems

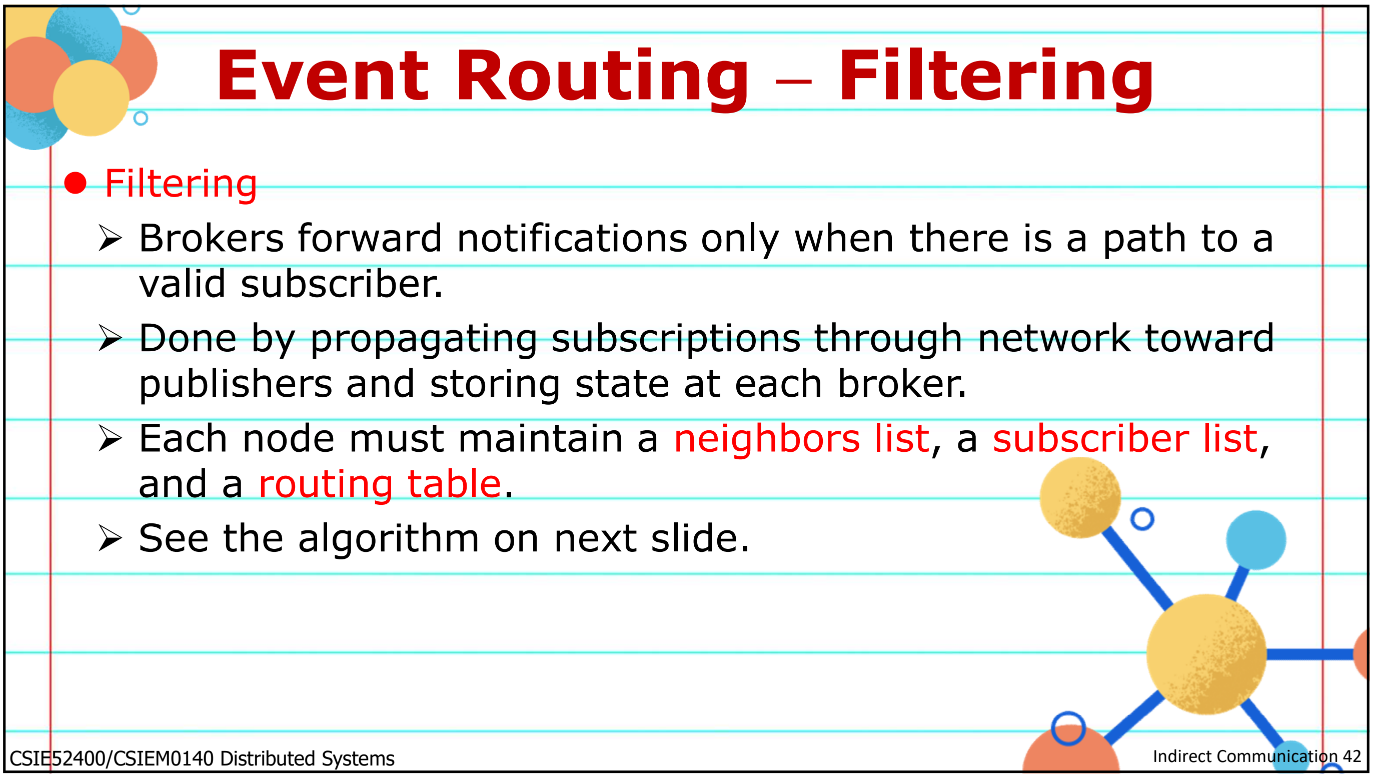
Implementation Architecture





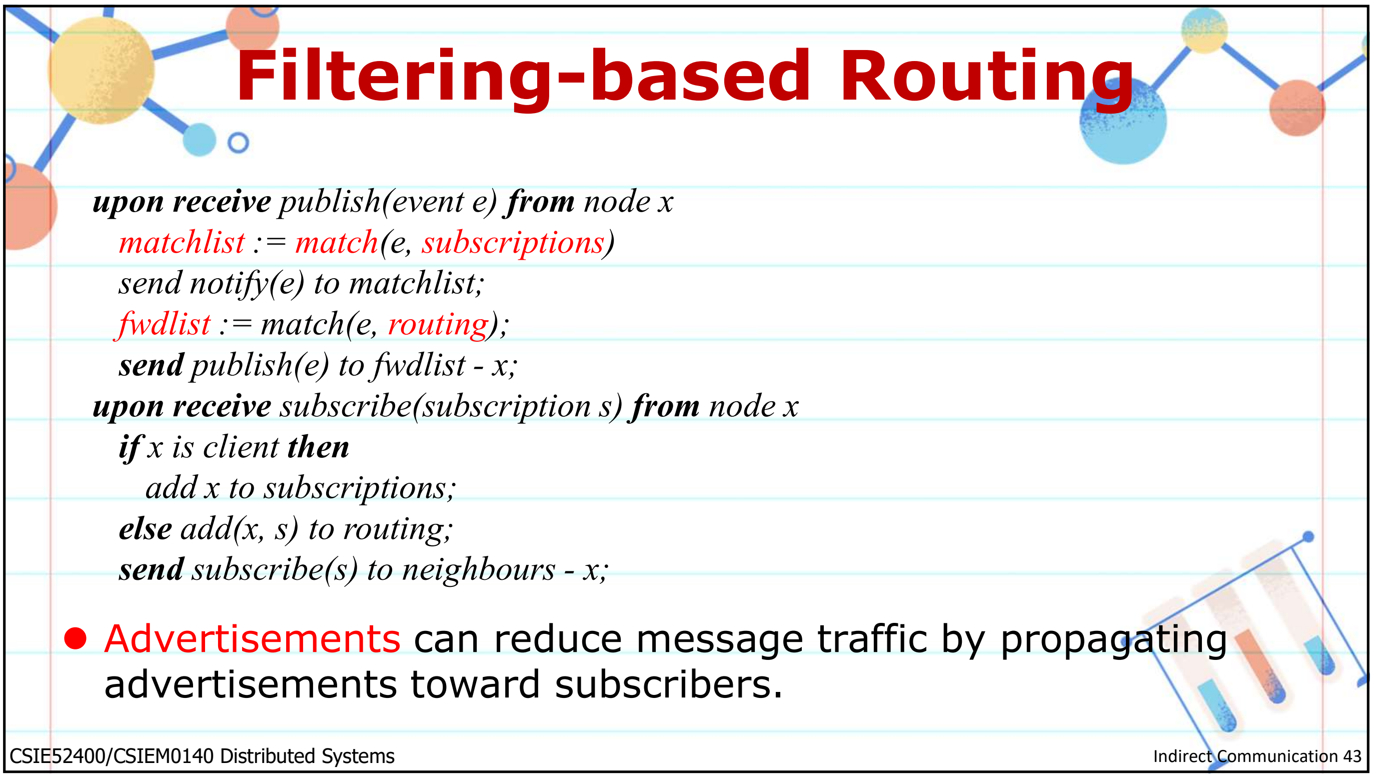
Event Routing – Flooding

- Flooding
  - Send event notification to all nodes and match at the subscriber end.
  - Can also send subscriptions back to all publishers with matching done at the publisher end. Matched events sent directly to the subscribers.
  - Simple, easy, but can result in lot of traffic.
- Other techniques try to optimize the number of message exchanged by considering content.



Event Routing – Filtering

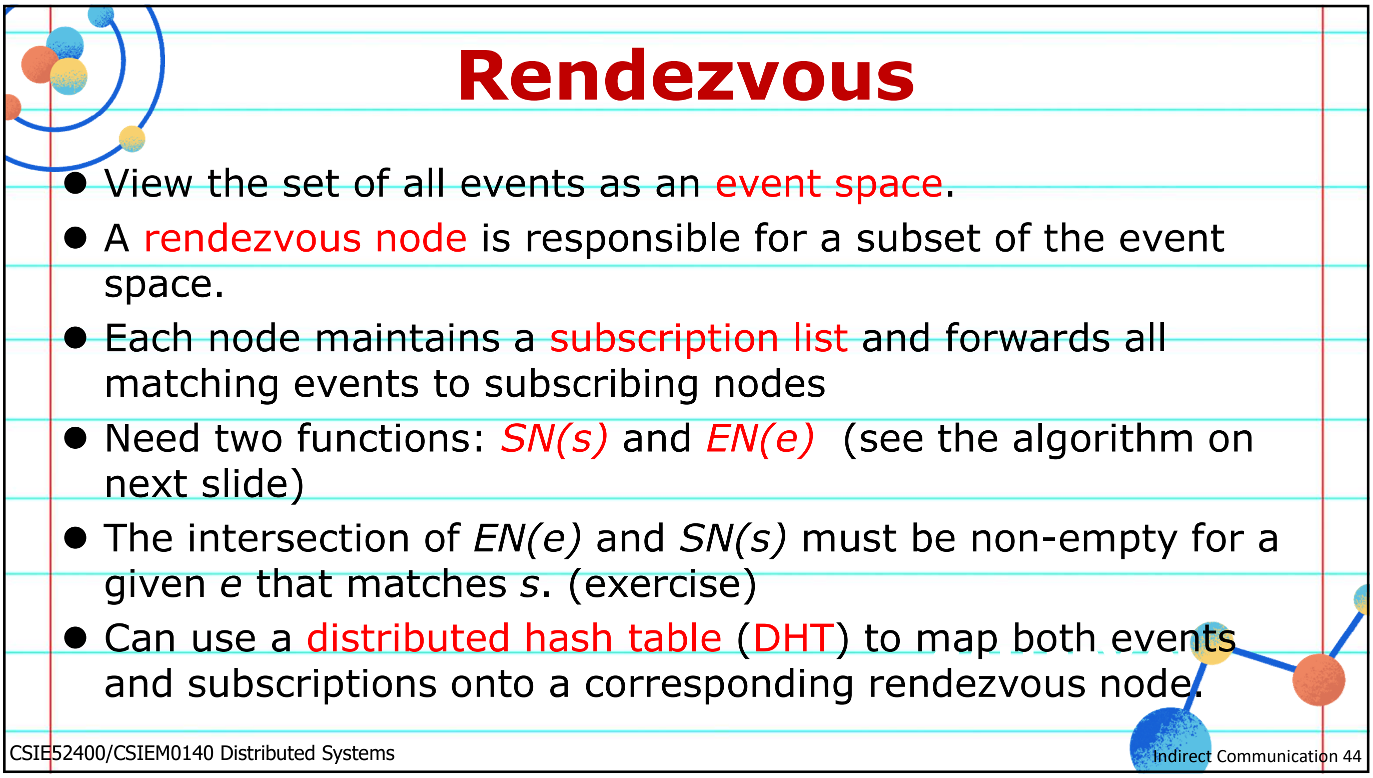
- Filtering
  - Brokers forward notifications only when there is a path to a valid subscriber.
  - Done by propagating subscriptions through network toward publishers and storing state at each broker.
  - Each node must maintain a neighbors list, a subscriber list, and a routing table.
  - See the algorithm on next slide.



Filtering-based Routing

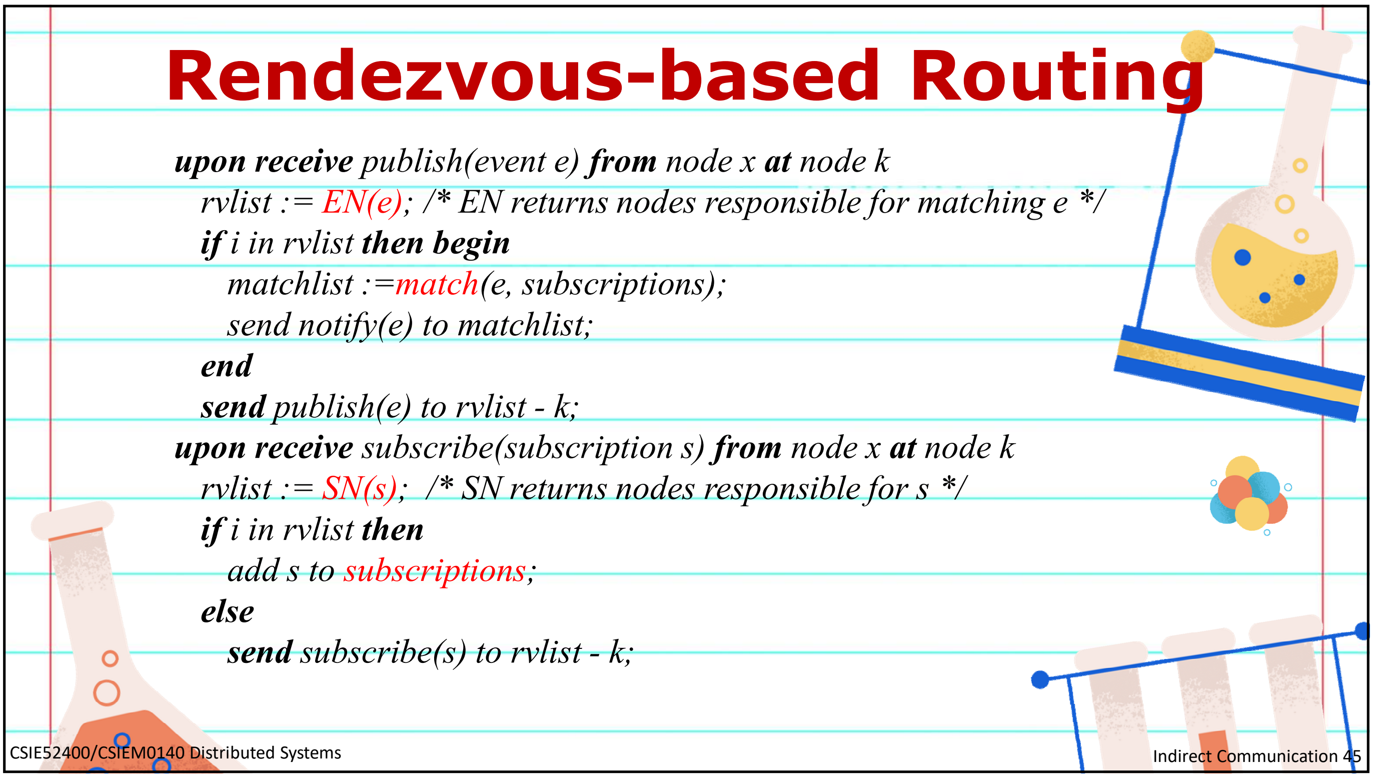
```
upon receive publish(event e) from node x
  matchlist := match(e, subscriptions)
  send notify(e) to matchlist;
  fwdlist := match(e, routing);
  send publish(e) to fwdlist - x;
upon receive subscribe(subscription s) from node x
  if x is client then
    add x to subscriptions;
  else add(x, s) to routing;
  send subscribe(s) to neighbours - x;
```

- Advertisements can reduce message traffic by propagating advertisements toward subscribers.



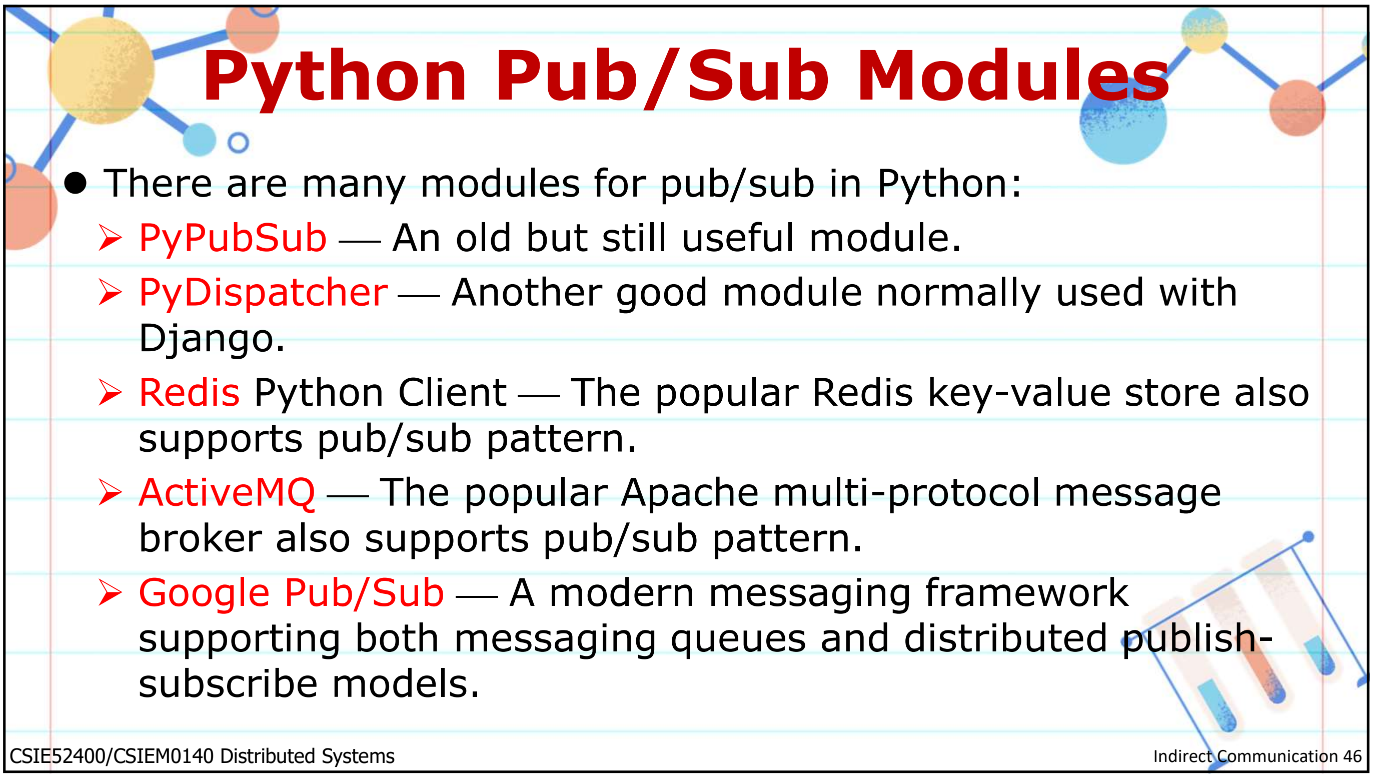
Rendezvous

- View the set of all events as an event space.
- A rendezvous node is responsible for a subset of the event space.
- Each node maintains a subscription list and forwards all matching events to subscribing nodes
- Need two functions: *SN(s)* and *EN(e)* (see the algorithm on next slide)
- The intersection of *EN(e)* and *SN(s)* must be non-empty for a given *e* that matches *s*. (exercise)
- Can use a distributed hash table (DHT) to map both events and subscriptions onto a corresponding rendezvous node.



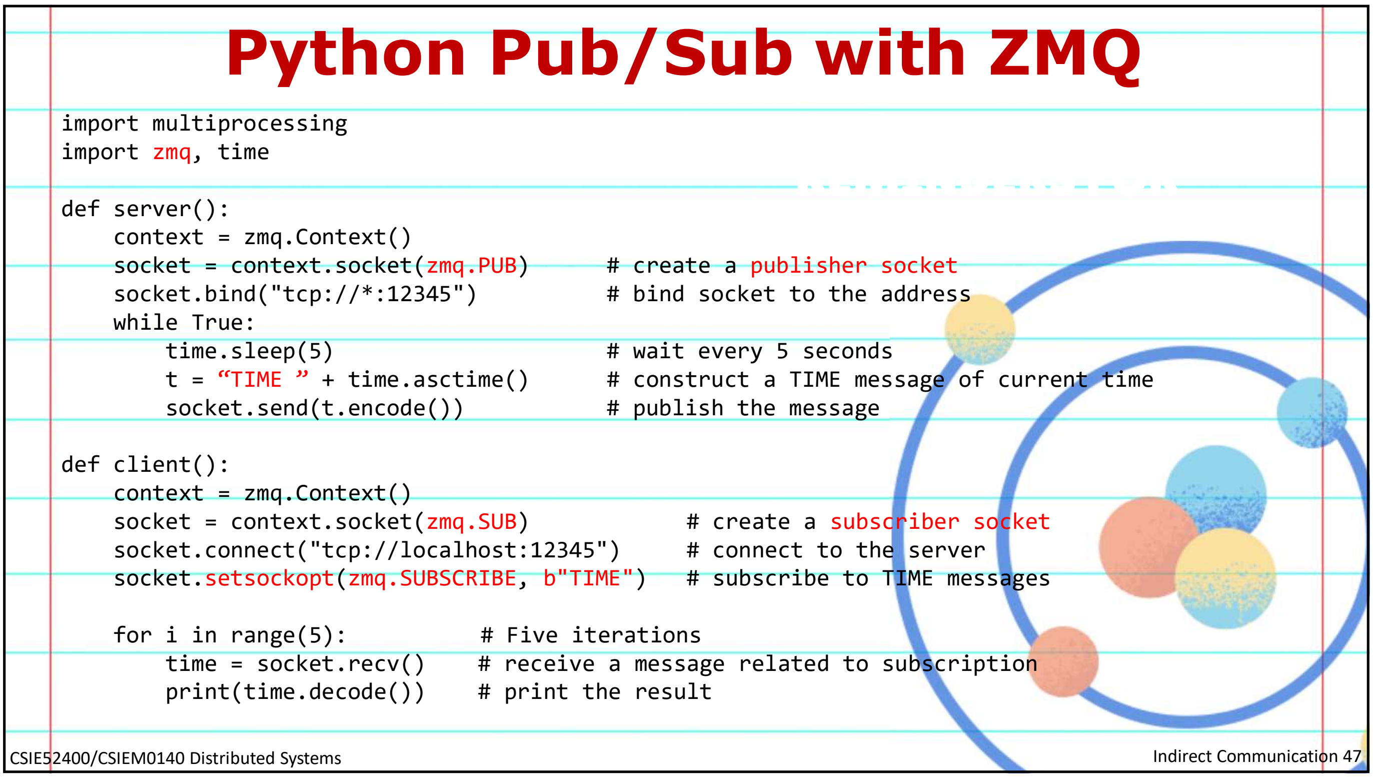
Rendezvous-based Routing

```
upon receive publish(event e) from node x at node k
  rvlist := EN(e); /* EN returns nodes responsible for matching e */
  if i in rvlist then begin
    matchlist :=match(e, subscriptions);
    send notify(e) to matchlist;
  end
  send publish(e) to rvlist - k;
upon receive subscribe(subscription s) from node x at node k
  rvlist := SN(s);  /* SN returns nodes responsible for s */
  if i in rvlist then
    add s to subscriptions;
  else
    send subscribe(s) to rvlist - k;
```



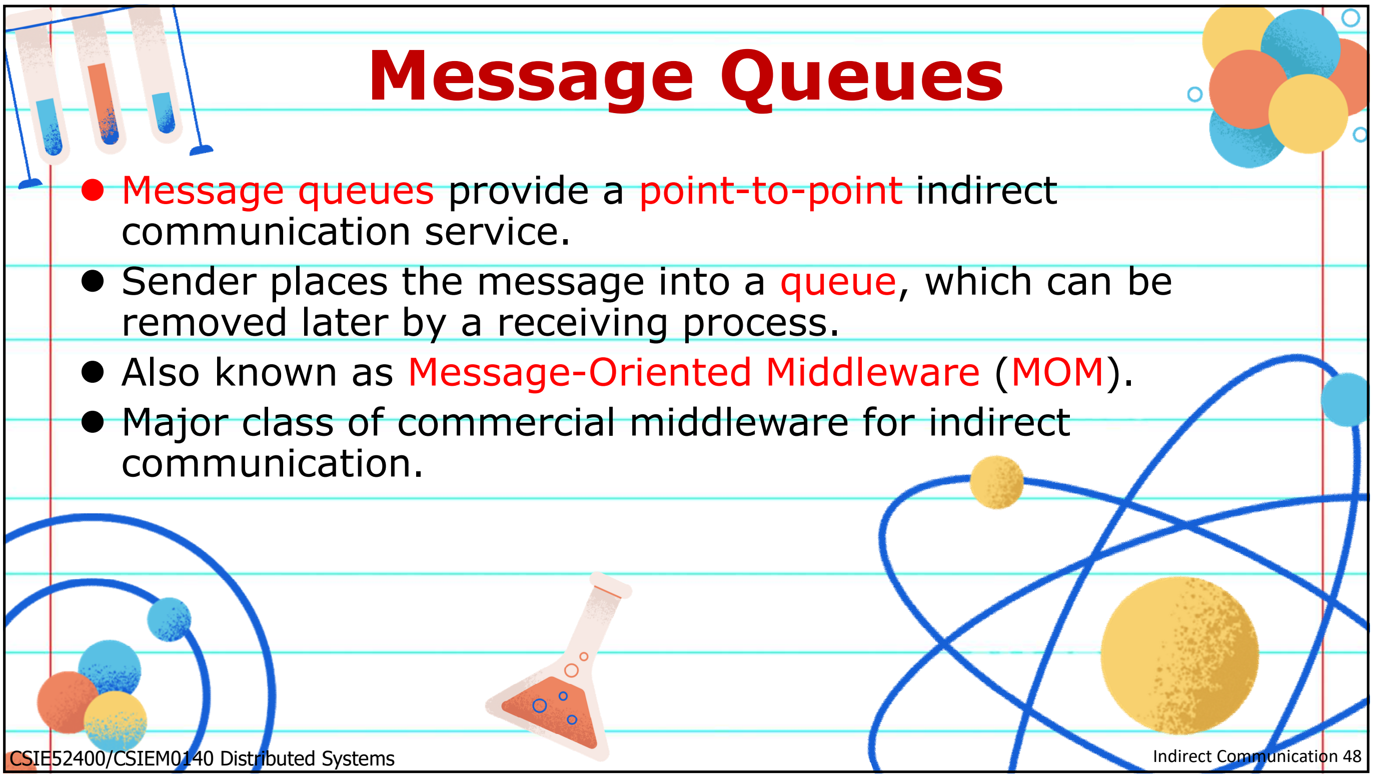
Python Pub/Sub Modules

- There are many modules for pub/sub in Python:
  - PyPubSub — An old but still useful module.
  - PyDispatcher — Another good module normally used with Django.
  - Redis Python Client — The popular Redis key-value store also supports pub/sub pattern.
  - ActiveMQ — The popular Apache multi-protocol message broker also supports pub/sub pattern.
  - Google Pub/Sub — A modern messaging framework supporting both messaging queues and distributed publish-subscribe models.



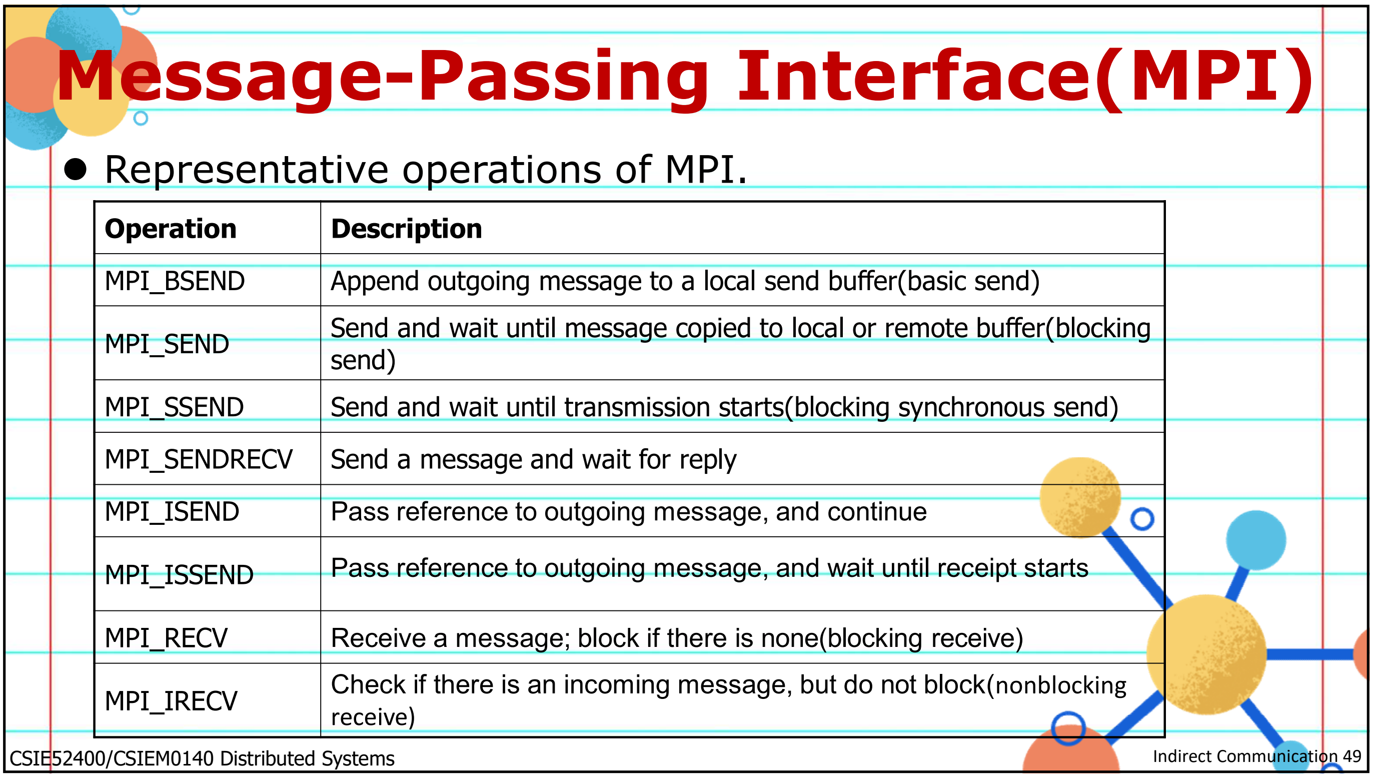
Python Pub/Sub with ZMQ

```
import multiprocessing
import zmq, time

def server():
    context = zmq.Context()
    socket = context.socket(zmq.PUB)        # create a publisher socket
    socket.bind("tcp://*:12345")            # bind socket to the address
    while True:
        time.sleep(5)                       # wait every 5 seconds
        t = “TIME ” + time.asctime()        # construct a TIME message of current time
        socket.send(t.encode())             # publish the message

def client():
    context = zmq.Context()
    socket = context.socket(zmq.SUB)              # create a subscriber socket
    socket.connect("tcp://localhost:12345")      # connect to the server
    socket.setsockopt(zmq.SUBSCRIBE, b"TIME")    # subscribe to TIME messages

    for i in range(5):            # Five iterations
        time = socket.recv()      # receive a message related to subscription
        print(time.decode())      # print the result
```



Message Queues

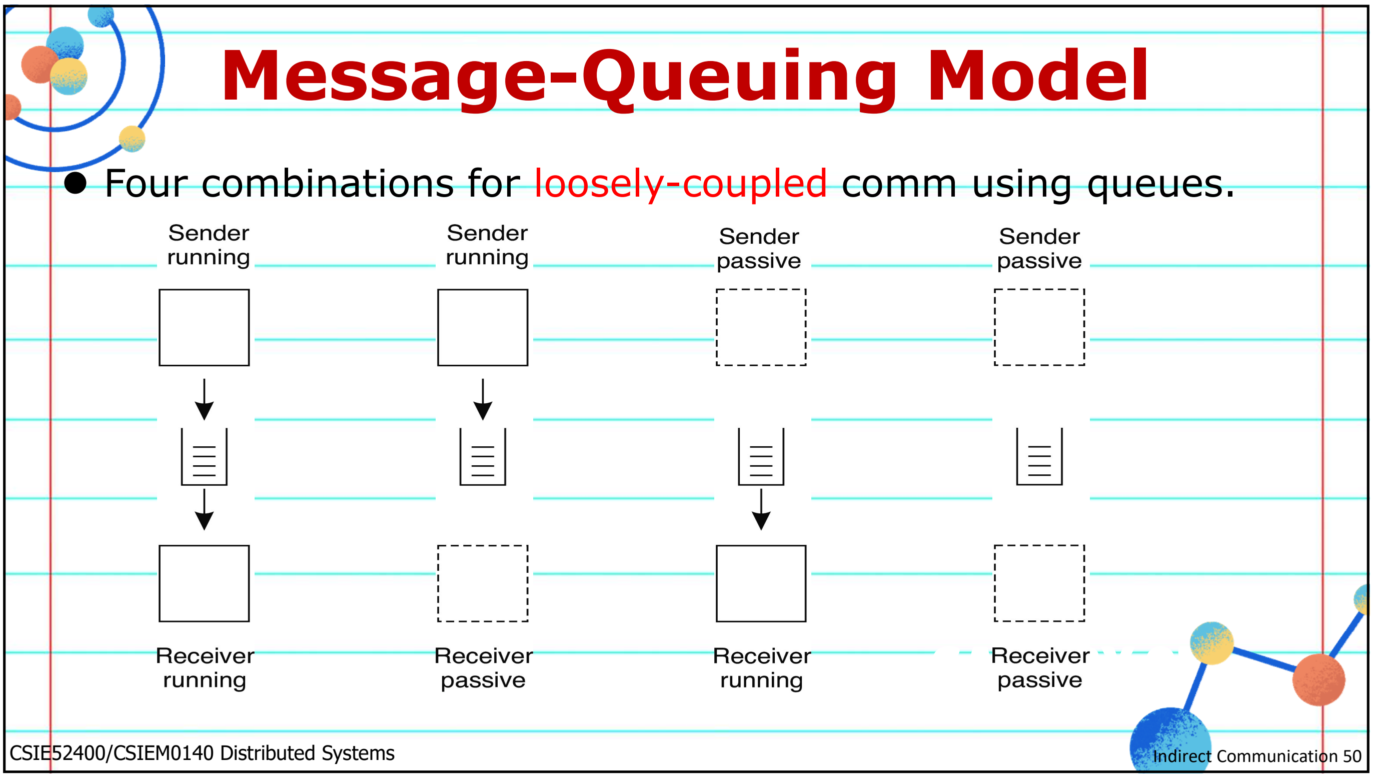
- Message queues provide a point-to-point indirect communication service.
- Sender places the message into a queue, which can be removed later by a receiving process.
- Also known as Message-Oriented Middleware (MOM).
- Major class of commercial middleware for indirect communication.



Message-Passing Interface(MPI)

- Representative operations of MPI.

Operation	Description
MPI_BSEND	Append outgoing message to a local send buffer(basic send)
MPI_SEND	Send and wait until message copied to local or remote buffer(blocking send)
MPI_SSEND	Send and wait until transmission starts(blocking synchronous send)
MPI_SENDRECV	Send a message and wait for reply
MPI_ISEND	Pass reference to outgoing message, and continue
MPI_ISSEND	Pass reference to outgoing message, and wait until receipt starts
MPI_RECV	Receive a message; block if there is none(blocking receive)
MPI_IRECV	Check if there is an incoming message, but do not block(nonblocking receive)



Message-Queuing Model

- Four combinations for loosely-coupled comm using queues.

Sender running

Receiver running

Sender running

Receiver passive

Sender passive

Receiver running

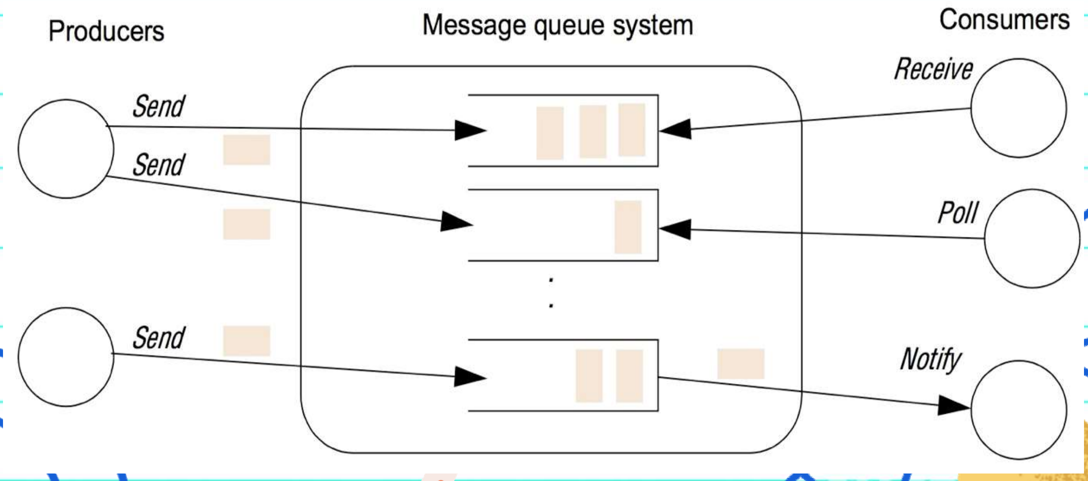
Sender passive

Receiver passive

Programming Model

- Communication through queues.
- Processes can send messages to a queue.
- Other processes can receive messages from that queue.
- Styles of receive
  - ➢ Blocking receive, block until msg available
  - ➢ Non-blocking receive (polling), check the queue for msg availability
  - ➢ Notify, issue a notification when a msg is available

Message-Queue Paradigm

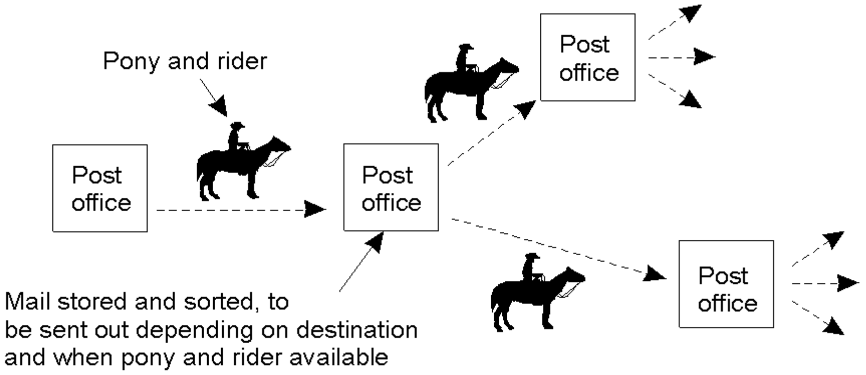


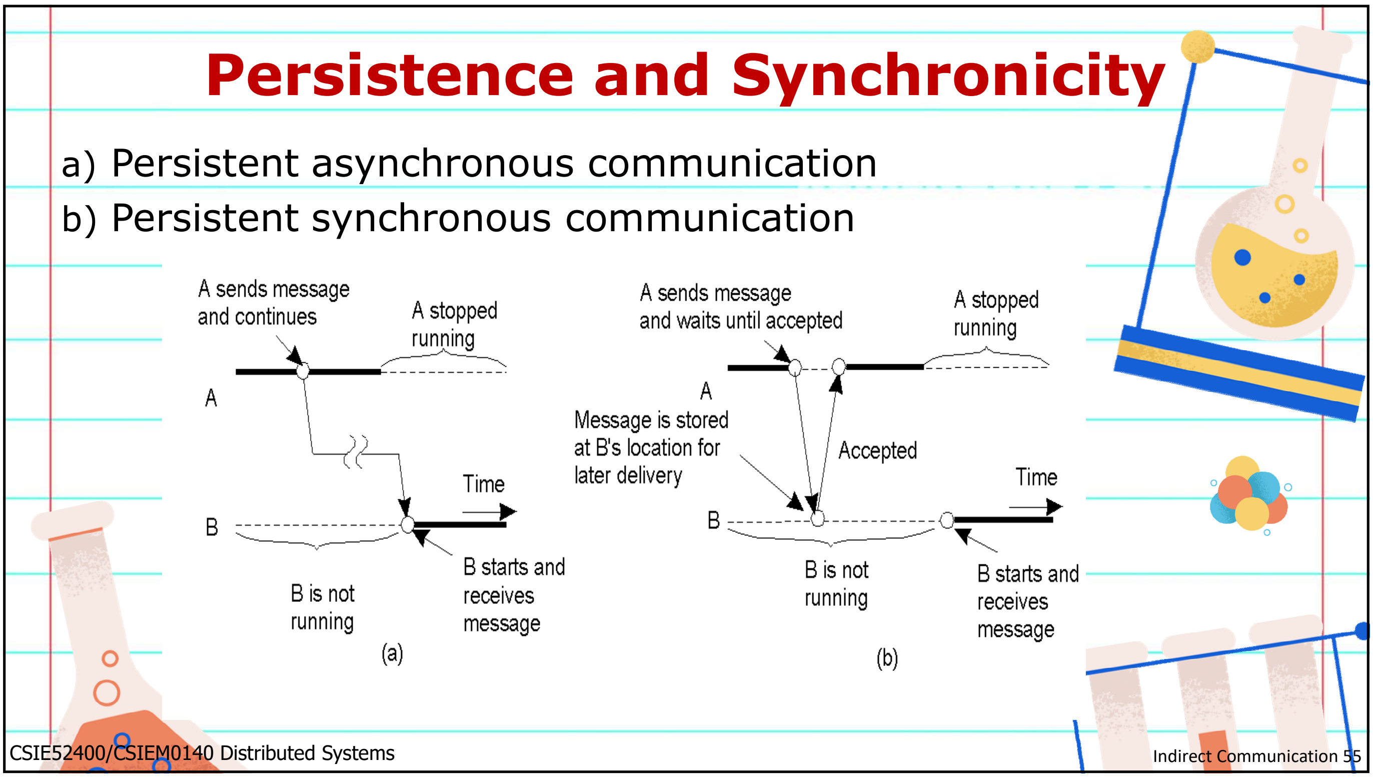
Messaging Characteristics

- Message communication can be
  - persistent – a submitted message is stored by the communication system as long as it takes to deliver.
  - transient – a message is stored only as the sending and receiving application are executing.
- Message communication can also be
  - asynchronous – sender continues immediately after it has submitted its message
  - synchronous – sender is blocked until its message is stored in a local buffer at the receiving host, or actually delivered

Persistence Communication

- Persistent communication of letters back in the days of the Pony Express.


Pony and rider
Post office
Post office
Post office
Post office
Mail stored and sorted, to be sent out depending on destination and when pony and rider available

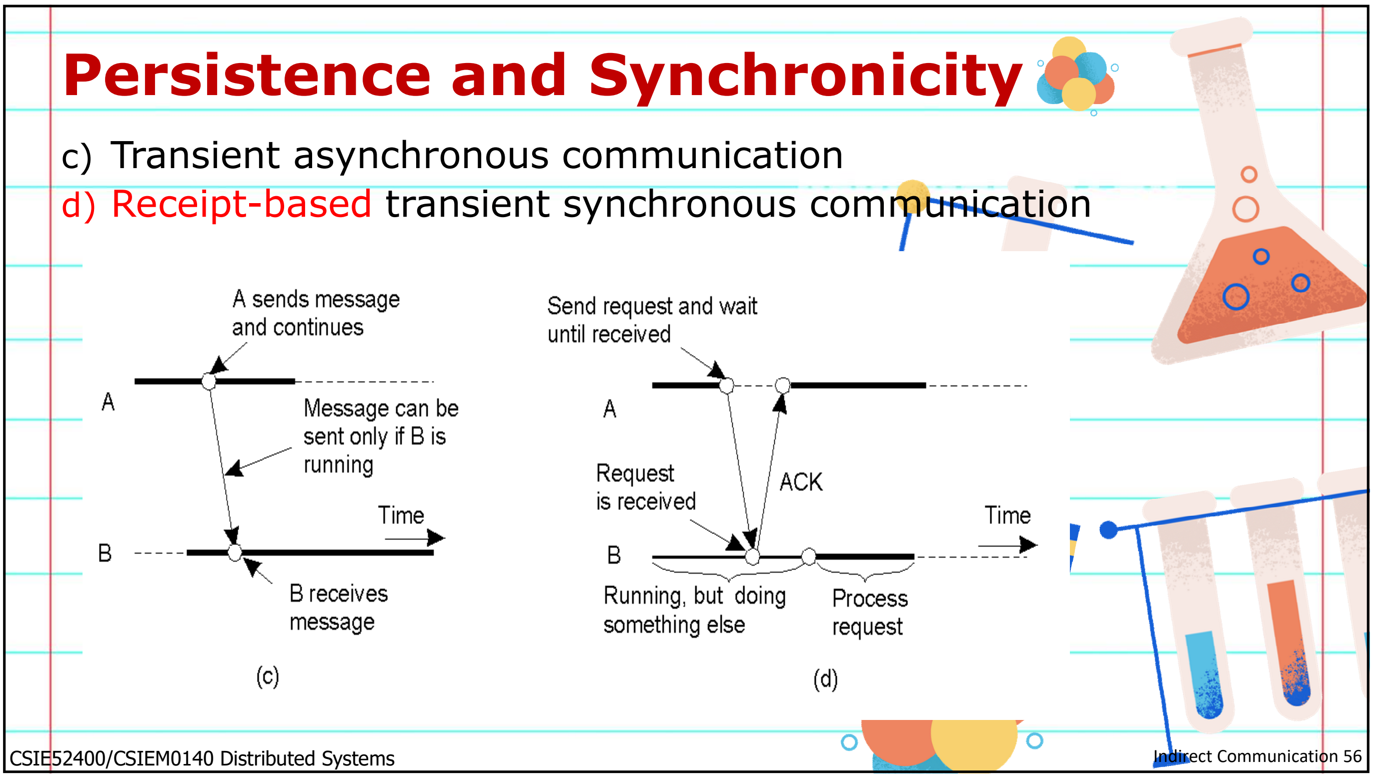


Persistence and Synchronicity

a) Persistent asynchronous communication

b) Persistent synchronous communication

A sends message and continues
A stopped running
A
B
Time
B is not running
B starts and receives message
(a)
A sends message and waits until accepted
A stopped running
A
Message is stored at B's location for later delivery
Accepted
Time
B
B is not running
B starts and receives message
(b)

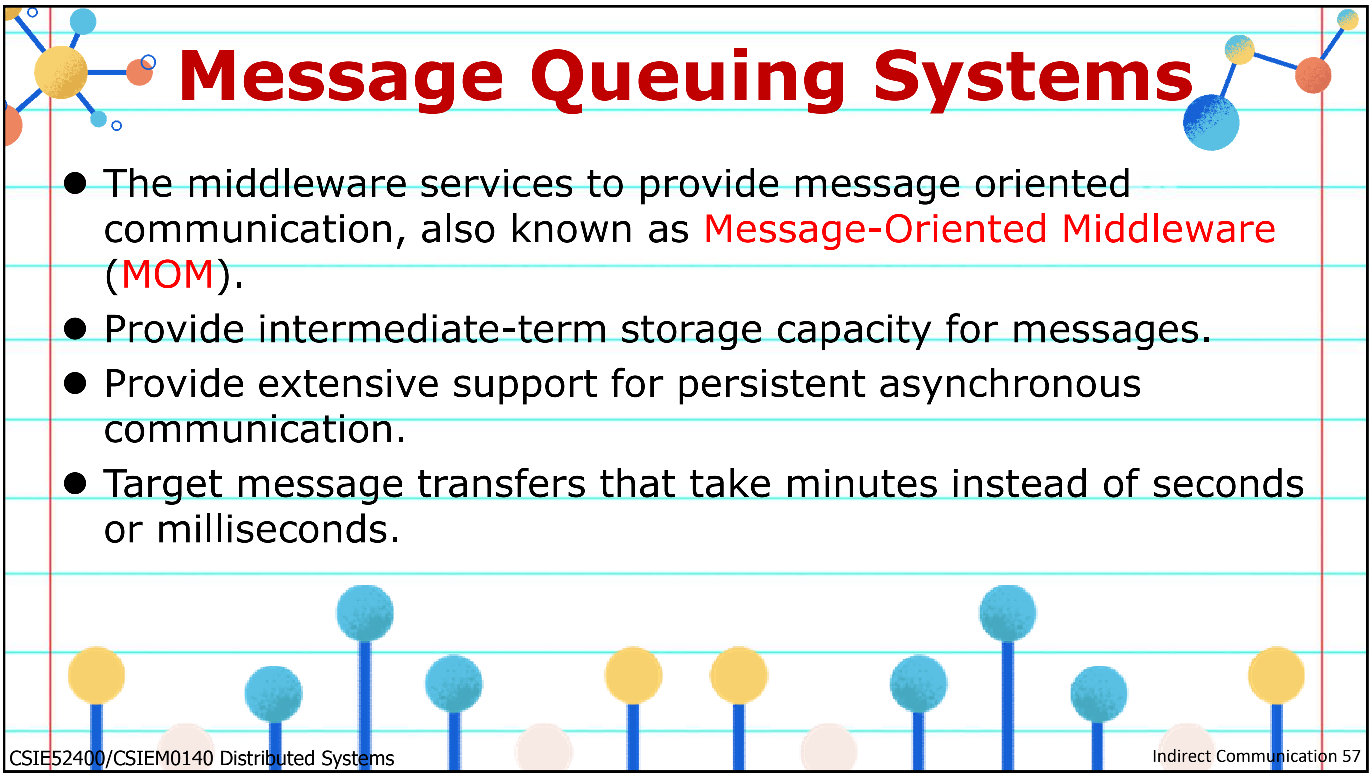


Persistence and Synchronicity

c) Transient asynchronous communication

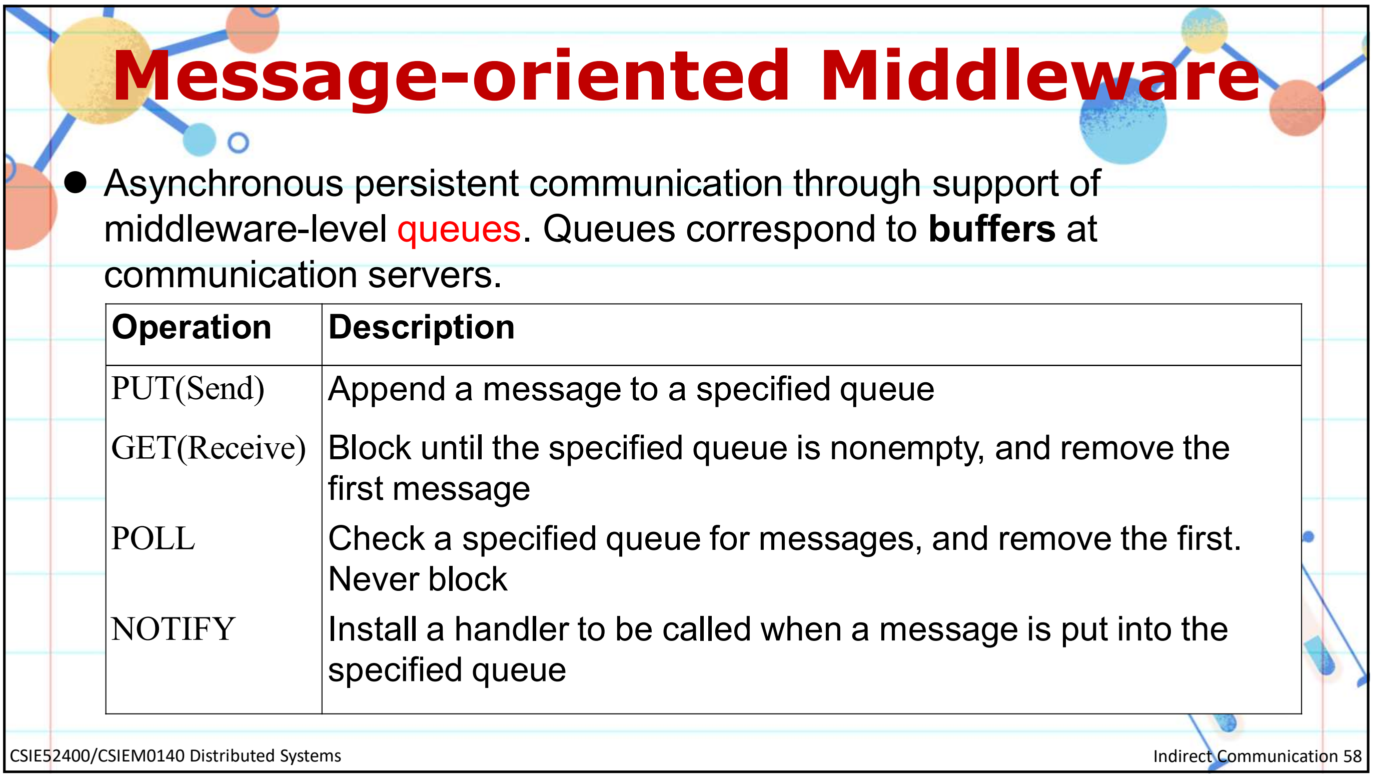
d) Receipt-based transient synchronous communication

A sends message and continues
A
Message can be sent only if B is running
Time
B
B receives message
(c)
Send request and wait until received
A
Request is received
ACK
Time
B
Running, but doing something else
Process request
(d)



Message Queuing Systems

- The middleware services to provide message oriented communication, also known as Message-Oriented Middleware (MOM).
- Provide intermediate-term storage capacity for messages.
- Provide extensive support for persistent asynchronous communication.
- Target message transfers that take minutes instead of seconds or milliseconds.



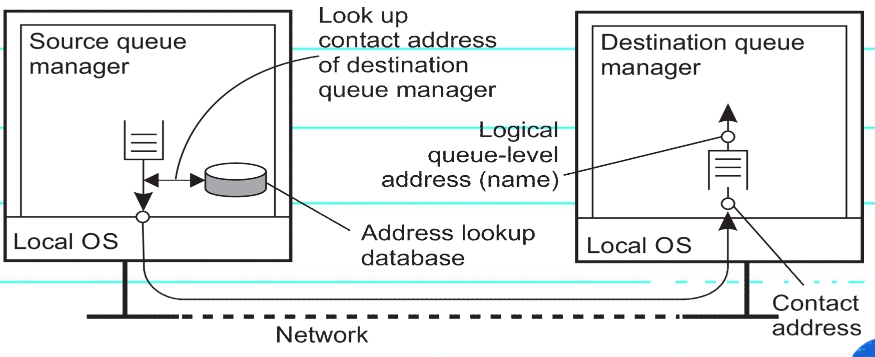
Message-oriented Middleware

- Asynchronous persistent communication through support of middleware-level queues. Queues correspond to **buffers** at communication servers.

Operation	**Description**
PUT(Send)	Append a message to a specified queue
GET(Receive)	Block until the specified queue is nonempty, and remove the first message
POLL	Check a specified queue for messages, and remove the first. Never block
NOTIFY	Install a handler to be called when a message is put into the specified queue

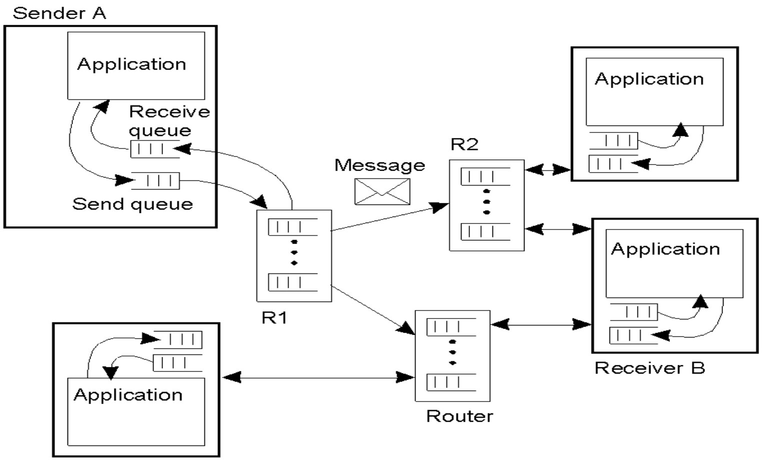
MOM Architecture – General Model

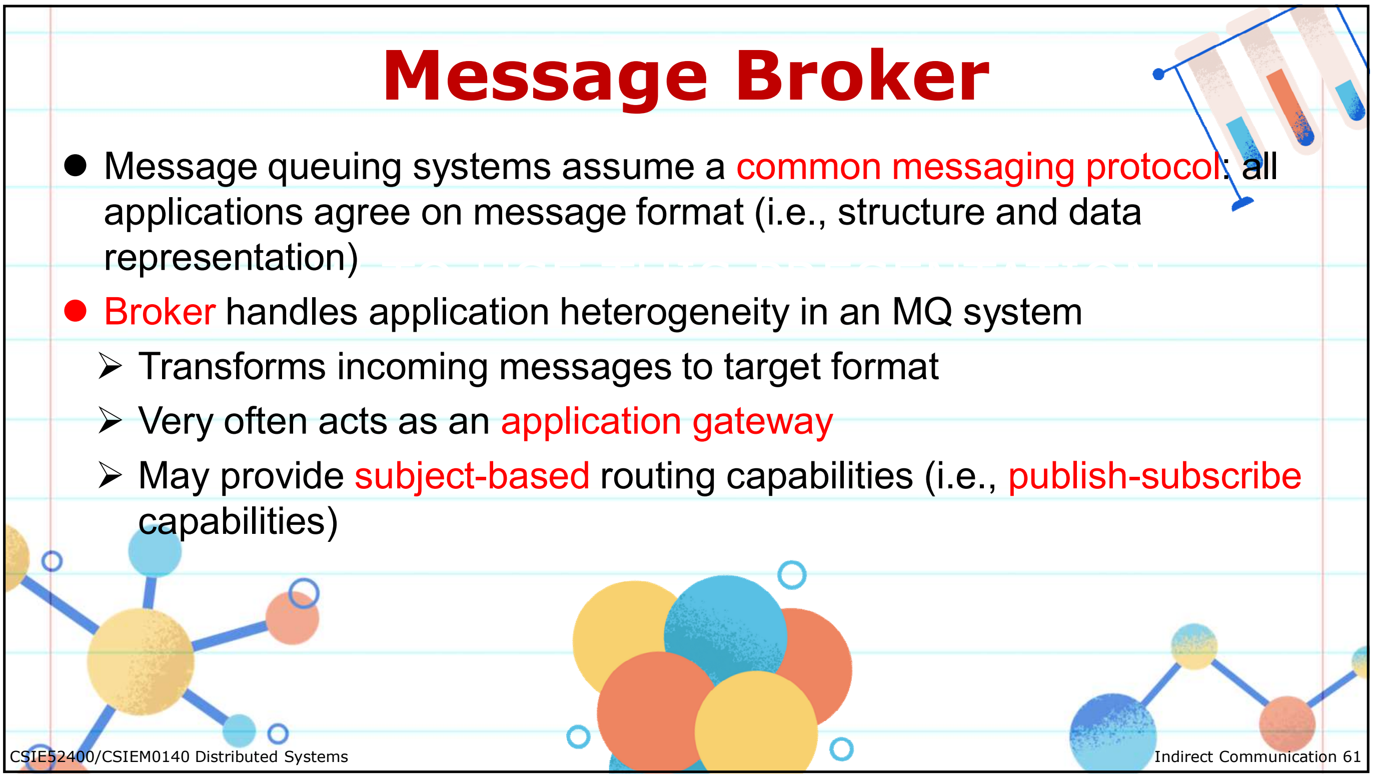
- Queues are managed by queue managers. An application can put messages only into a local queue. Getting a message is possible by extracting it from a local queue only ⇒ queue managers need to route messages.
- Routing


Source queue manager
Look up contact address of destination queue manager
Destination queue manager
Logical queue-level address (name)
Local OS
Address lookup database
Local OS
Contact address
Network

MOM Architecture - Routers

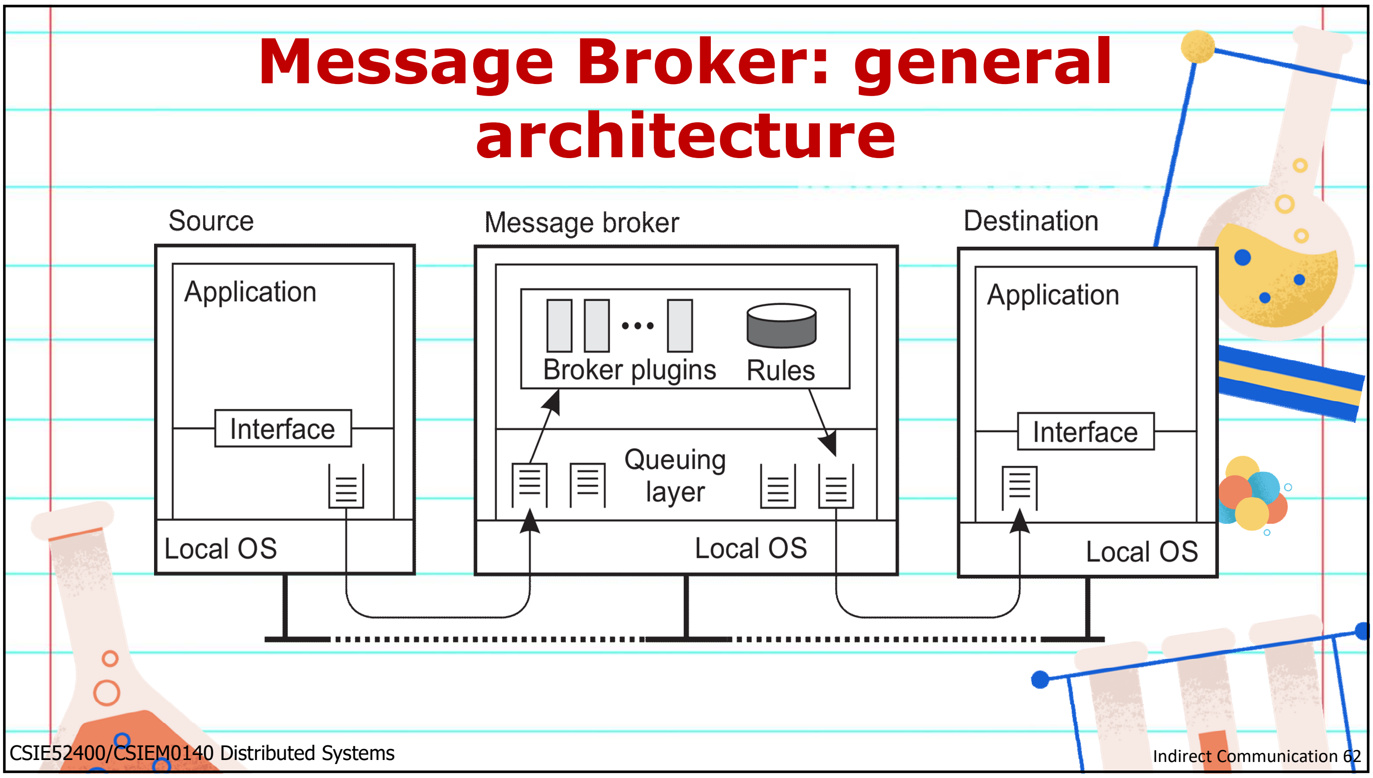
- The general organization of a message-queuing system with routers.



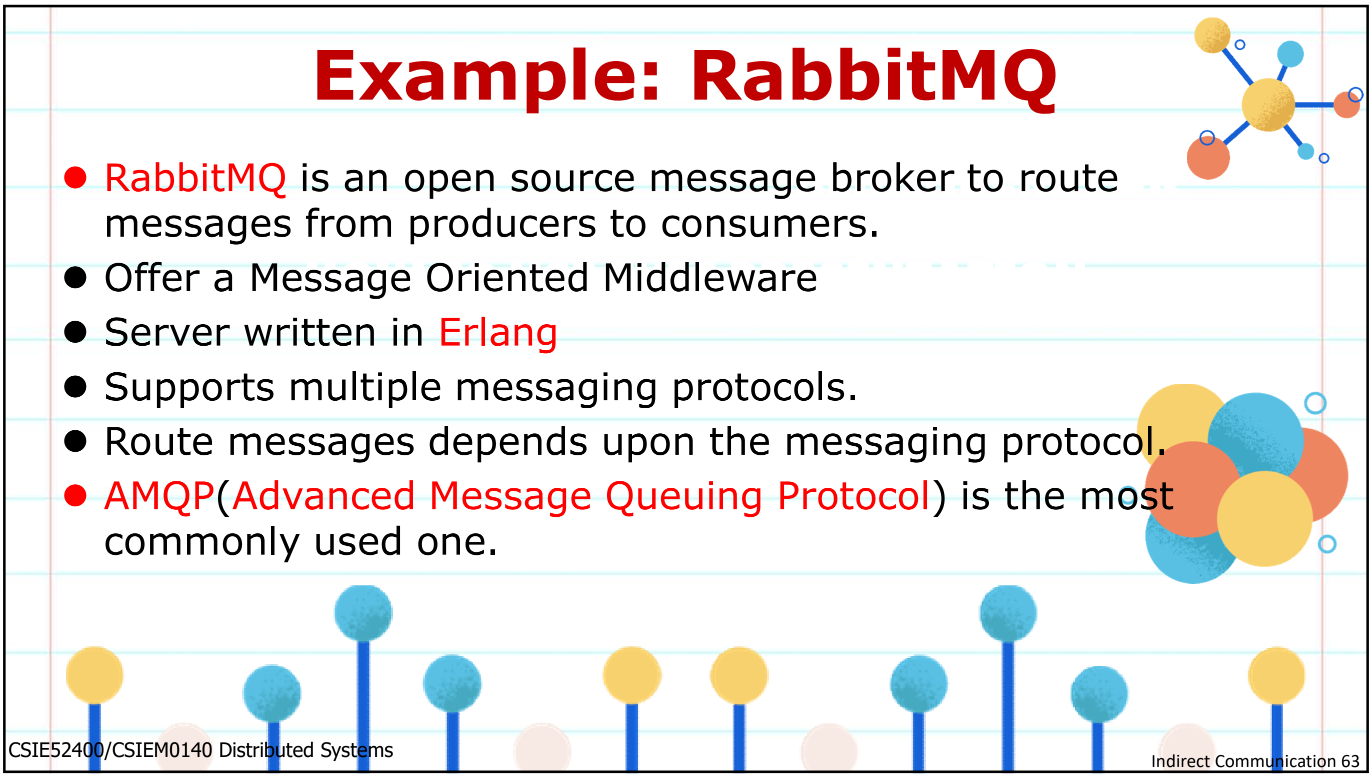


Message Broker

- Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)
- Broker handles application heterogeneity in an MQ system
  - Transforms incoming messages to target format
  - Very often acts as an application gateway
  - May provide subject-based routing capabilities (i.e., publish-subscribe capabilities)

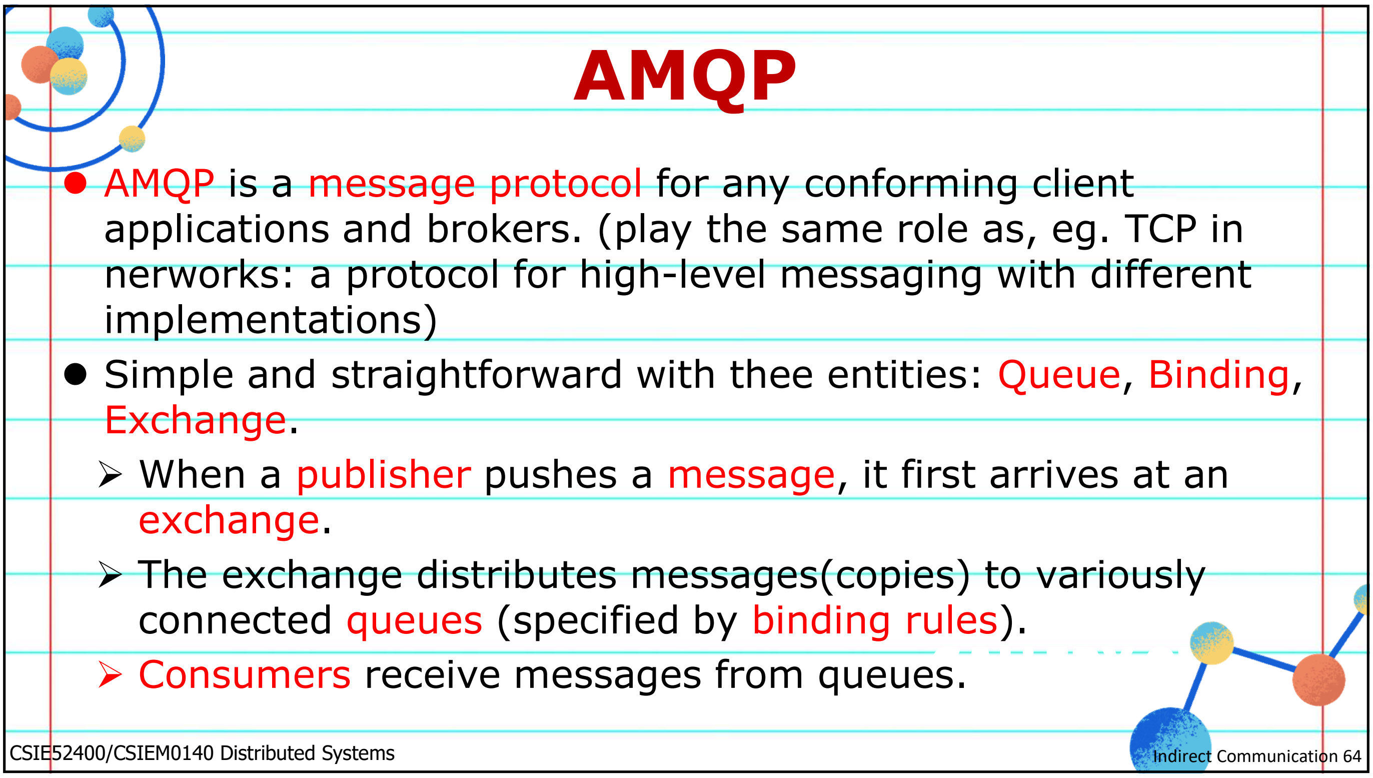


Message Broker: general architecture



Example: RabbitMQ

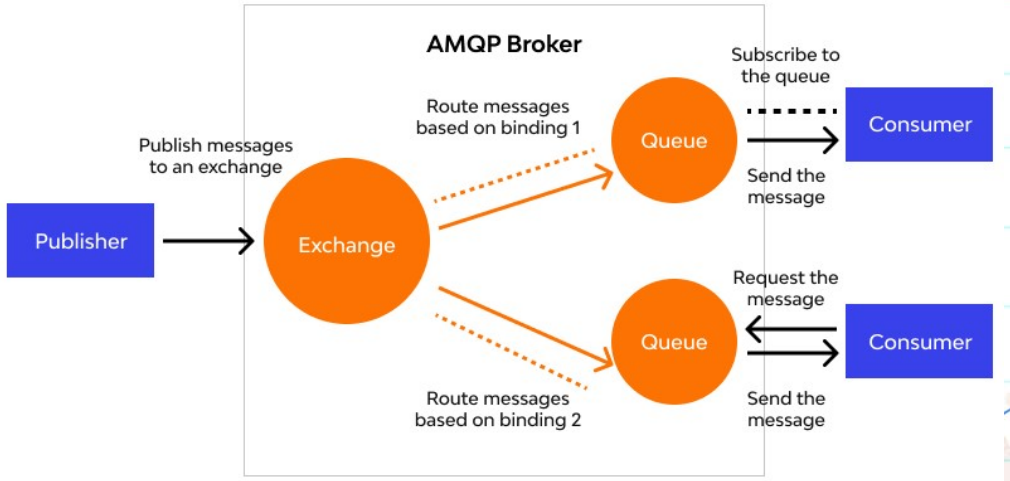
- RabbitMQ is an open source message broker to route messages from producers to consumers.
- Offer a Message Oriented Middleware
- Server written in Erlang
- Supports multiple messaging protocols.
- Route messages depends upon the messaging protocol.
- AMQP(Advanced Message Queuing Protocol) is the most commonly used one.



AMQP

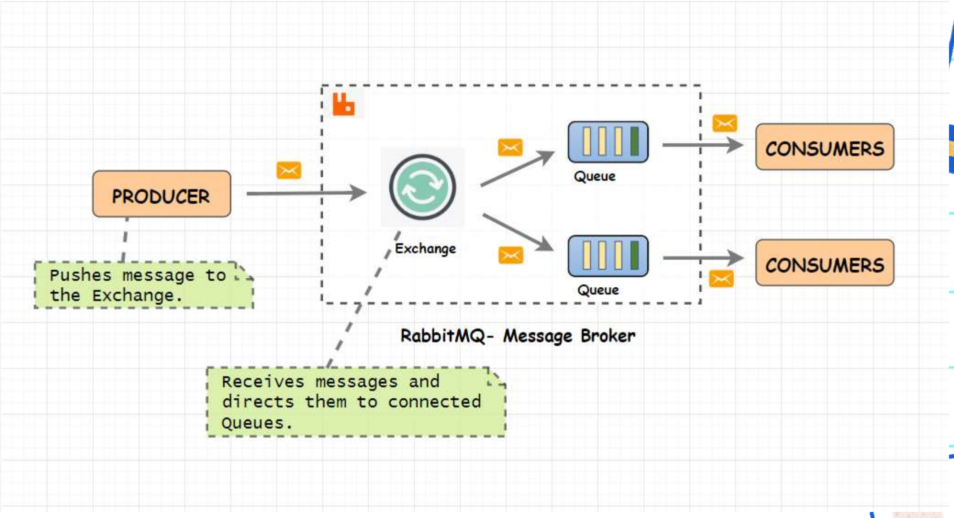
- AMQP is a message protocol for any conforming client applications and brokers. (play the same role as, eg. TCP in nerworks: a protocol for high-level messaging with different implementations)
- Simple and straightforward with thee entities: Queue, Binding, Exchange.
  - ➢ When a publisher pushes a message, it first arrives at an exchange.
  - ➢ The exchange distributes messages(copies) to variously connected queues (specified by binding rules).
  - ➢ Consumers receive messages from queues.

AMQP Model



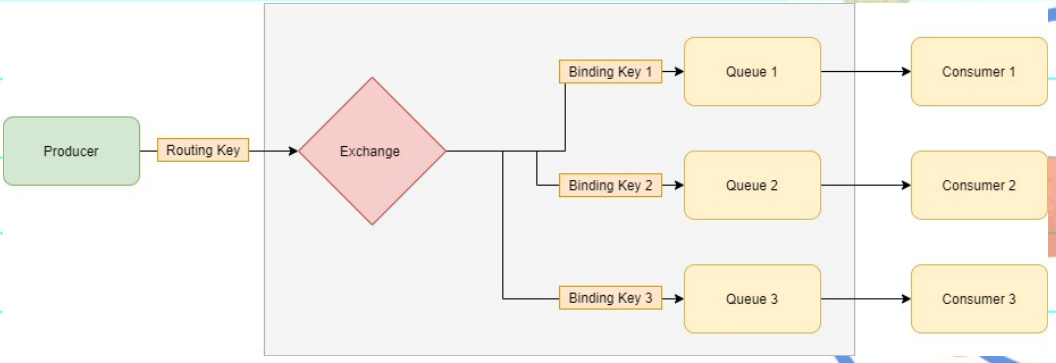
RabbitMQ and AMQP

- RabbitMQ implements AMQP (among others)


PRODUCER
Pushes message to the Exchange.
Exchange
Queue
Queue
CONSUMERS
CONSUMERS
RabbitMQ- Message Broker
Receives messages and directs them to connected Queues.

RabbitMQ Architecture

- RabbitMQ employs a flexible mechanism to implement AMQP with routing key and binding key.
- Different types of key matching allow different types of exchanges.

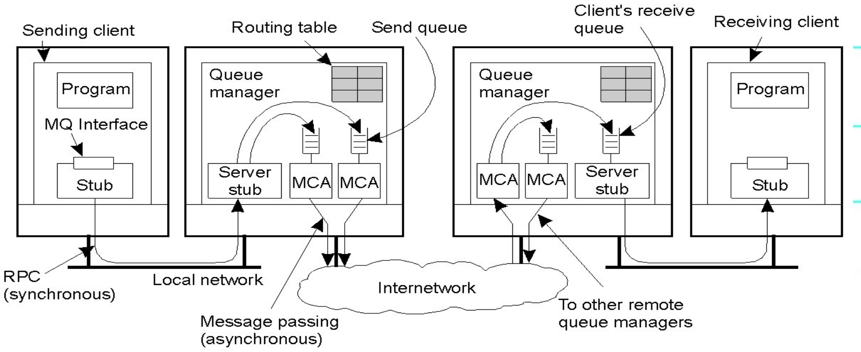


RabbitMQ Exchanges

- Many types of exchanges are available:
  - ➢ Direct exchange
  - ➢ Topic exchange
  - ➢ Fanout exchange
  - ➢ Headers exchange
  - ➢ Default exchange
  - ➢ Dead Letter exchange
  - ➢ (https://hevodata.com/learn/rabbitmq-exchange-type/)

Example: IBM MQ

- IBM's MOM: MQSeries(1993) → WebSphere MQ(2002) → IBM MQ(2014[8.0], 2016[9.0], 2022[9.3]).
- Queue managers are connected by unidirectional and reliable message channels managed by message channel agent (MCA).



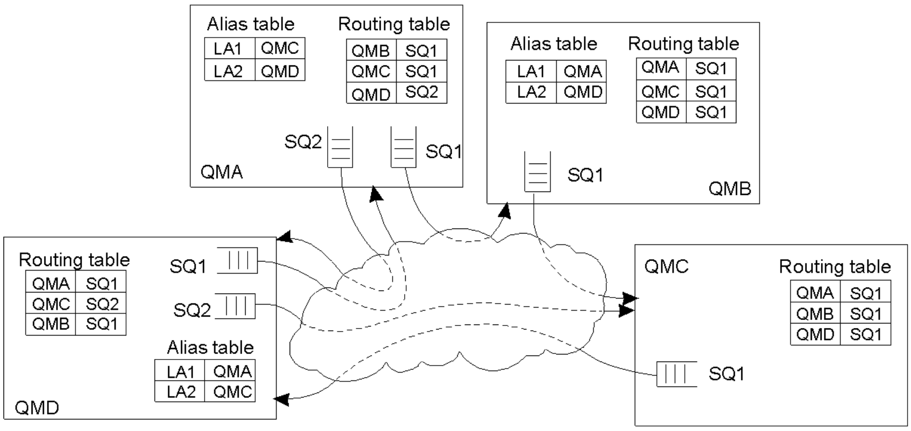
Channels

- Some attributes associated with message channel agents.

Attribute	Description
Transport type	Determines the transport protocol to be used
FIFO delivery	Indicates that messages are to be delivered in the order they are sent
Message length	Maximum length of a single message
Setup retry count	Specifies maximum number of retries to start up the remote MCA
Delivery retries	Maximum times MCA will try to put received message into queue

Message Transfer (1)

- The general organization of an message queuing network using routing tables and aliases.



QMA

Alias table	
LA1	QMC
LA2	QMD

Routing table	
QMB	SQ1
QMC	SQ1
QMD	SQ2

QMB

Alias table	
LA1	QMA
LA2	QMD

Routing table	
QMA	SQ1
QMC	SQ1
QMD	SQ1

QMD

Routing table	
QMA	SQ1
QMC	SQ2
QMB	SQ1

Alias table	
LA1	QMA
LA2	QMC

QMC

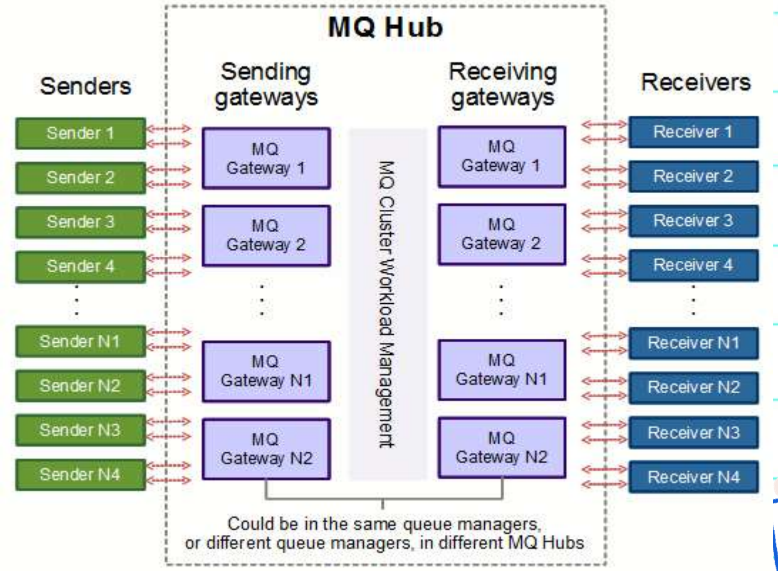
Routing table	
QMA	SQ1
QMB	SQ1
QMD	SQ1

Message Transfer (2)

- Examples of primitives available in the Message Queue Interface (MQI)

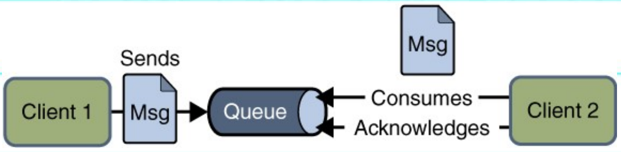
Primitive	Description
MQopen	Open a (possibly remote) queue
MQclose	Close a queue
MQput	Put a message into an opened queue
MQget	Get a message from a (local) queue

IBM MQ Hub (w/o local queue manager)

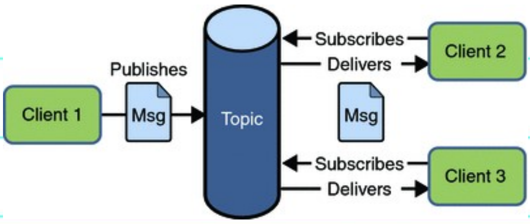


Java Message Service(JMS)

- Java MOM API for passing messages between clients.
- JMS point-to-point messaging domain



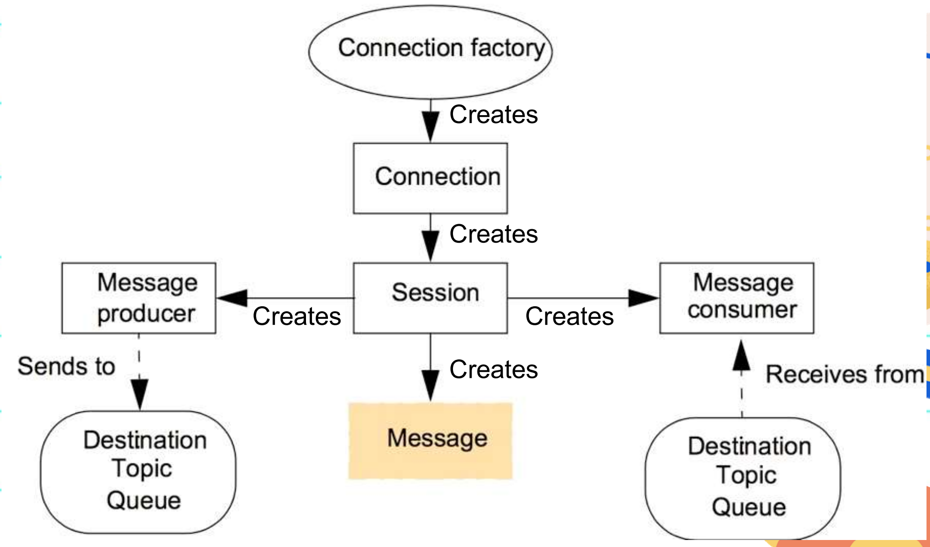
- JMS publish/subscribe messaging domain



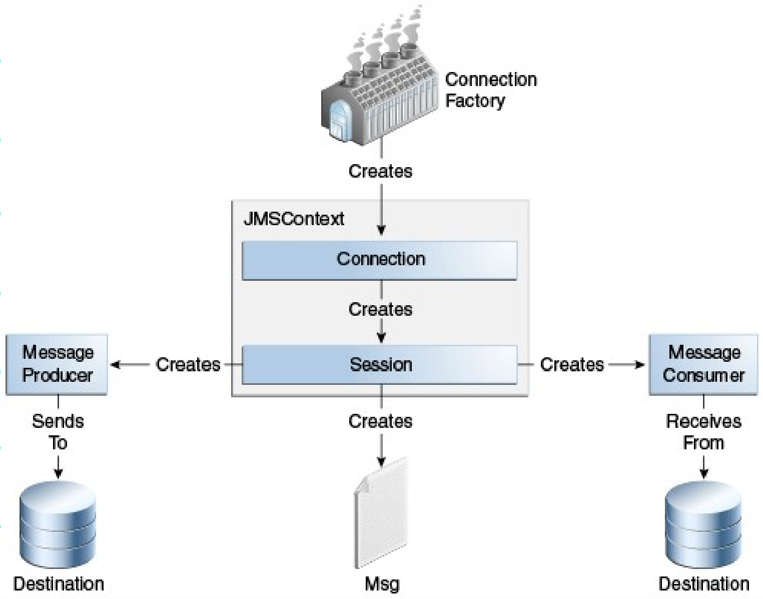
JMS Participating Objects

- JMS Administered Objects: Connection Factory, Connection, Session, JMSContext(replace C&S)
- JMS producer: creates and produces messages
- JMS consumer: receives and consumes messages
- JMS client: a producer or consumer
- JMS provider: a system that implement the JMS specification
- JMS message: the message object
- JMS destination: an object supporting JMS (either a JMS topic or a JMS queue)

JMS Programming Model(old)



JMS Programming Model(new)



JMS Administered Objects

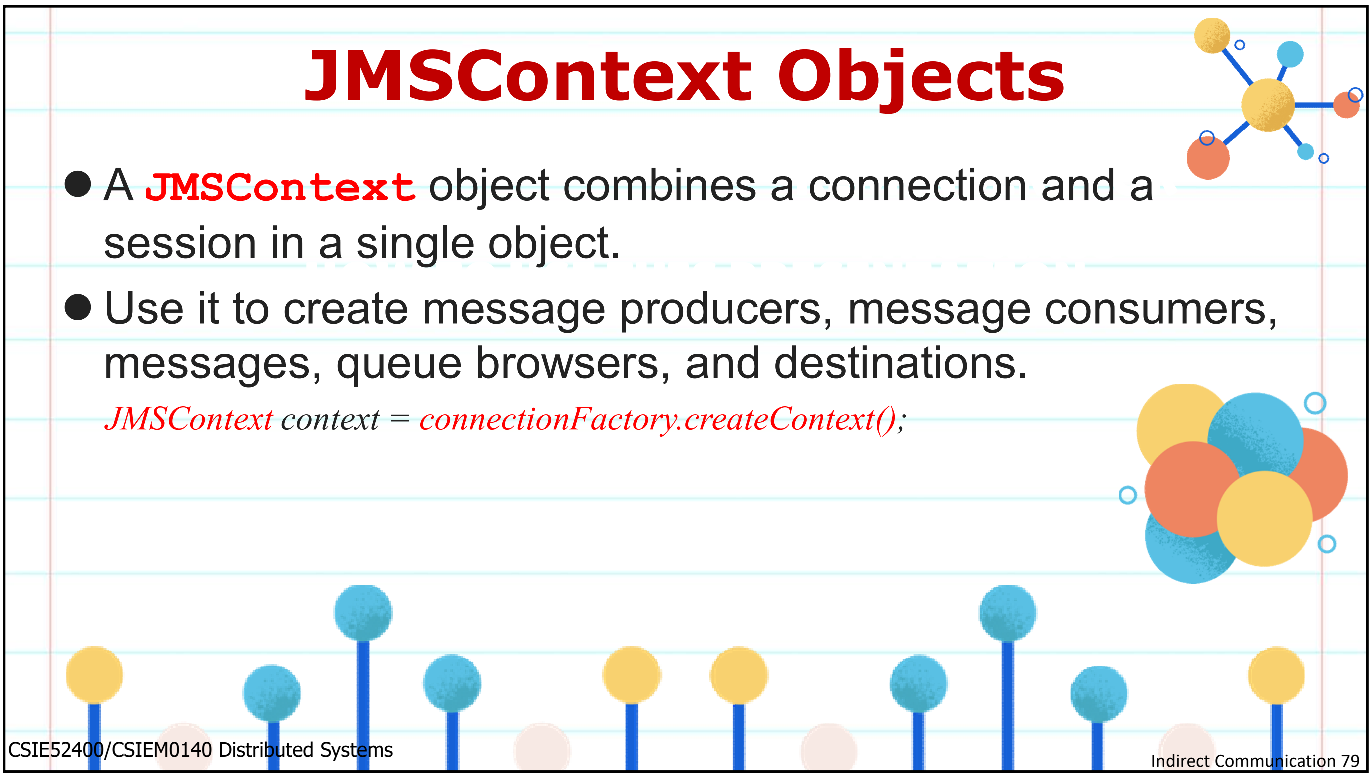
- JMS Connection Factories

  @Resource(lookup = "jms/ConnectionFactory")
  private static ConnectionFactory connectionFactory;

- JMS Destinations

  @Resource(lookup = "jms/Queue")
  private static Queue queue;

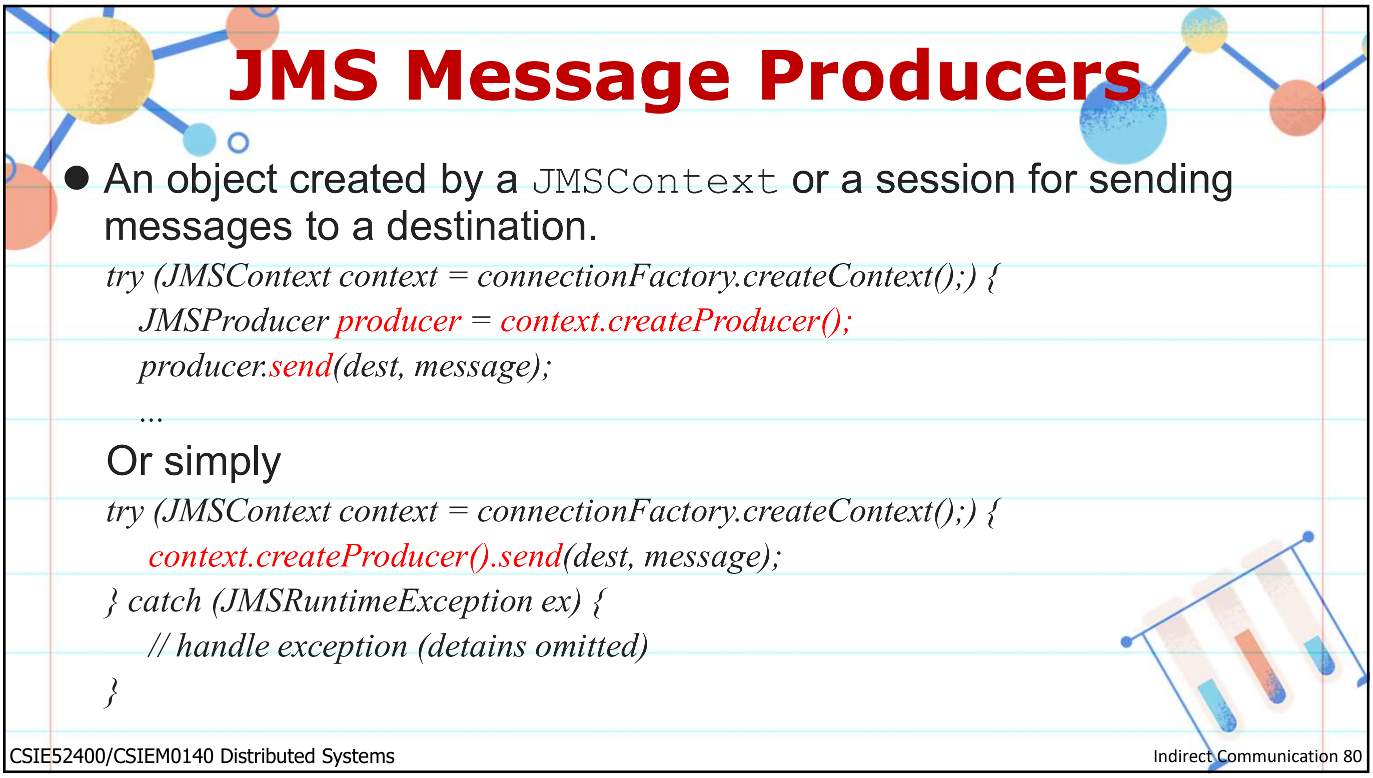
  @Resource(lookup = "jms/Topic")
  private static Topic topic;



JMSContext Objects

- A **JMSContext** object combines a connection and a session in a single object.
- Use it to create message producers, message consumers, messages, queue browsers, and destinations.

```
JMSContext context = connectionFactory.createContext();
```



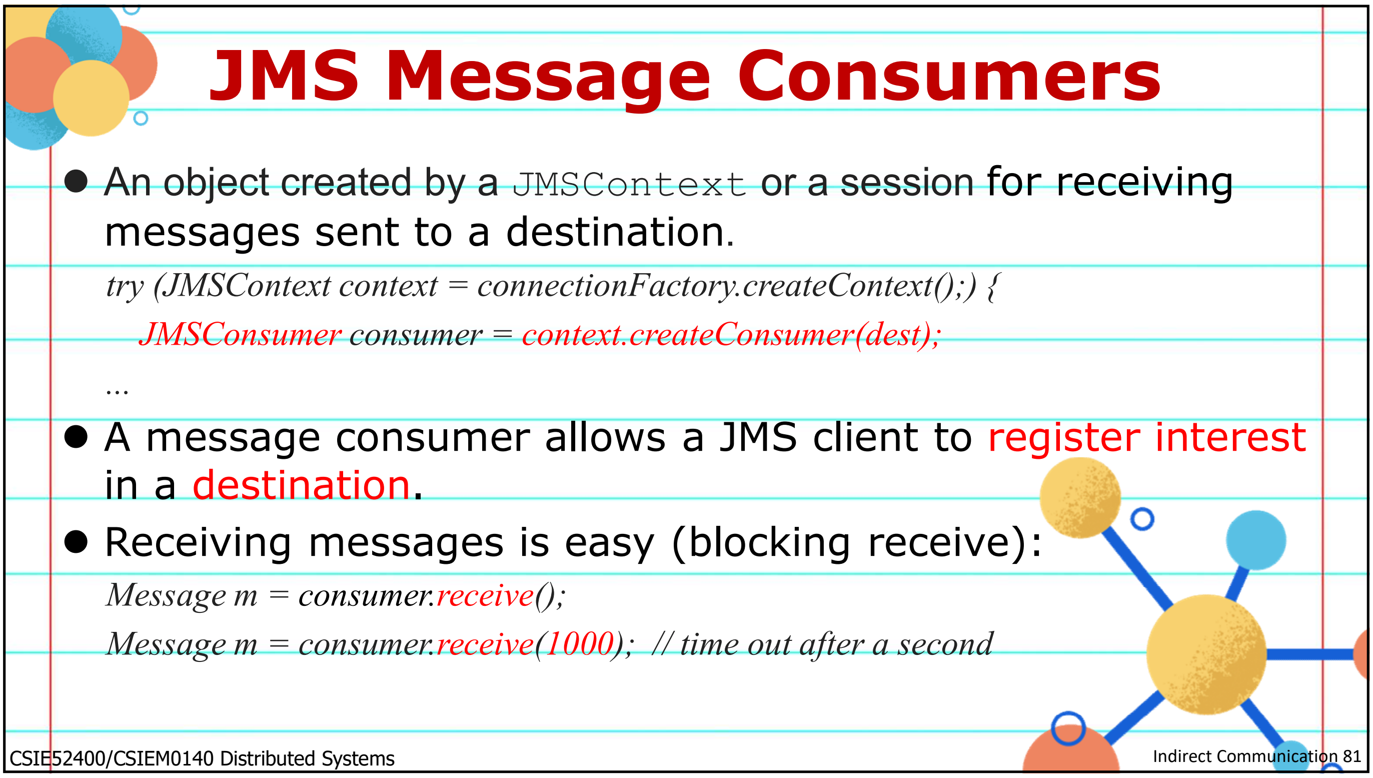
JMS Message Producers

- An object created by a `JMSContext` or a session for sending messages to a destination.

```
try (JMSContext context = connectionFactory.createContext();) {
   JMSProducer producer = context.createProducer();
   producer.send(dest, message);
   ...
```

Or simply

```
try (JMSContext context = connectionFactory.createContext();) {
    context.createProducer().send(dest, message);
} catch (JMSRuntimeException ex) {
    // handle exception (detains omitted)
}
```



JMS Message Consumers

- An object created by a `JMSContext` or a session for receiving messages sent to a destination.

```
try (JMSContext context = connectionFactory.createContext();) {
   JMSConsumer consumer = context.createConsumer(dest);
…
```

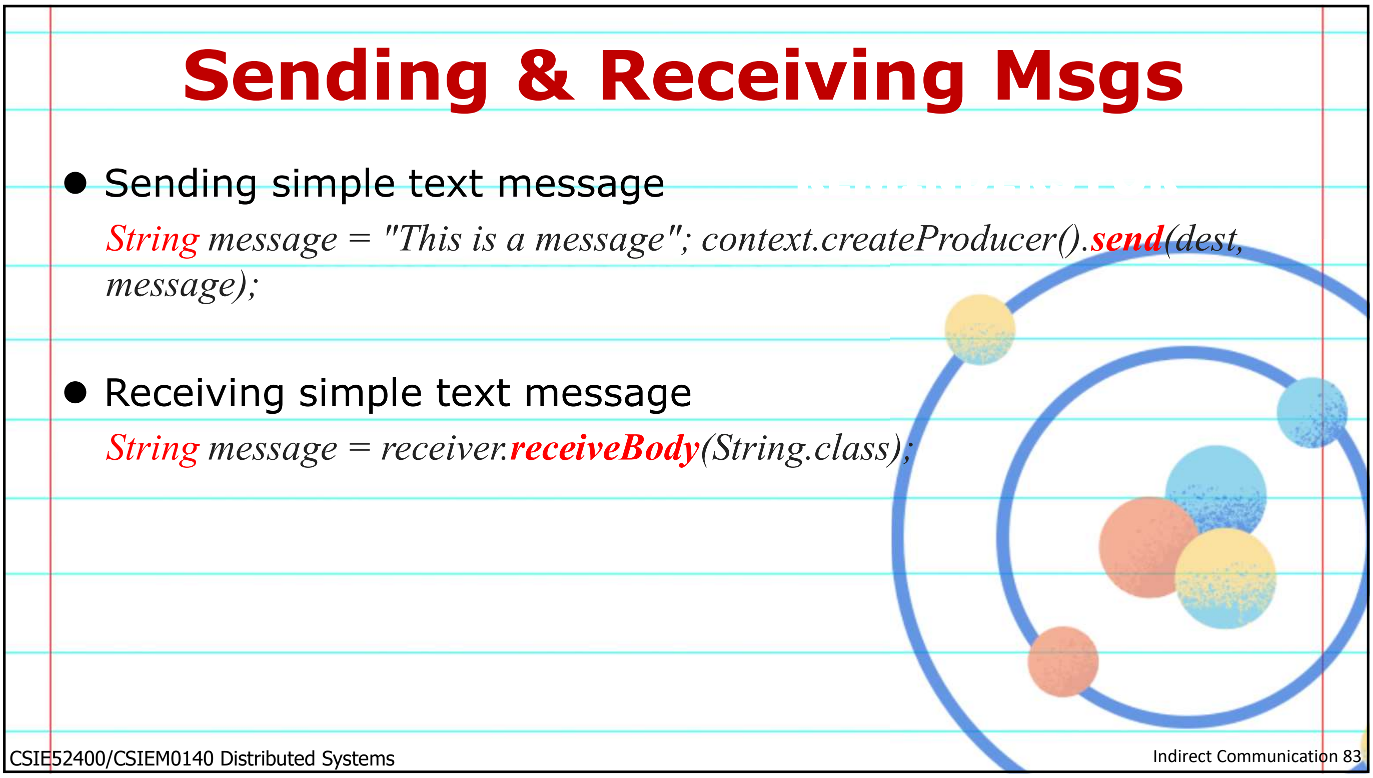
- A message consumer allows a JMS client to register interest in a destination.
- Receiving messages is easy (blocking receive):

```
Message m = consumer.receive();
Message m = consumer.receive(1000);  // time out after a second
```

JMS Message Listeners

- An object that acts as an asynchronous event handler for messages.
- This object implements the **`MessageListener`** interface, which contains one method, **`onMessage`**.
- In the `onMessage` method, you define the actions to be taken when a message arrives.

```
Listener myListener = new Listener(); consumer.setMessageListener(myListener);
```



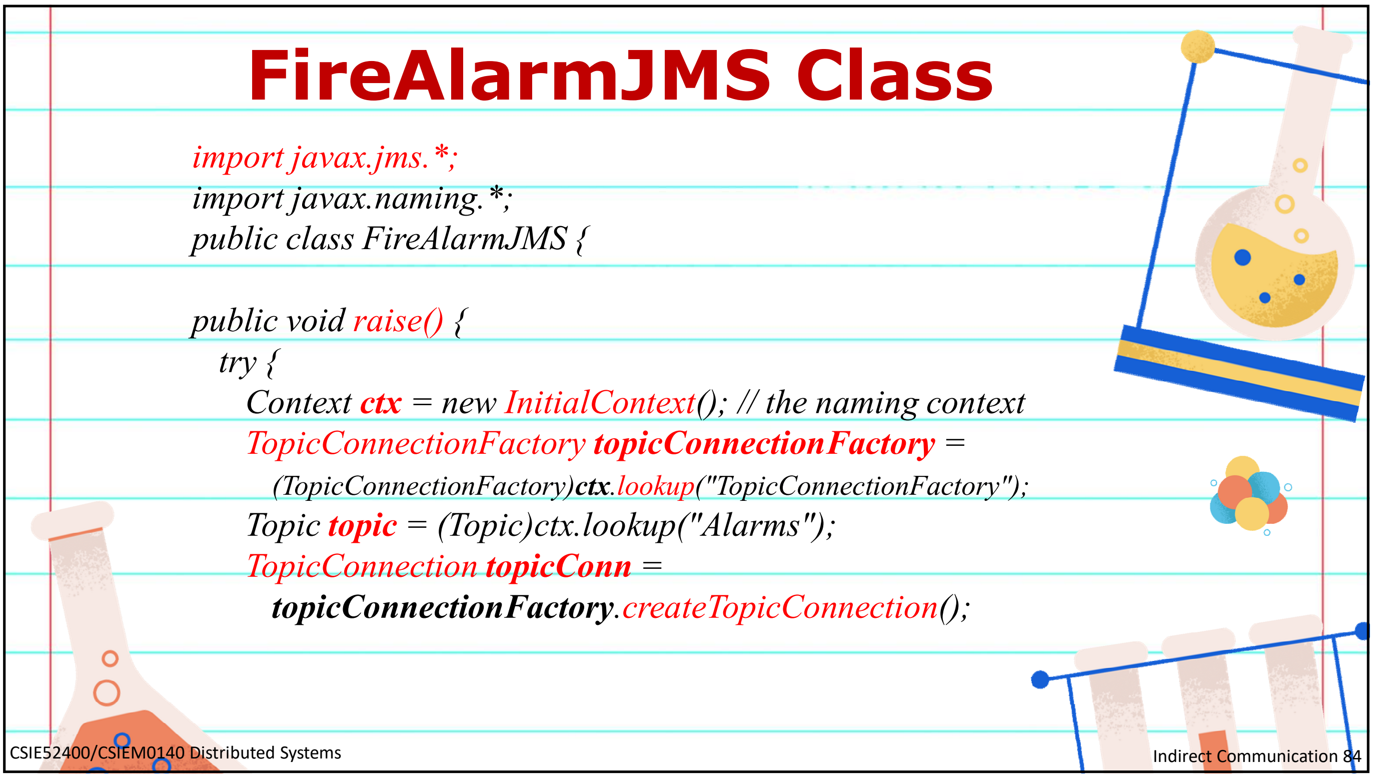
Sending & Receiving Msgs

- Sending simple text message

```
String message = "This is a message"; context.createProducer().send(dest,
message);
```

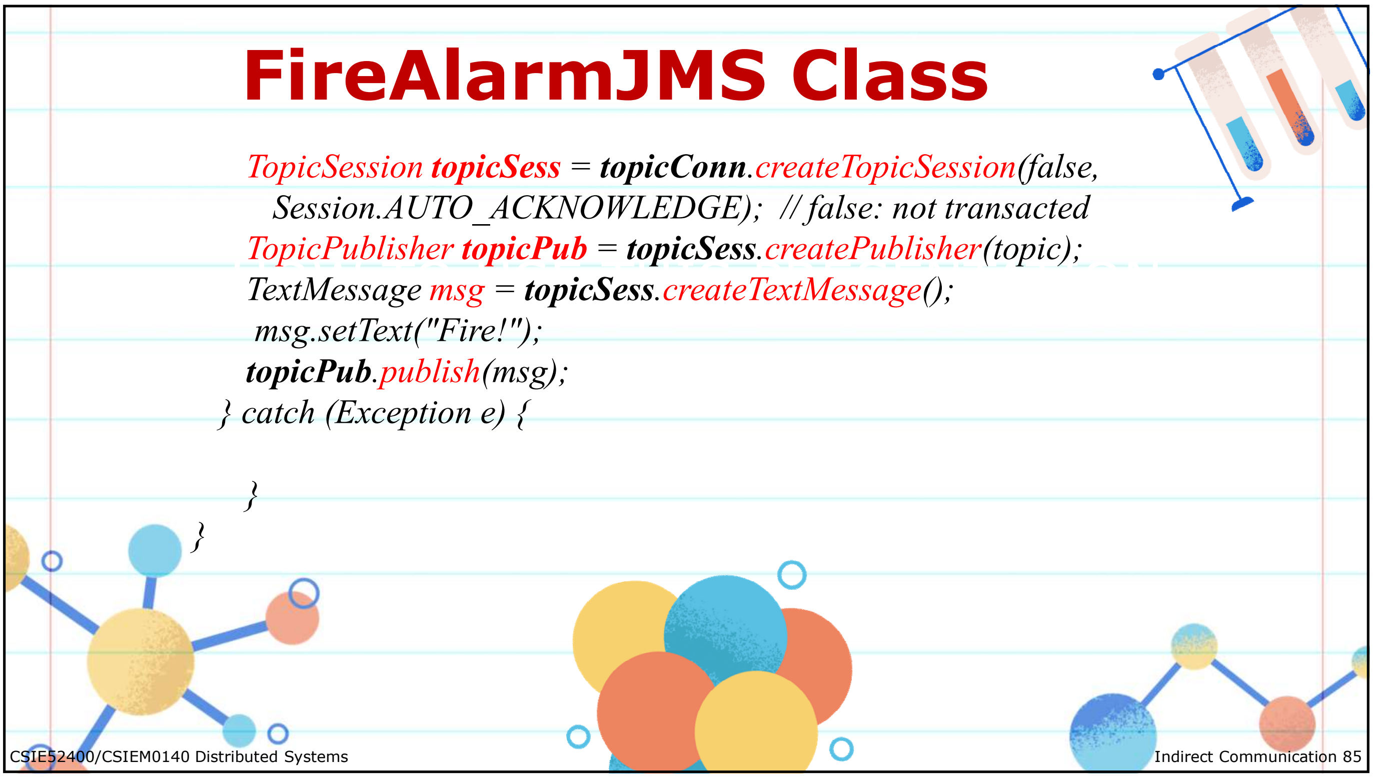
- Receiving simple text message

```
String message = receiver.receiveBody(String.class);
```



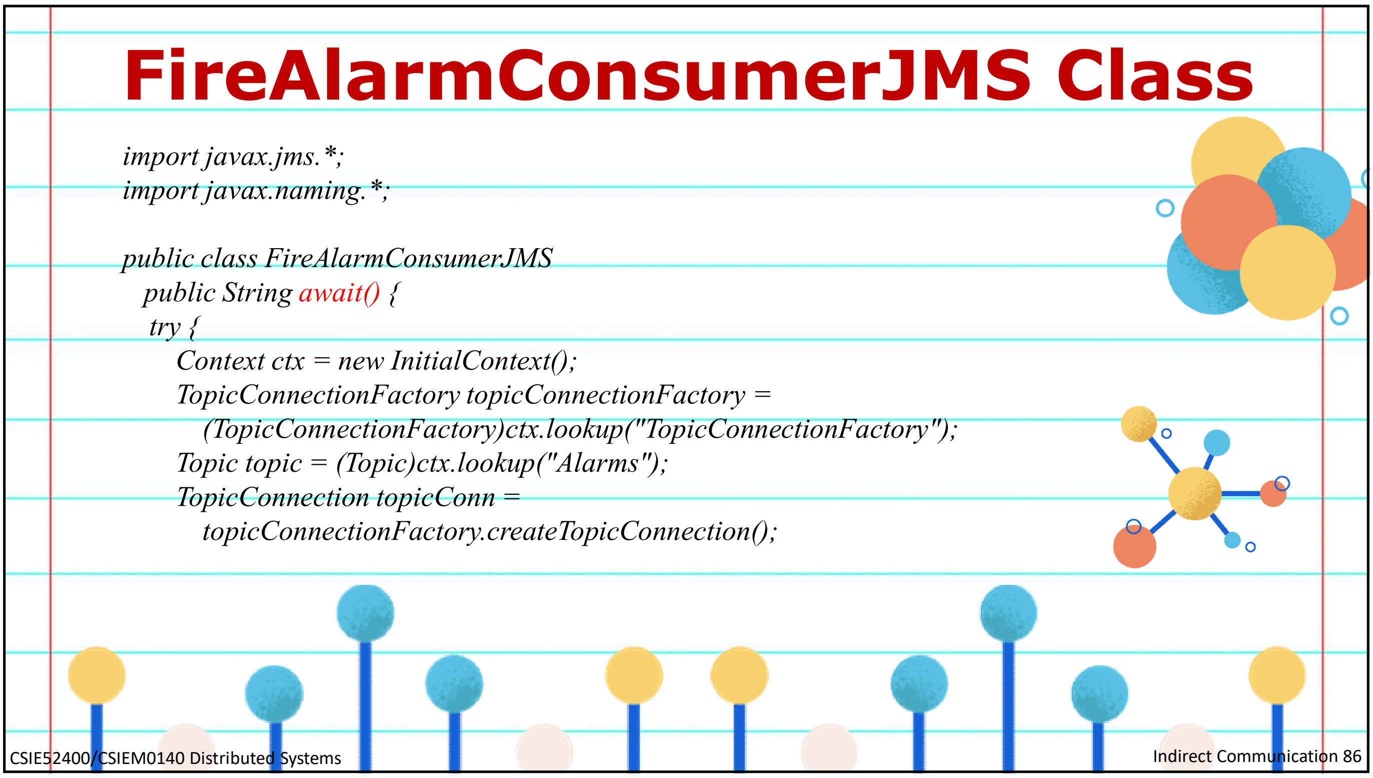
FireAlarmJMS Class

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {

public void raise() {
  try {
    Context ctx = new InitialContext(); // the naming context
    TopicConnectionFactory topicConnectionFactory =
      (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
    Topic topic = (Topic)ctx.lookup("Alarms");
    TopicConnection topicConn =
      topicConnectionFactory.createTopicConnection();
```



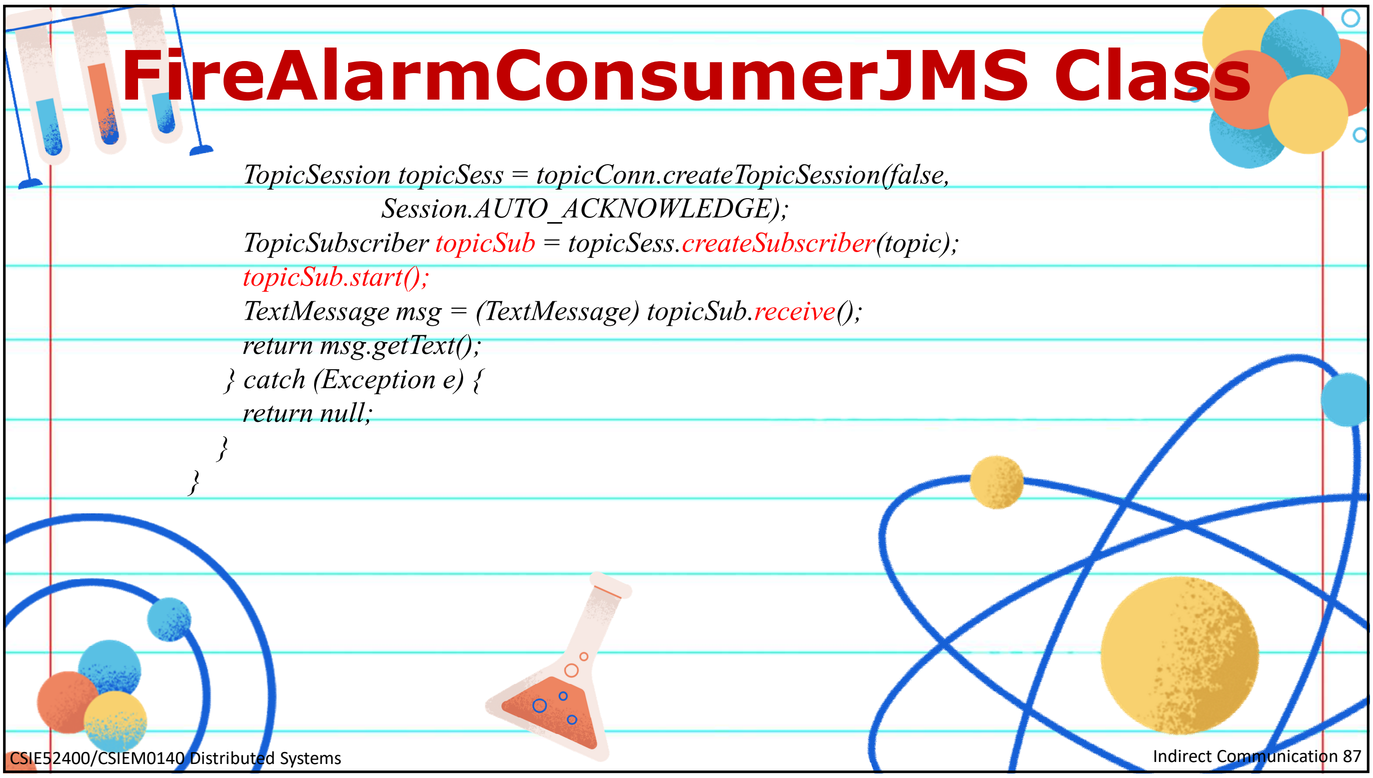
FireAlarmJMS Class

```
      TopicSession topicSess = topicConn.createTopicSession(false,
         Session.AUTO_ACKNOWLEDGE);  // false: not transacted
      TopicPublisher topicPub = topicSess.createPublisher(topic);
      TextMessage msg = topicSess.createTextMessage();
       msg.setText("Fire!");
      topicPub.publish(msg);
    } catch (Exception e) {

      }
  }
```



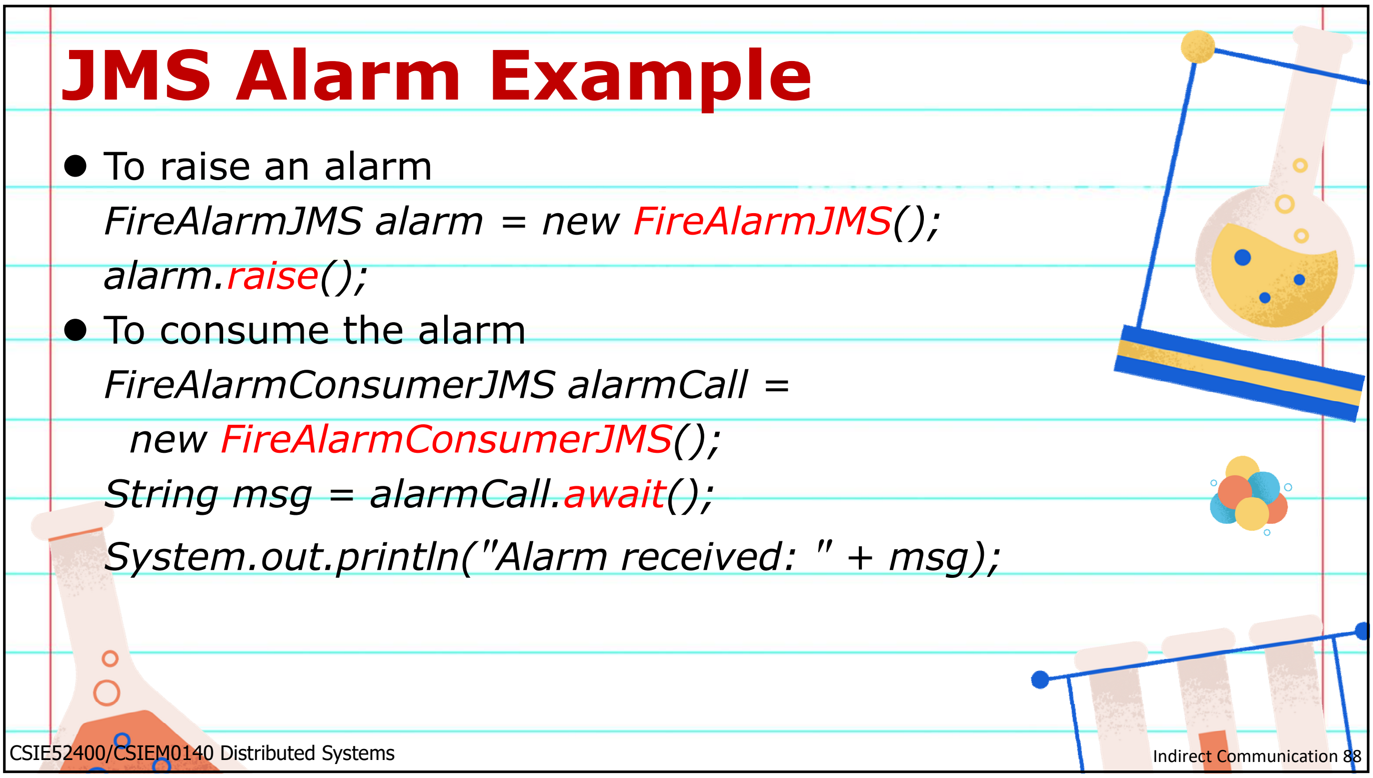
FireAlarmConsumerJMS Class

```
import javax.jms.*;
import javax.naming.*;

public class FireAlarmConsumerJMS
   public String await() {
    try {
        Context ctx = new InitialContext();
        TopicConnectionFactory topicConnectionFactory =
           (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
        Topic topic = (Topic)ctx.lookup("Alarms");
        TopicConnection topicConn =
           topicConnectionFactory.createTopicConnection();
```



FireAlarmConsumerJMS Class

```
    TopicSession topicSess = topicConn.createTopicSession(false,
                     Session.AUTO_ACKNOWLEDGE);
    TopicSubscriber topicSub = topicSess.createSubscriber(topic);
    topicSub.start();
    TextMessage msg = (TextMessage) topicSub.receive();
    return msg.getText();
  } catch (Exception e) {
    return null;
  }
}
```



JMS Alarm Example

- To raise an alarm

```
FireAlarmJMS alarm = new FireAlarmJMS();
alarm.raise();
```

- To consume the alarm

```
FireAlarmConsumerJMS alarmCall =
  new FireAlarmConsumerJMS();
String msg = alarmCall.await();
System.out.println("Alarm received: " + msg);
```

Popular Message Queue Systems

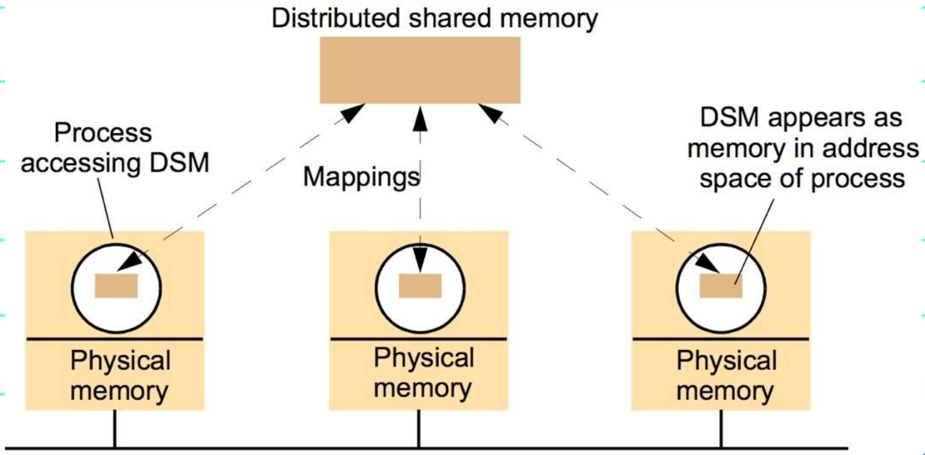
- RabbitMQ
- IBM MQ
- Amazon Simple Queue System (SQS)
- Apache ActiveMQ
- Apache Kafka
- Google Cloud Pub/Sub
- Microsoft Azure Service Bus
- Red Hat AMQ
- Anypoint MQ (MuleSoft)
- Solace PubSub+ Event Broker

Python Messaging Libraries

- Kombu — a messaging library supporting AMQP(Advanced Message Queuing Protocol)
- py-ampqlib — Another library for AMQP
- RabbitMQ & Pika client — a message broker which speaks AMQP, accessed through Pika
- Apache ActiveMQ & Stomp client — an open-source message broker accessed through Stomp
- mpi4py — MPI for Python, fast, support most MPI routines
- …
- Almost all message queue systems offer Python API.

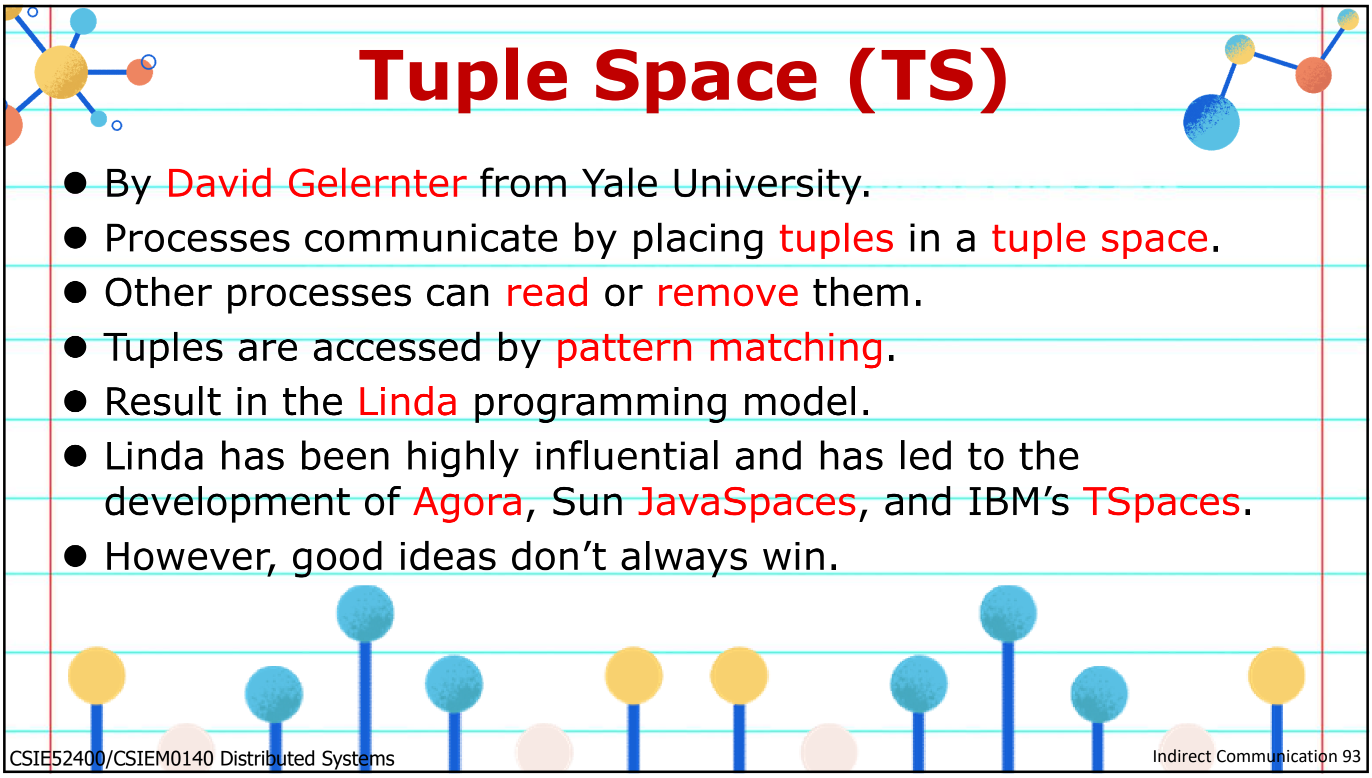
Distributed Shared Memory

- DSM is an abstraction for sharing data among nodes w/o shared physical memory.


Distributed shared memory
Process accessing DSM
Mappings
DSM appears as memory in address space of process
Physical memory
Physical memory
Physical memory

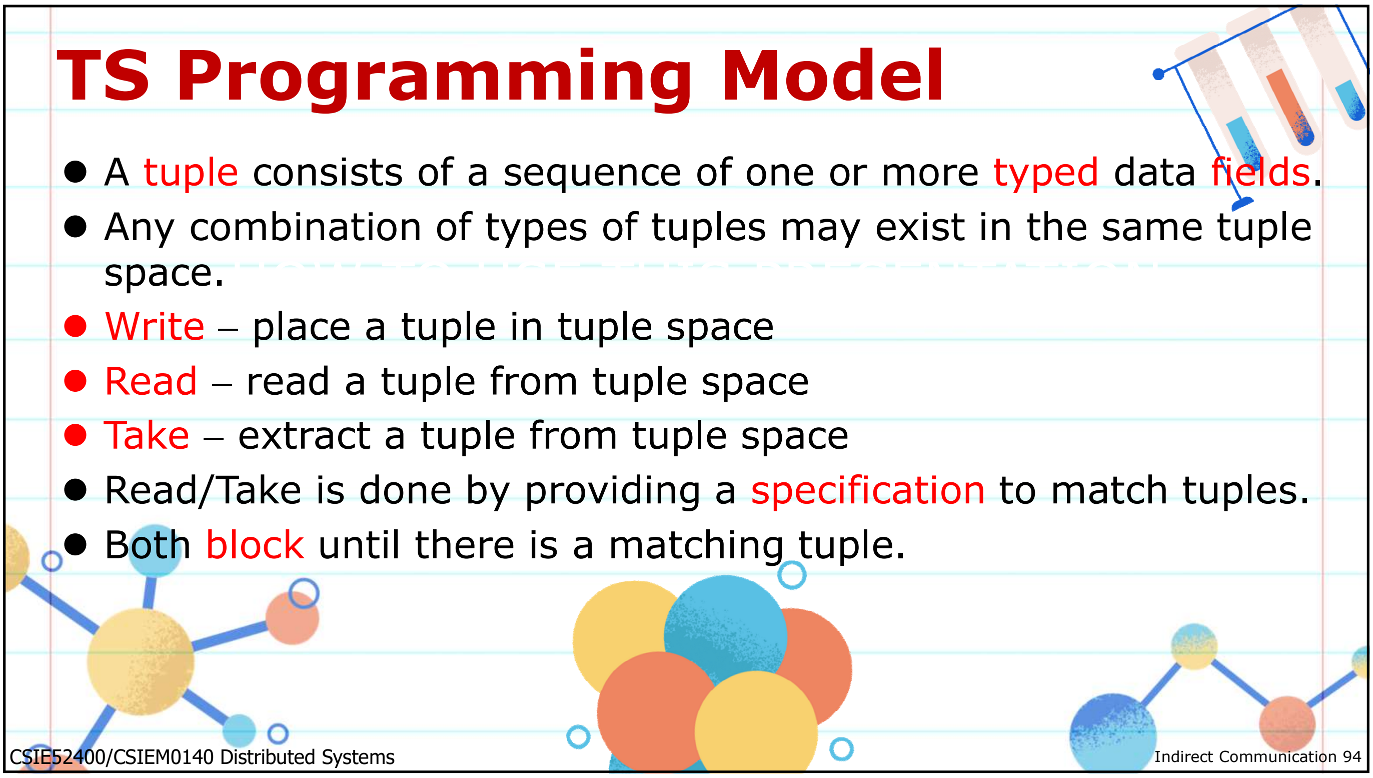
Message Passing vs. DSM

Message Passing	Distributed Shared Memory
Marshalling and transmission of variables between possibly heterogenous processes	Homogenous processes share variables
Processes communicate while being protected from each other	Processes share DMS with no support for encapsulation and information hiding
Synchronization between processes is achieved in the message model through message passing primitives	Synchronization is via normal constructs for shared-memory programming such as locks and semaphores
Processes communicating via message passing must execute at the same time	DSM can be made persistent, processes communicating via DSM may execute with nonoverlapping lifetimes



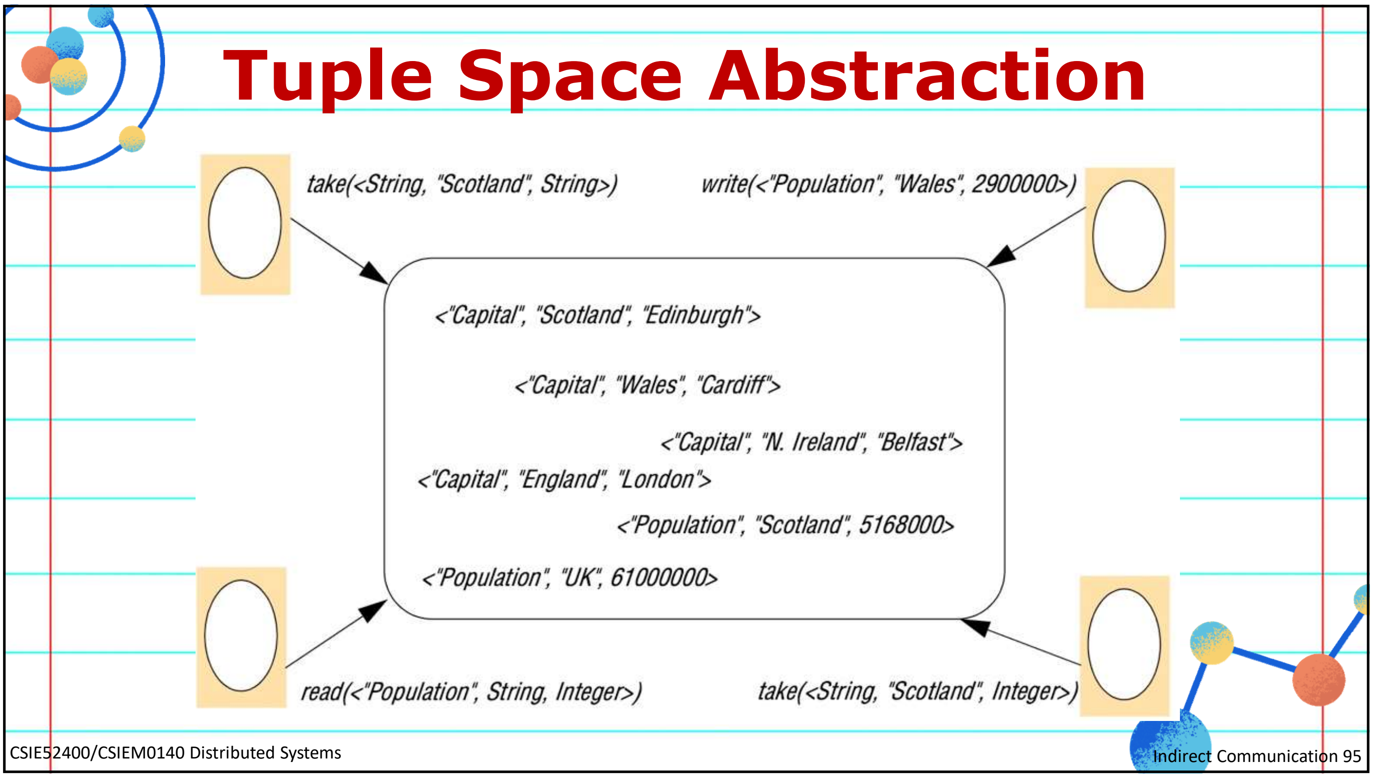
Tuple Space (TS)

- By David Gelernter from Yale University.
- Processes communicate by placing tuples in a tuple space.
- Other processes can read or remove them.
- Tuples are accessed by pattern matching.
- Result in the Linda programming model.
- Linda has been highly influential and has led to the development of Agora, Sun JavaSpaces, and IBM's TSpaces.
- However, good ideas don't always win.



TS Programming Model

- A tuple consists of a sequence of one or more typed data fields.
- Any combination of types of tuples may exist in the same tuple space.
- Write – place a tuple in tuple space
- Read – read a tuple from tuple space
- Take – extract a tuple from tuple space
- Read/Take is done by providing a specification to match tuples.
- Both block until there is a matching tuple.



Tuple Space Abstraction

take(<String, "Scotland", String>)

write(<"Population", "Wales", 2900000>)

<"Capital", "Scotland", "Edinburgh">

<"Capital", "Wales", "Cardiff">

<"Capital", "N. Ireland", "Belfast">

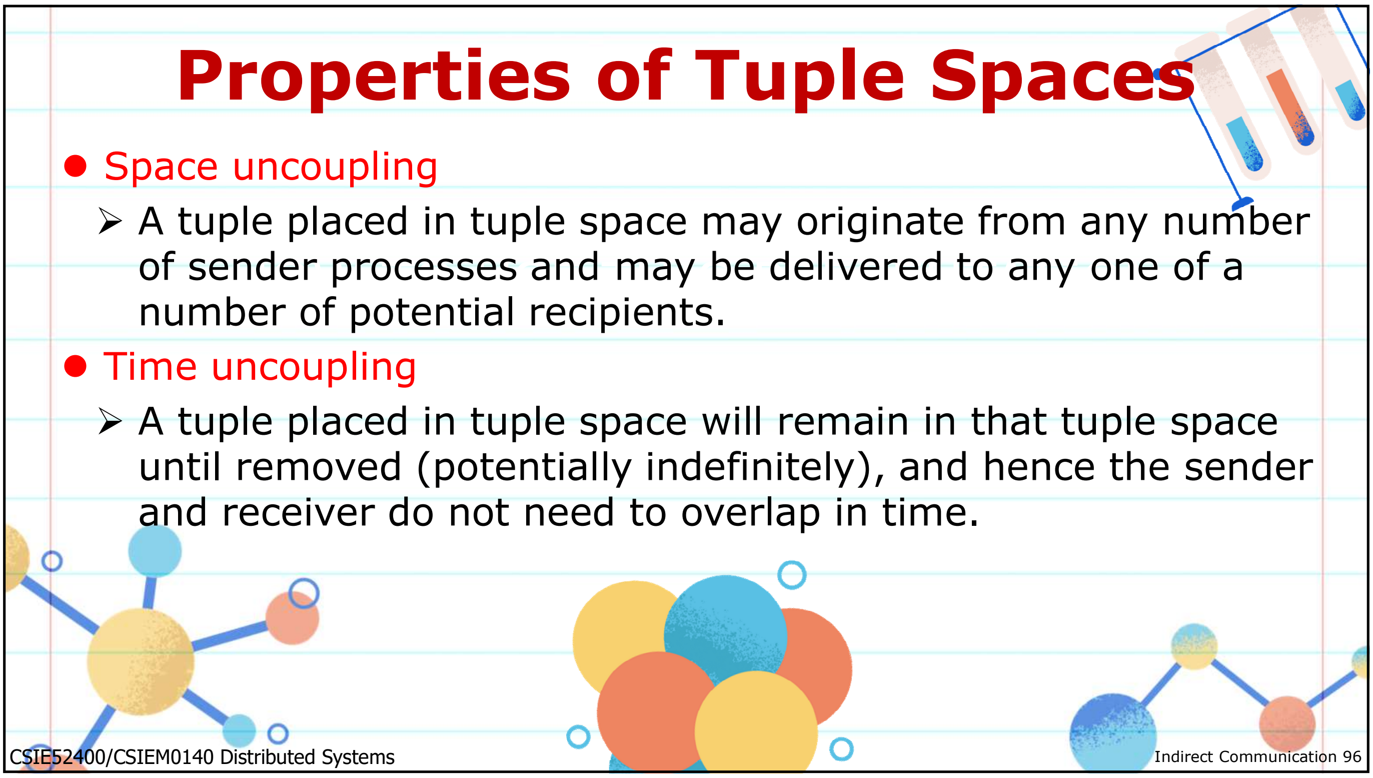
<"Capital", "England", "London">

<"Population", "Scotland", 5168000>

<"Population", "UK", 61000000>

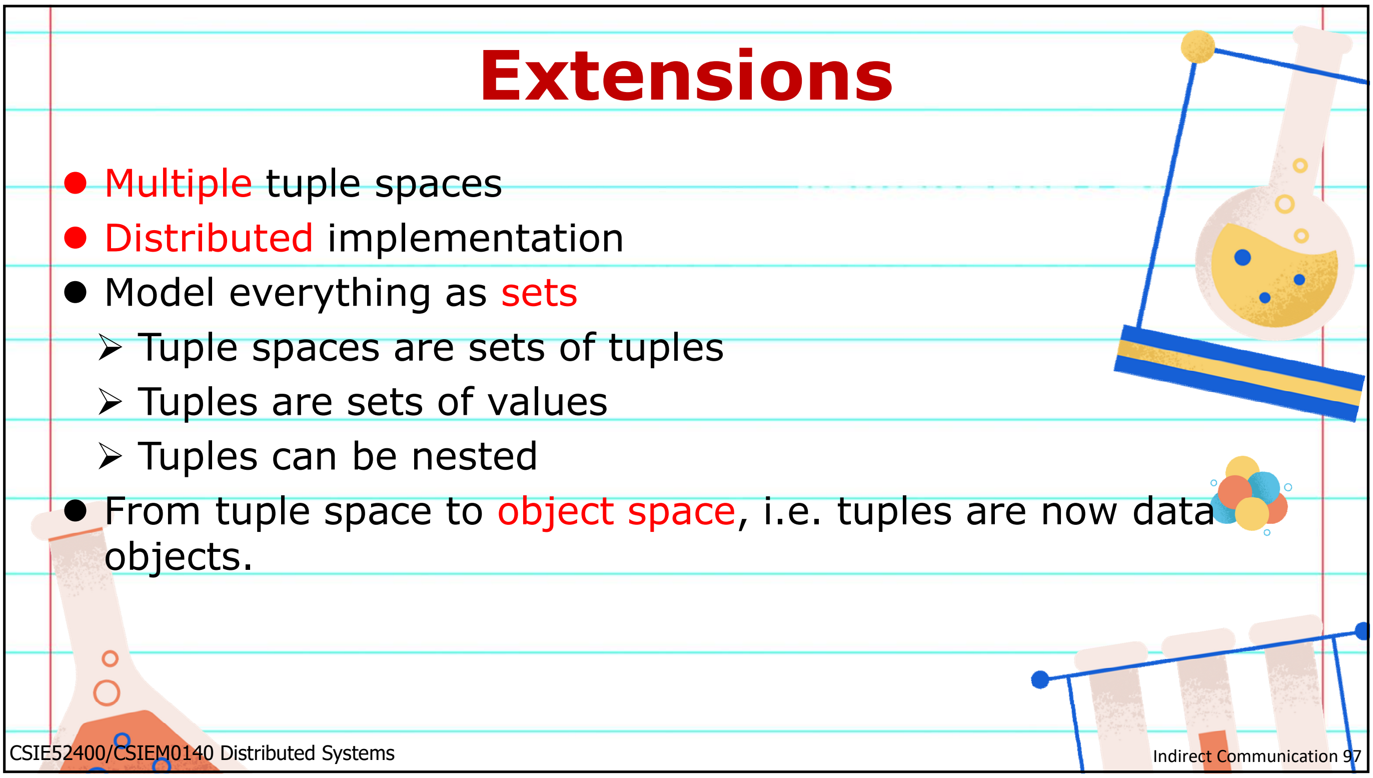
read(<"Population", String, Integer>)

take(<String, "Scotland", Integer>)



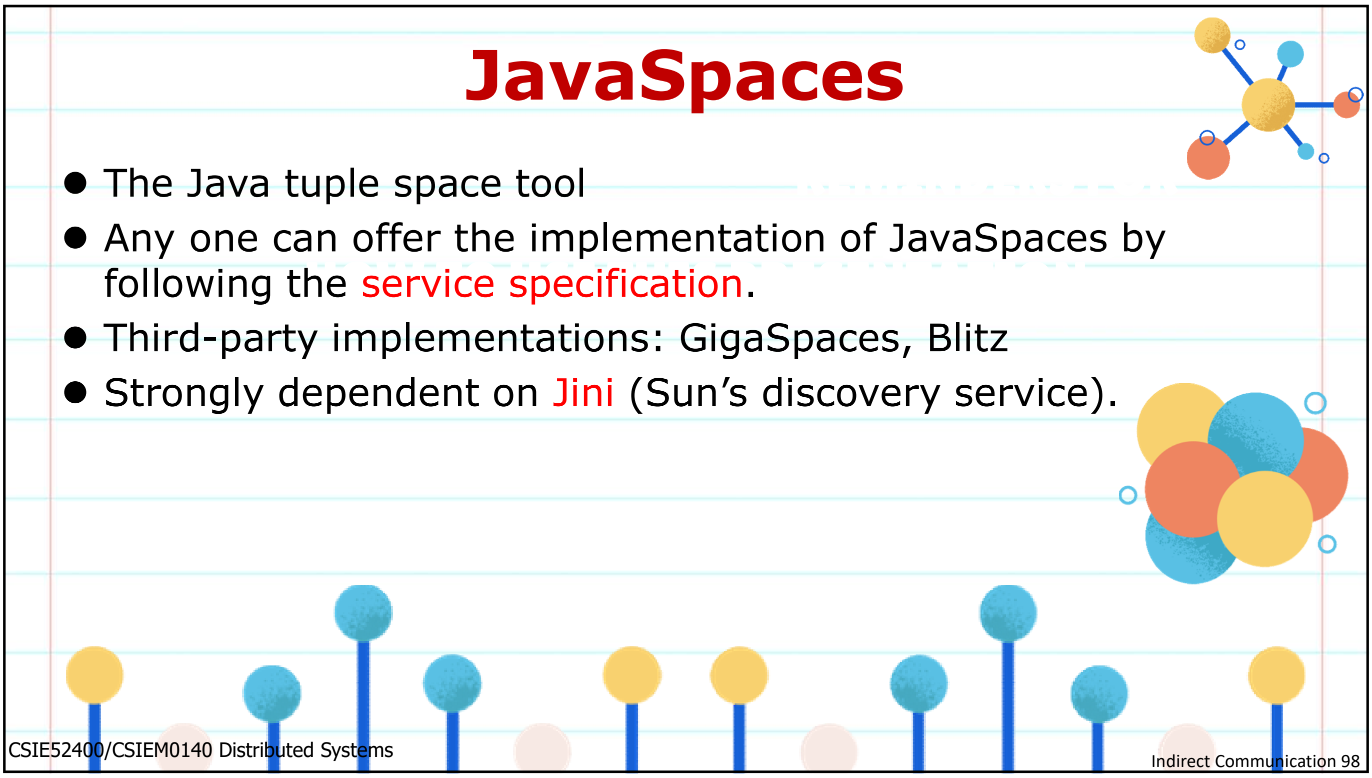
Properties of Tuple Spaces

- Space uncoupling
  - A tuple placed in tuple space may originate from any number of sender processes and may be delivered to any one of a number of potential recipients.
- Time uncoupling
  - A tuple placed in tuple space will remain in that tuple space until removed (potentially indefinitely), and hence the sender and receiver do not need to overlap in time.



Extensions

- Multiple tuple spaces
- Distributed implementation
- Model everything as sets
  - Tuple spaces are sets of tuples
  - Tuples are sets of values
  - Tuples can be nested
- From tuple space to object space, i.e. tuples are now data objects.



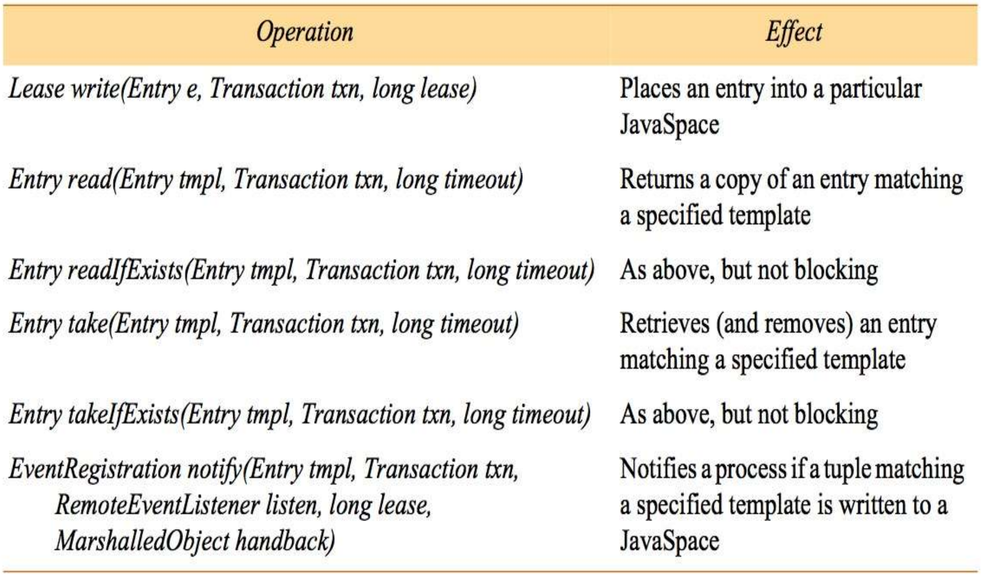
JavaSpaces

- The Java tuple space tool
- Any one can offer the implementation of JavaSpaces by following the service specification.
- Third-party implementations: GigaSpaces, Blitz
- Strongly dependent on Jini (Sun’s discovery service).

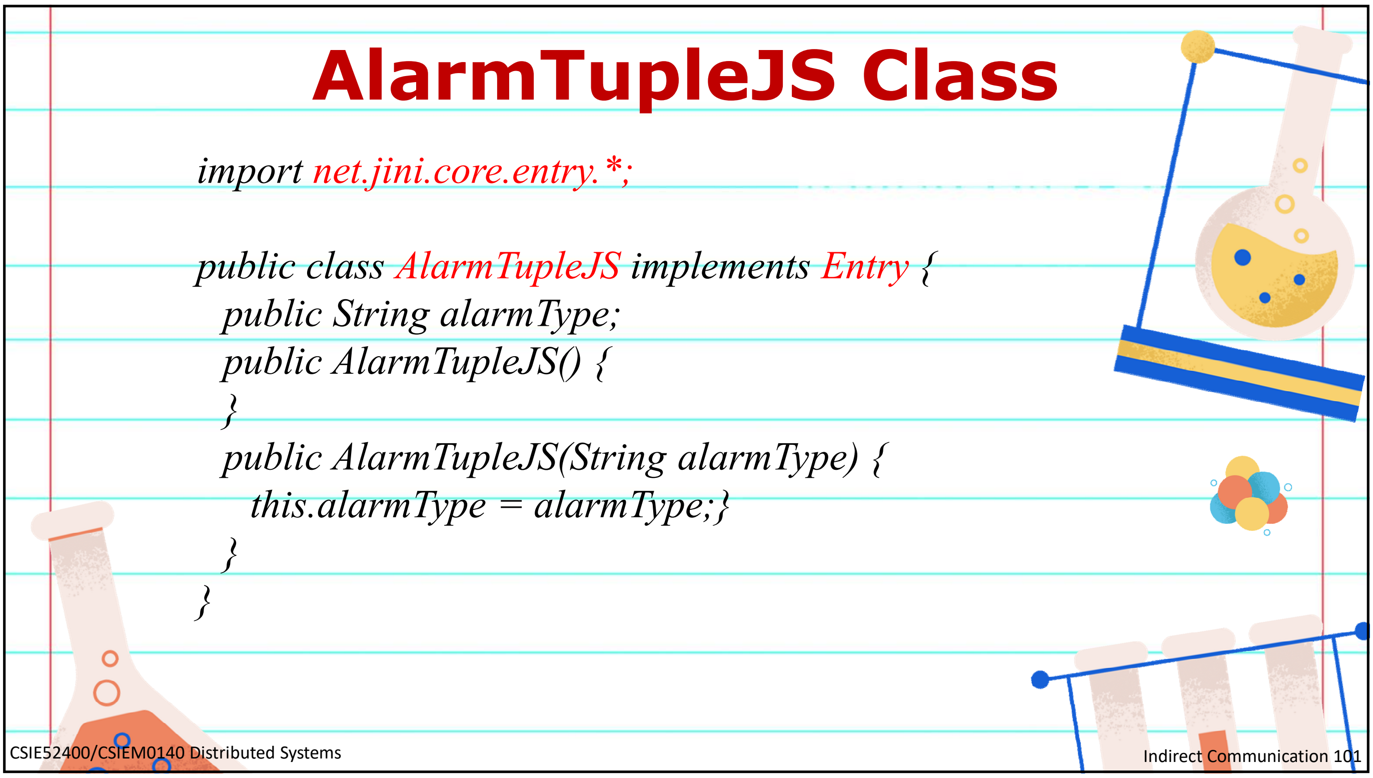
Programming JavaSpaces

- Can create any number of *JavaSpace*.
- An object in a JavaSpace is called an entry.
- A process can *write* an entry into a JavaSpace with an associated *lease* (time of availability).
- *read* returns a copy of a matching (specified by a template) entry.
- *take* removes a matching entry.
- *read/take* are blocking ops with timeout.
- *readIfExists/takeIfExists* return null if not exists
- All ops can be transactional.

JavaSpaces API

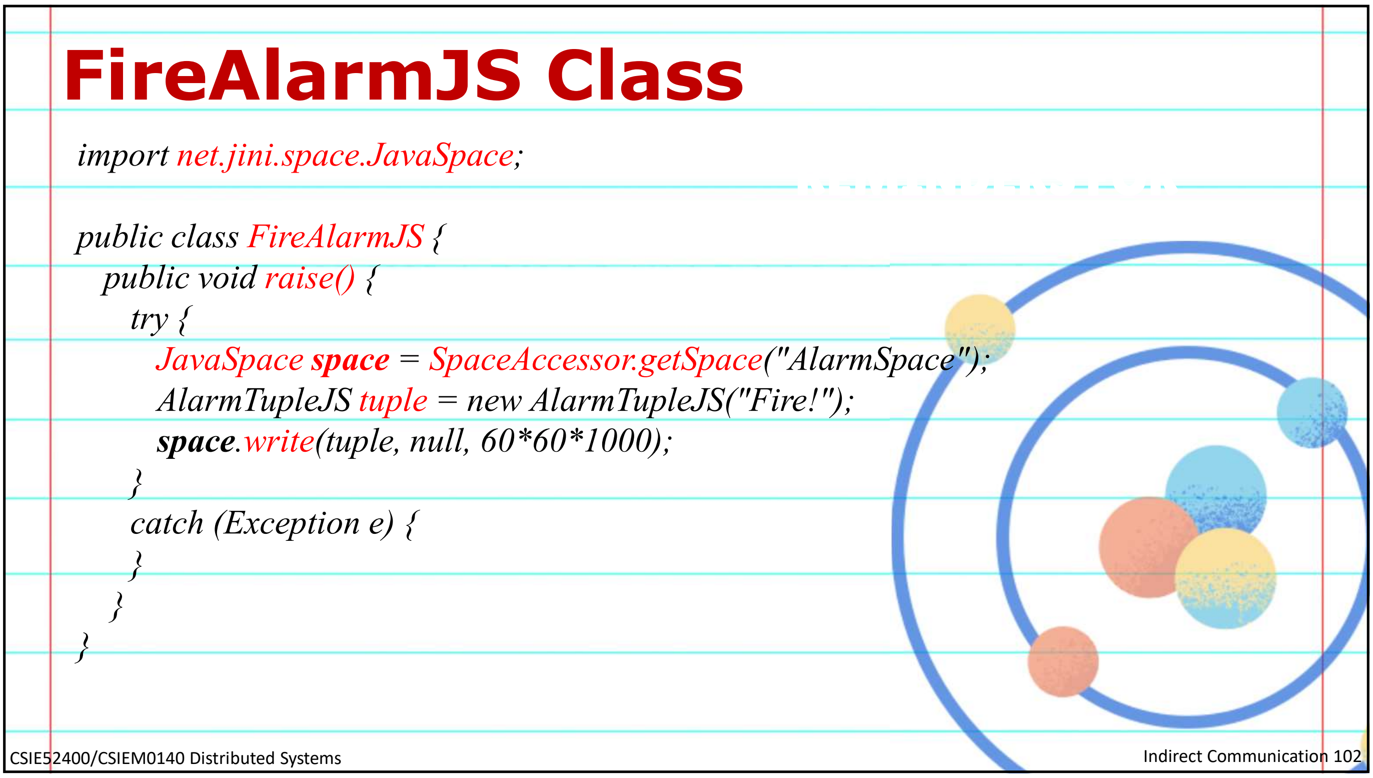


Operation	*Effect*
Lease write(Entry e, Transaction txn, long lease)	Places an entry into a particular JavaSpace
Entry read(Entry tmpl, Transaction txn, long timeout)	Returns a copy of an entry matching a specified template
Entry readIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
Entry take(Entry tmpl, Transaction txn, long timeout)	Retrieves (and removes) an entry matching a specified template
Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)	Notifies a process if a tuple matching a specified template is written to a JavaSpace



AlarmTupleJS Class

```
import net.jini.core.entry.*;

public class AlarmTupleJS implements Entry {
  public String alarmType;
  public AlarmTupleJS() {
  }
  public AlarmTupleJS(String alarmType) {
    this.alarmType = alarmType;}
  }
}
```



FireAlarmJS Class

```
import net.jini.space.JavaSpace;

public class FireAlarmJS {
  public void raise() {
    try {
      JavaSpace space = SpaceAccessor.getSpace("AlarmSpace");
      AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
      space.write(tuple, null, 60*60*1000);
    }
    catch (Exception e) {
    }
  }
}
```

FileAlarmConsumerJS Class

```
import net.jini.space.JavaSpace;

public class FireAlarmConsumerJS {
   public String await() {
      try {
               JavaSpace space = SpaceAccessor.getSpace("AlarmSpace");
               AlarmTupleJS template = new AlarmTupleJS("Fire!");
               AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,
                              Long.MAX_VALUE);
               return recvd.alarmType;
      }
      catch (Exception e) {
               return null;
         }
   }
}
```

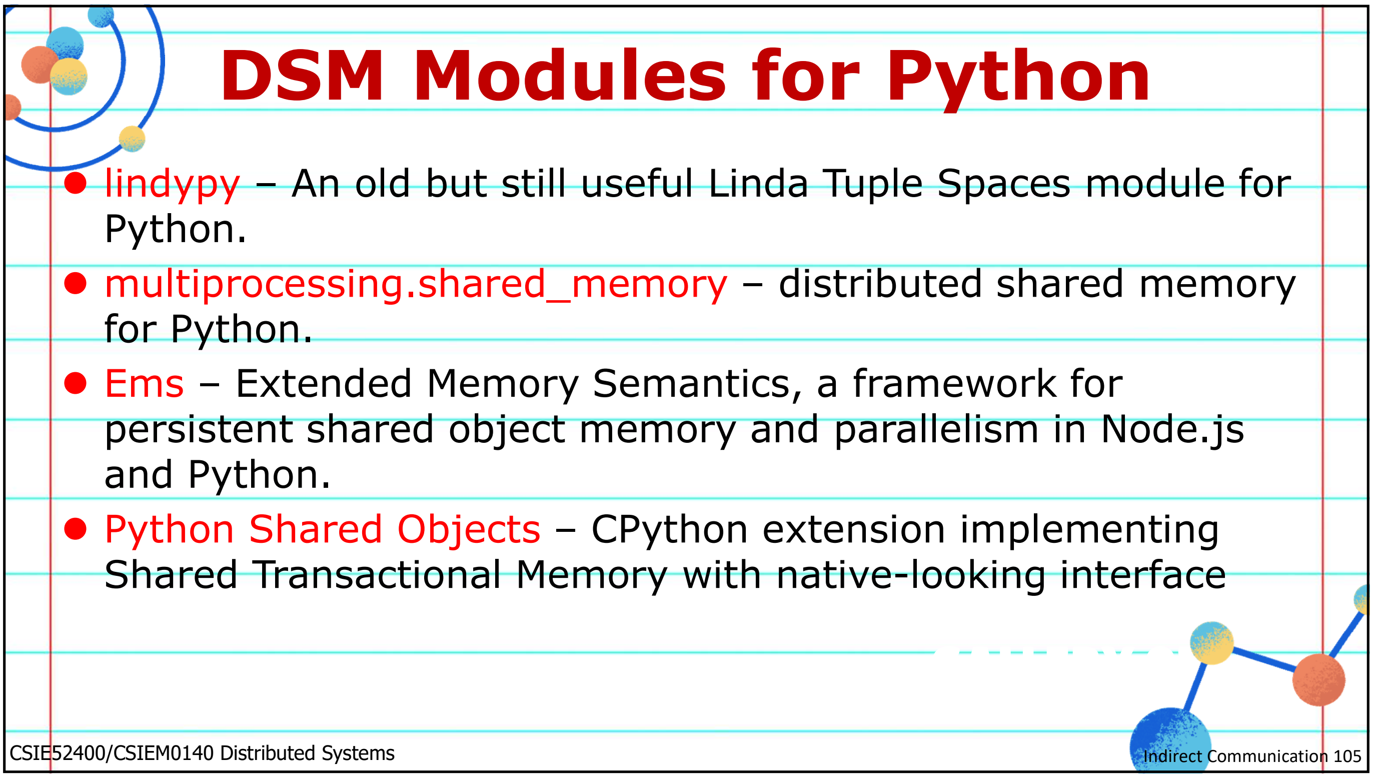
Usage Example

- To raise an alarm

```
FireAlarmJS alarm = new FireAlarmJS();
alarm.raise();
```

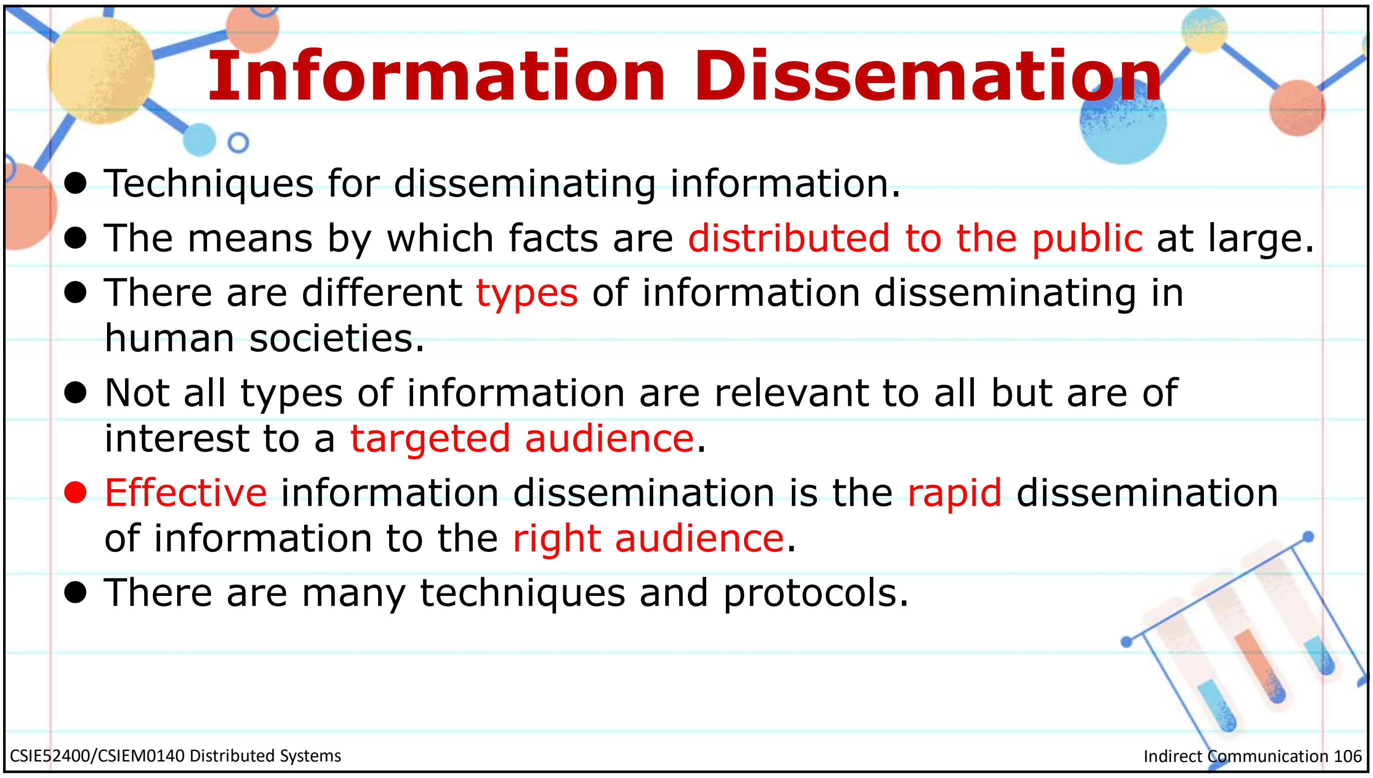
- To consume an alarm

```
FireAlarmConsumerJS alarmCall =
      new FireAlarmConsumerJS();
String msg = alarmCall.await();
System.out.println(“Alarm received: “ + msg);
```



DSM Modules for Python

- lindypy – An old but still useful Linda Tuple Spaces module for Python.
- multiprocessing.shared_memory – distributed shared memory for Python.
- Ems – Extended Memory Semantics, a framework for persistent shared object memory and parallelism in Node.js and Python.
- Python Shared Objects – CPython extension implementing Shared Transactional Memory with native-looking interface



Information Dissemation

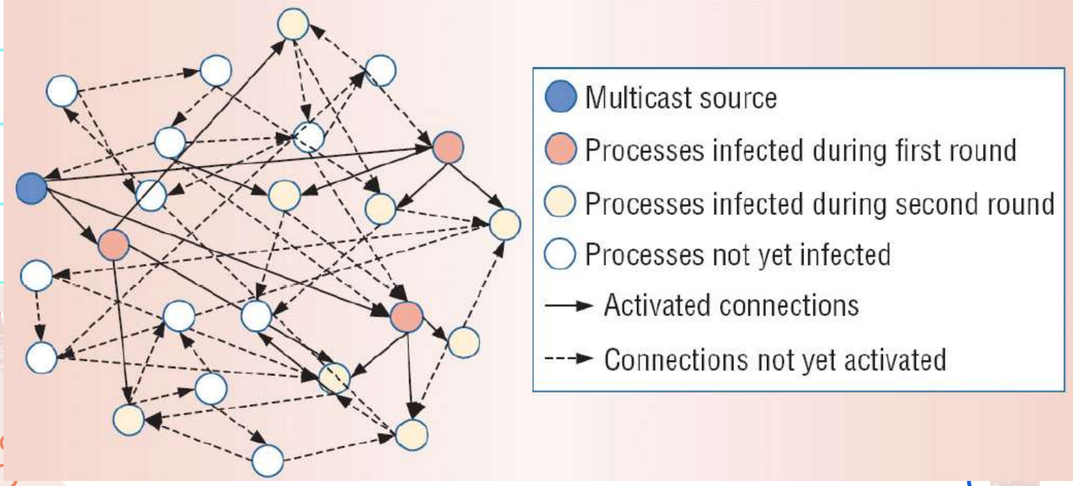
- Techniques for disseminating information.
- The means by which facts are distributed to the public at large.
- There are different types of information disseminating in human societies.
- Not all types of information are relevant to all but are of interest to a targeted audience.
- Effective information dissemination is the rapid dissemination of information to the right audience.
- There are many techniques and protocols.

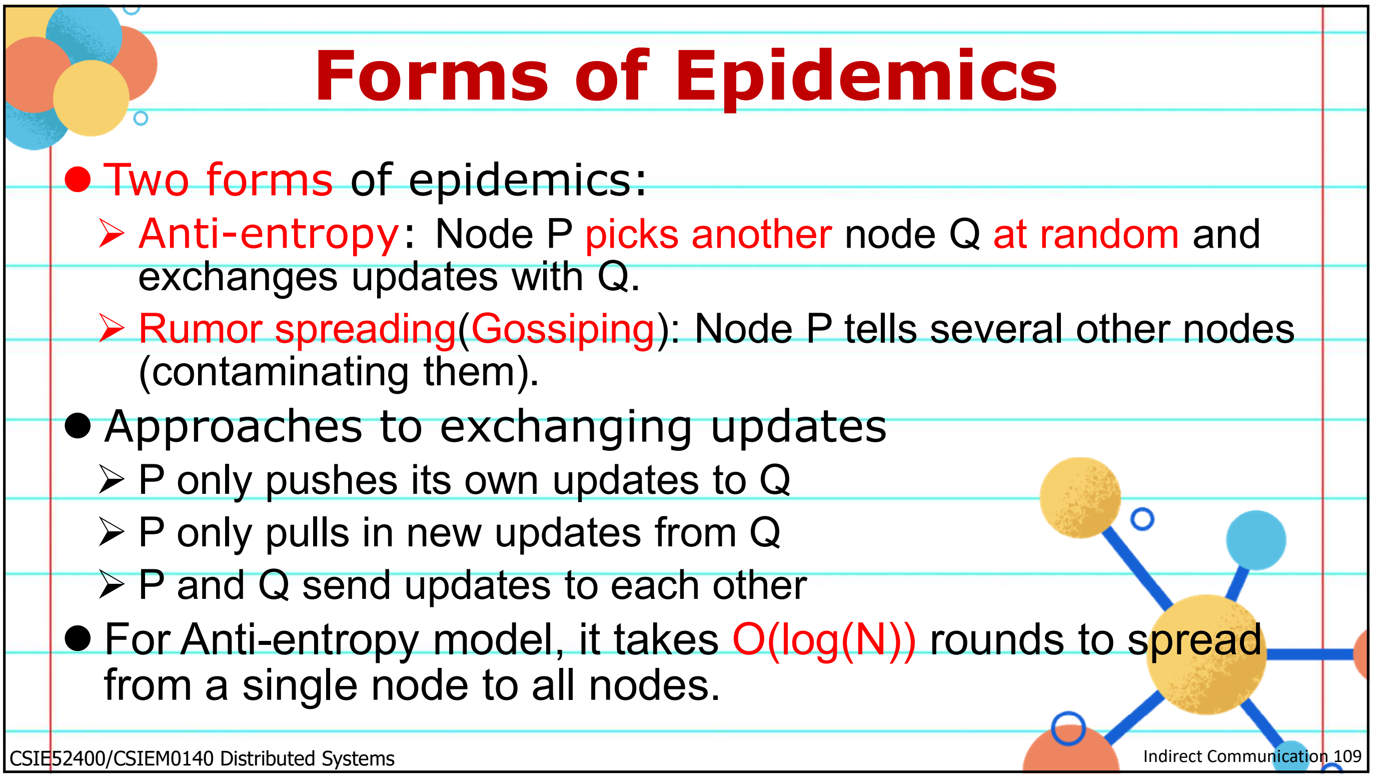
Flooding

- *P* sends a message *m* to each of its neighbors. Each neighbor will forward that message, except to *P*, and only if it had not seen *m* before.
- Variation: *Q* forward a message with a certain probability p_{flood}, possibly even dependent on its own number of neighbors (i.e., node degree) or the degree of its neighbors.
- The effect can be dramatic: the total number of messages sent will drop linearly in p_{flood}.
- The **risk**: the lower p_{flood}, the higher the chance that not all nodes in the network will be reached. (why?)

Epidemic(Gossip) Protocols

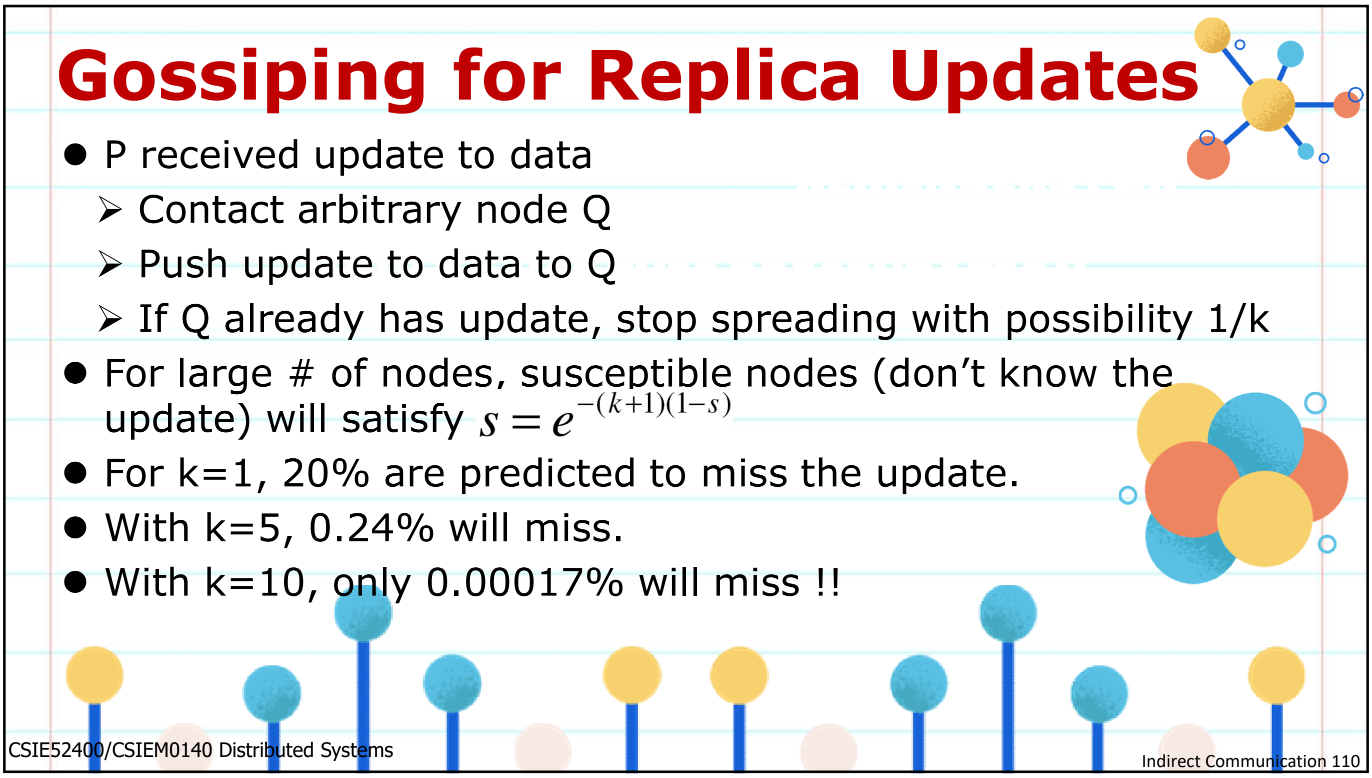
- Like diseases or rumors spread among people


● Multicast source
● Processes infected during first round
● Processes infected during second round
○ Processes not yet infected
→ Activated connections
⇢ Connections not yet activated



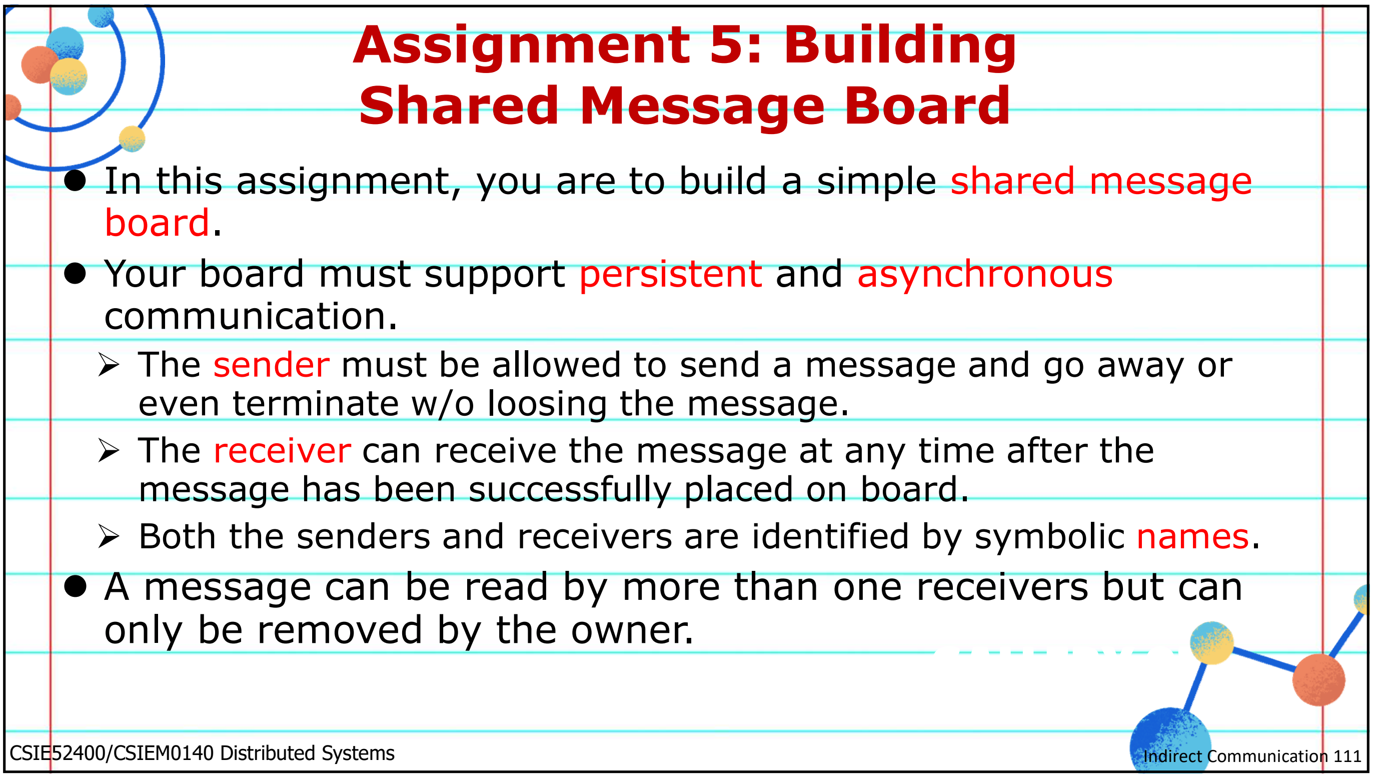
Forms of Epidemics

- Two forms of epidemics:
  - Anti-entropy: Node P picks another node Q at random and exchanges updates with Q.
  - Rumor spreading(Gossiping): Node P tells several other nodes (contaminating them).
- Approaches to exchanging updates
  - P only pushes its own updates to Q
  - P only pulls in new updates from Q
  - P and Q send updates to each other
- For Anti-entropy model, it takes O(log(N)) rounds to spread from a single node to all nodes.



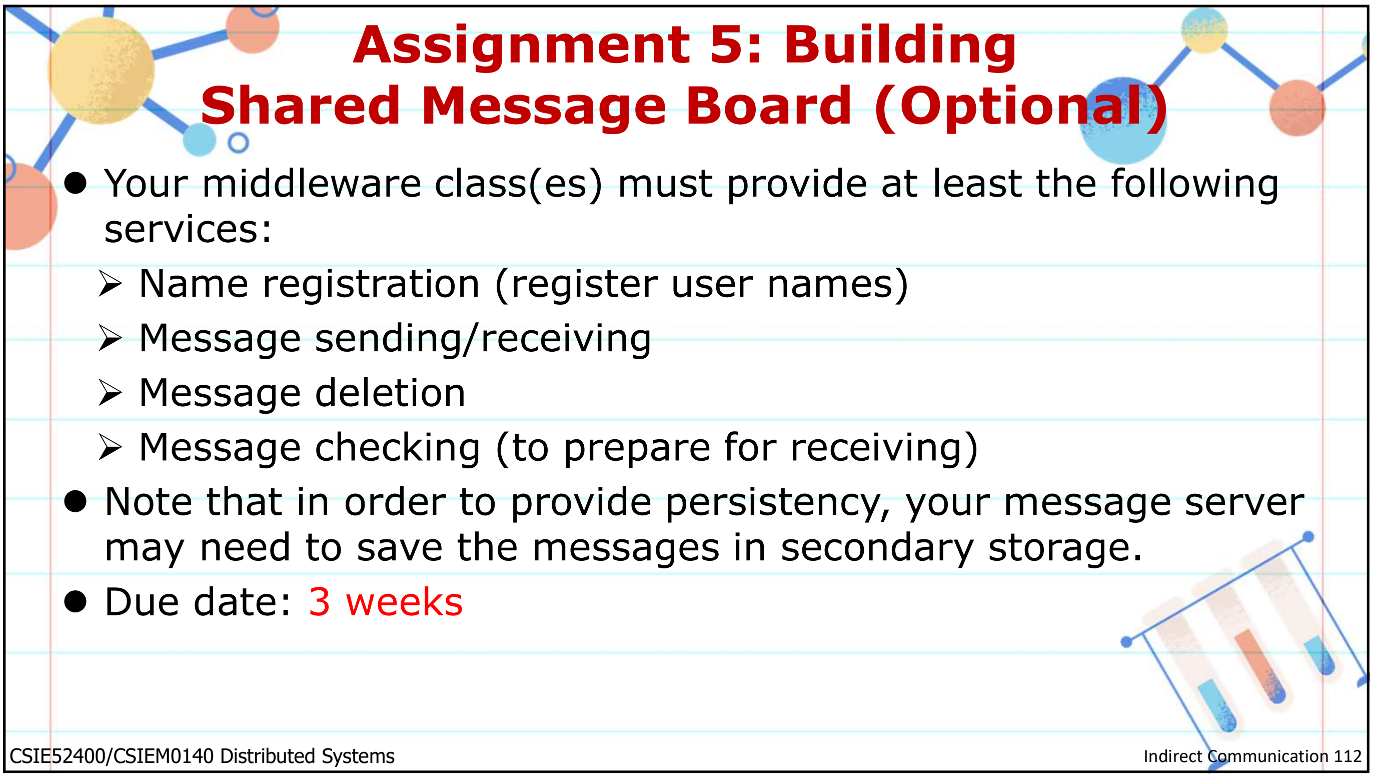
Gossiping for Replica Updates

- P received update to data
  - Contact arbitrary node Q
  - Push update to data to Q
  - If Q already has update, stop spreading with possibility 1/k
- For large # of nodes, susceptible nodes (don't know the update) will satisfy $s = e^{-(k+1)(1-s)}$
- For k=1, 20% are predicted to miss the update.
- With k=5, 0.24% will miss.
- With k=10, only 0.00017% will miss !!



Assignment 5: Building Shared Message Board

- In this assignment, you are to build a simple shared message board.
- Your board must support persistent and asynchronous communication.
  - The sender must be allowed to send a message and go away or even terminate w/o loosing the message.
  - The receiver can receive the message at any time after the message has been successfully placed on board.
  - Both the senders and receivers are identified by symbolic names.
- A message can be read by more than one receivers but can only be removed by the owner.



Assignment 5: Building Shared Message Board (Optional)

- Your middleware class(es) must provide at least the following services:
  - Name registration (register user names)
  - Message sending/receiving
  - Message deletion
  - Message checking (to prepare for receiving)
- Note that in order to provide persistency, your message server may need to save the messages in secondary storage.
- Due date: 3 weeks