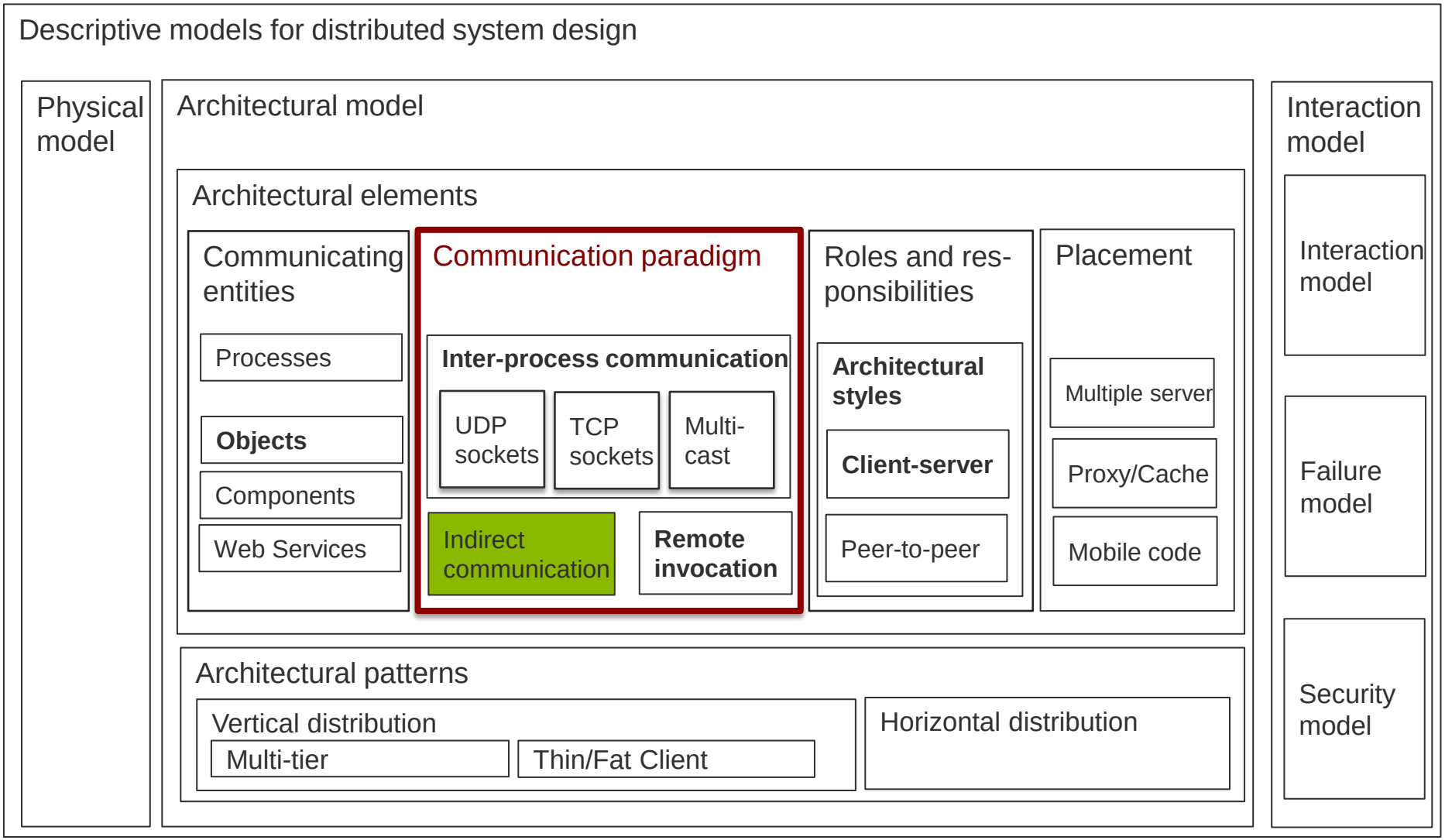
Freie Universität Berlin

Indirect Communication - II

Netzprogrammierung
(Algorithmen und Programmierung V)

Our topics last week



Our topics today

- Message queues
- Shared Memory
- Tuple Spaces

Indirect communication

Message queues

Characteristics

Whereas groups and publish/subscribe systems provide a one-to-many style of communication, messages queues provide a point-to-point service using the concept of a **message queue as an indirection**.

Message queues are also referred to as **Message-Oriented Middleware**.

Commercial middleware such as

- IBM WebSphere MQ
- Microsoft MSMQ
- Oracle Stream Advanced Queuing (AQ)

Applications

Freie Universität Berlin

Enterprise application integration (EAI)

- is an integration framework composed of a collection of technologies and services which form a **middleware** (e.g. distributed message queues) to enable integration of systems and applications across the enterprise

Transaction Processing Systems

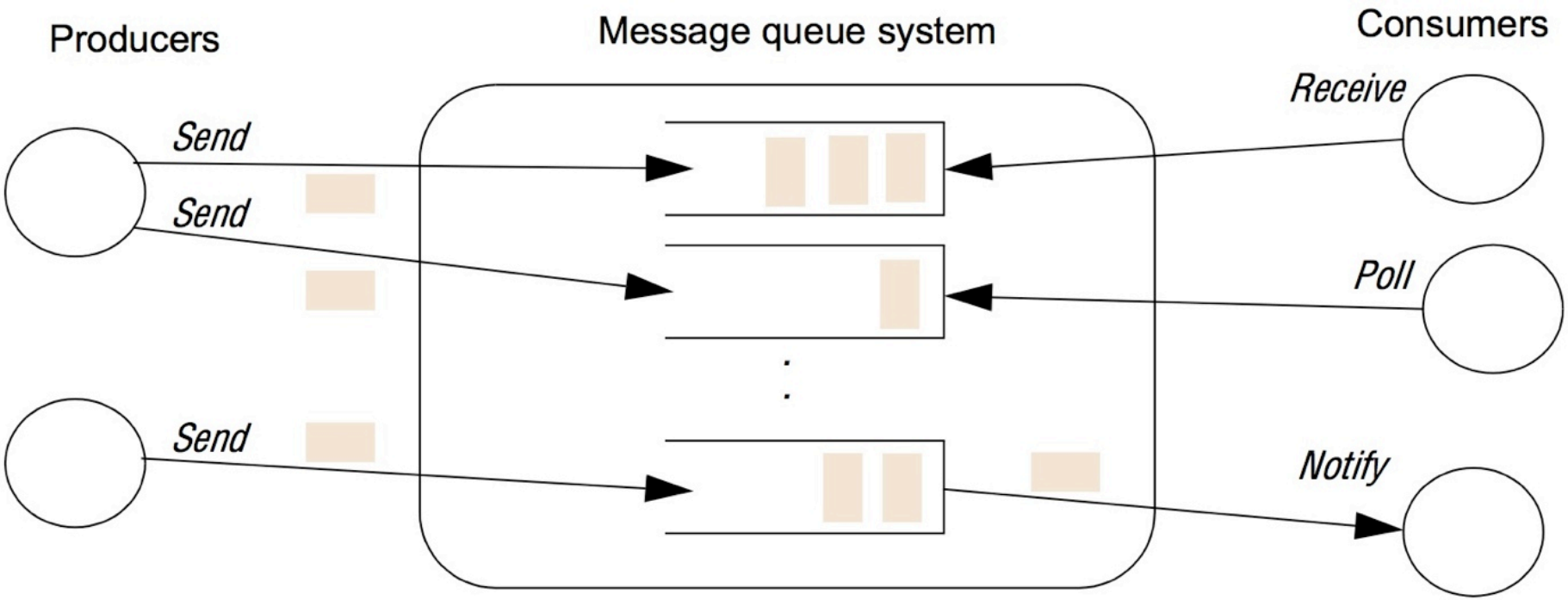
- Message queues have intrinsic support of transactions for transactions

Freie Universität Berlin

Message Queues

The programming model

Programming model



Queuing policy typically FIFO, priority or other selection criteria (e.g. metadata based)

Styles of “Receive”

blocking receive

- will block until an appropriate message is available

non-blocking receive (a polling operation)

- will check the status of the queue and return a message if available, or a not available indication otherwise

notify operation

- will issue an event notification when a message is available in the associated queue.

Properties

Persistent messages

- Messages are stored until they are consumed

Reliable delivery

- Integrity: the message received is the same as the one sent, and no messages are delivered twice
- Validity: message is eventually received (but not guarantees about the timing of delivery)

Transactional: „all steps in a transaction or nothing“

- Typically supported by additional external tranaction service

+ (message transformation, security, …)

Freie Universität Berlin

Message Queues

Implementation issues

Centralized vs. Decentralized

Centralized

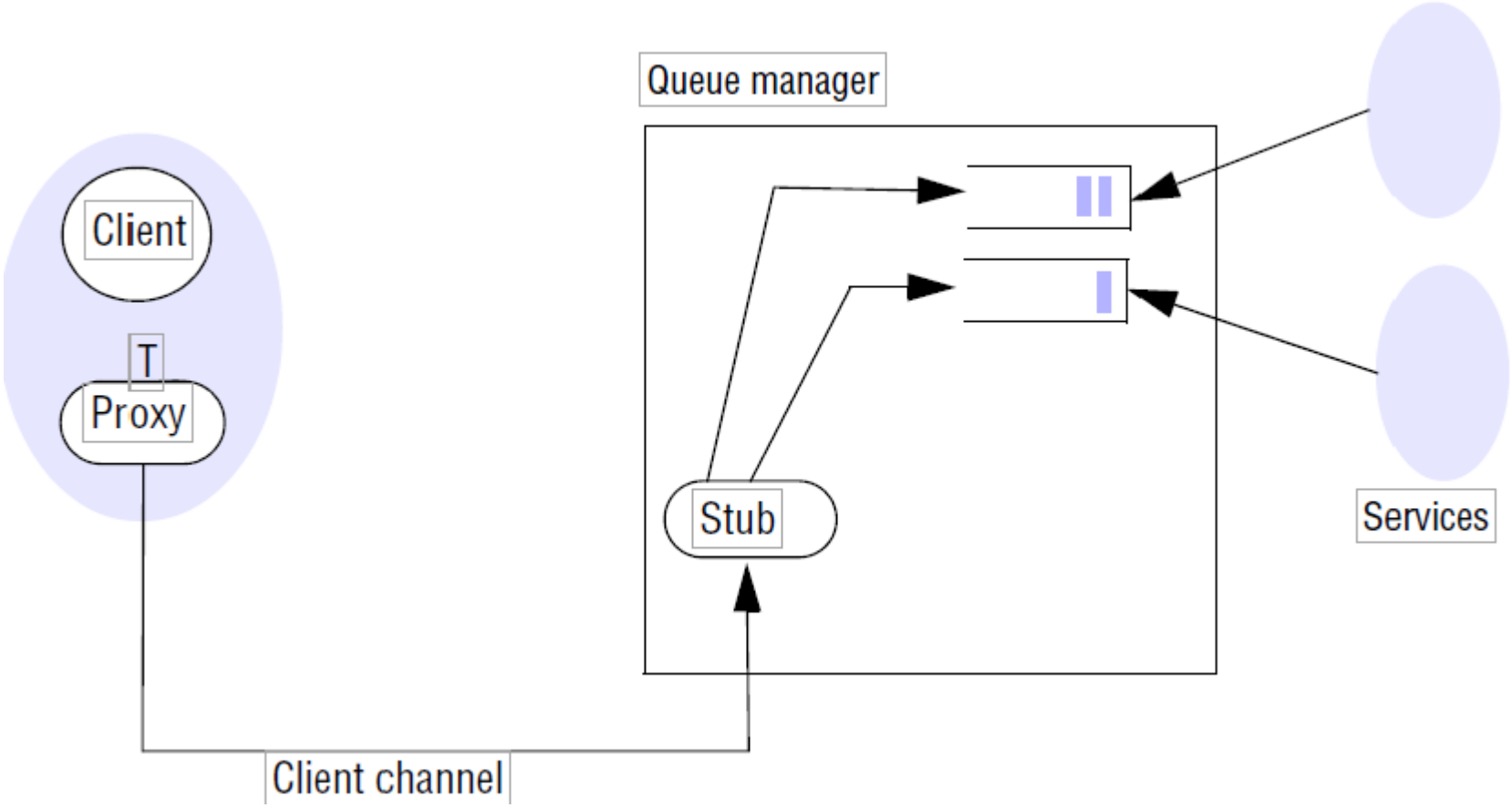
one or more message queues managed by a centralized queue manager located at a given node

- Advantage: simplicity
- Drawback: manager can become heavy-weight, bottelneck, single-point-of-failure

Decentralized

federated structure with **message channel** as a unidirectional connection between two queue managers that is used to forward messages asynchronously from one queue to another. A message channel is managed by a **message channel agent** (MCA) at each end

Centralized Queue Manager



Client Channels issue commands of the **Message Queue Interface** (MQI) on the proxy and sent them transparently to the **Message Queue Manager** via RPC.

Decentralized Topologies

Freie Universität Berlin

Decentralized queue managers allow network topologies, e.g., trees, meshes or a bus-based networks

Example: Hub-and-spoke approach

- Often used in large scale geographically distributed deployments
  - ability to connect to a local spoke over a high-bandwidth connection
- One queue manager is **hub** providing main services
  - placed somewhere appropriate in the network, on a node with sufficient resources to deal with the volume of traffic
- Clients connect through queue managers called **spokes** which relay messages to the queues of the hub
  - placed strategically around the network to support different clients

Freie Universität Berlin

Java Messaging Service (JMS)

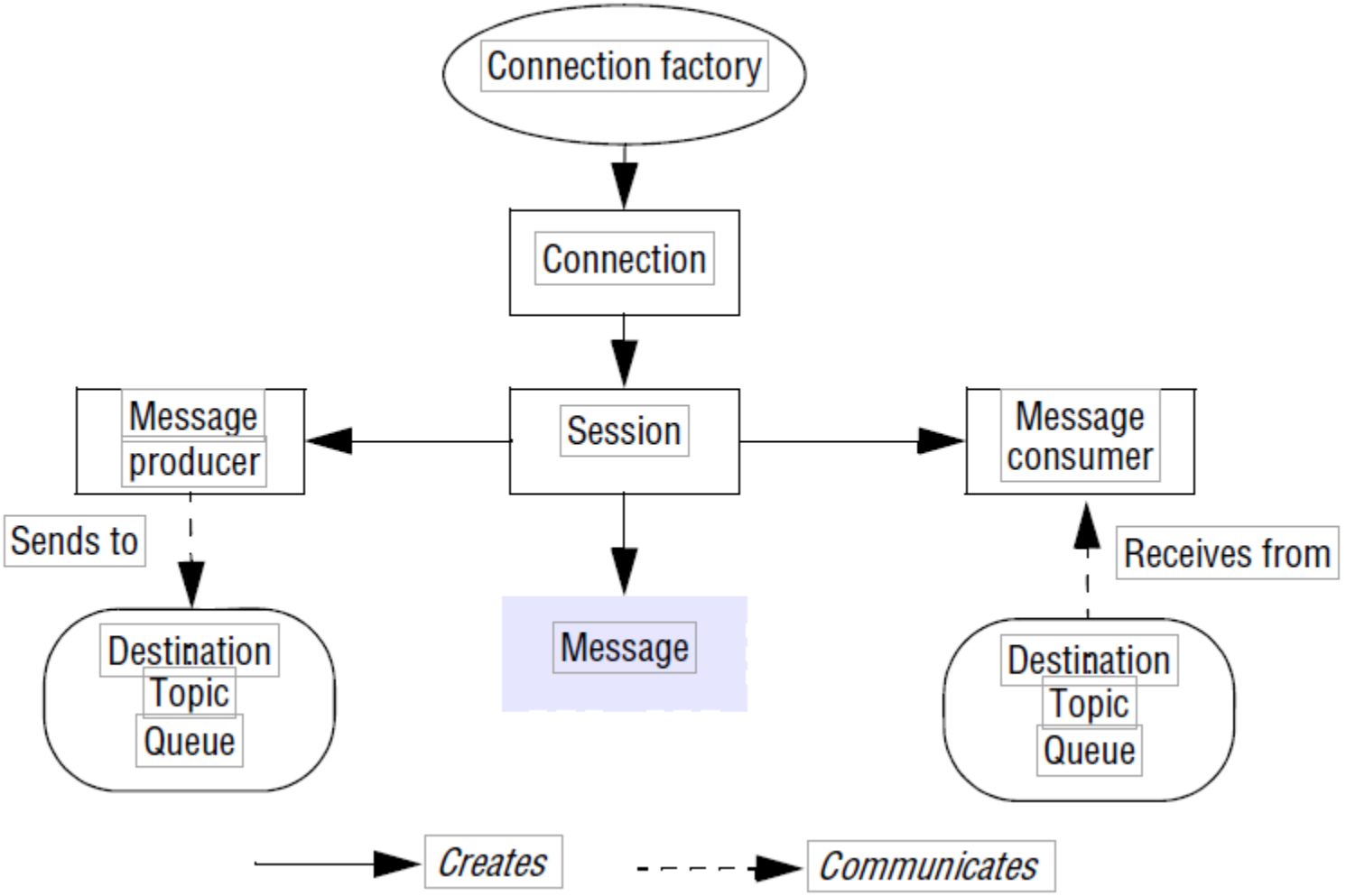
- standardized (java.sun.com XI) way for distributed Java programs to communicate indirectly
  - unifies the publish-subscribe and message queue paradigms by supporting topics and queues

Java Messaging Service (JMS) – Programming Model Terminology

Freie Universität Berlin

- A JMS client is a Java program or component that produces or consumes messages, a JMS producer is a program that creates and produces messages and a JMS consumer is a program that receives and consumes messages.
- A JMS provider is any of the multiple systems that implement the JMS specification.
- A JMS message is an object that is used to communicate information between JMS clients (from producers to consumers).
- A JMS destination is an object supporting indirect communication in JMS. It is either a JMS topic or a JMS queue.

Programming Model of JMS API



two type of connections: TopicConnection or a QueueConnection (+ combination)

- TopicConnection can support one or more topic sessions
- QueueConnection can support one or more queue sessions

JMS Implementation Objects

Freie Universität Berlin

- **message**: consists of three parts
  - a header, a set of properties and the body of the message
  - The header contains all the information needed to identify and route the message, including the destination (a reference to either a topic or a queue), the priority of the message, the expiration date, a message ID and a timestamp.
- **message producer**: is an object used to publish messages under a particular topic or to send messages to a queue.
- **message consumer**: is an object used to subscribe to messages concerned with a given topic or to receive messages from a queue
  - associate filters with message consumers by specifying what is known as a **message selector**: a predicate defined over the values in the header and properties parts of a message
  - two modes provided for receiving messages: the program either can block using a receive operation or it can establish a message listener object which must provide an *onMessage* method that is invoked whenever a suitable message is identified.

Code Example - Producer

Java class FireAlarmJMS

```
import javax.jms.*; import javax.naming.*;
public class FireAlarmJMS {
        public void raise() {
        try {
                Context ctx = new InitialContext();
                TopicConnectionFactory topicFactory =
                (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
                Topic topic = (Topic)ctx.lookup("Alarms");
                TopicConnection topicConn =
                topicConnectionFactory.createTopicConnection();
                TopicSession topicSess = topicConn.createTopicSession(false,
                Session.AUTO_ACKNOWLEDGE);
                TopicPublisher topicPub = topicSess.createPublisher(topic);
                TextMessage msg = topicSess.createTextMessage();
                msg.setText("Fire!");
                topicPub.publish(message);
        } catch (Exception e) {}}
```

Code Example - Consumer

Java class FireAlarmConsumerJMS

```
import javax.jms.*; import javax.naming.*;
public class FireAlarmConsumerJMS {
    public String await() {
    try {
        Context ctx = new InitialContext();
        TopicConnectionFactory topicFactory =
        TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
        Topic topic = (Topic)ctx.lookup("Alarms");
        TopicConnection topicConn =
        topicConnectionFactory.createTopicConnection();
        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;}}
```

Freie Universität Berlin

Indirect communication

Shared Memory and Tuple Spaces

Share Memory and Tuple Spaces

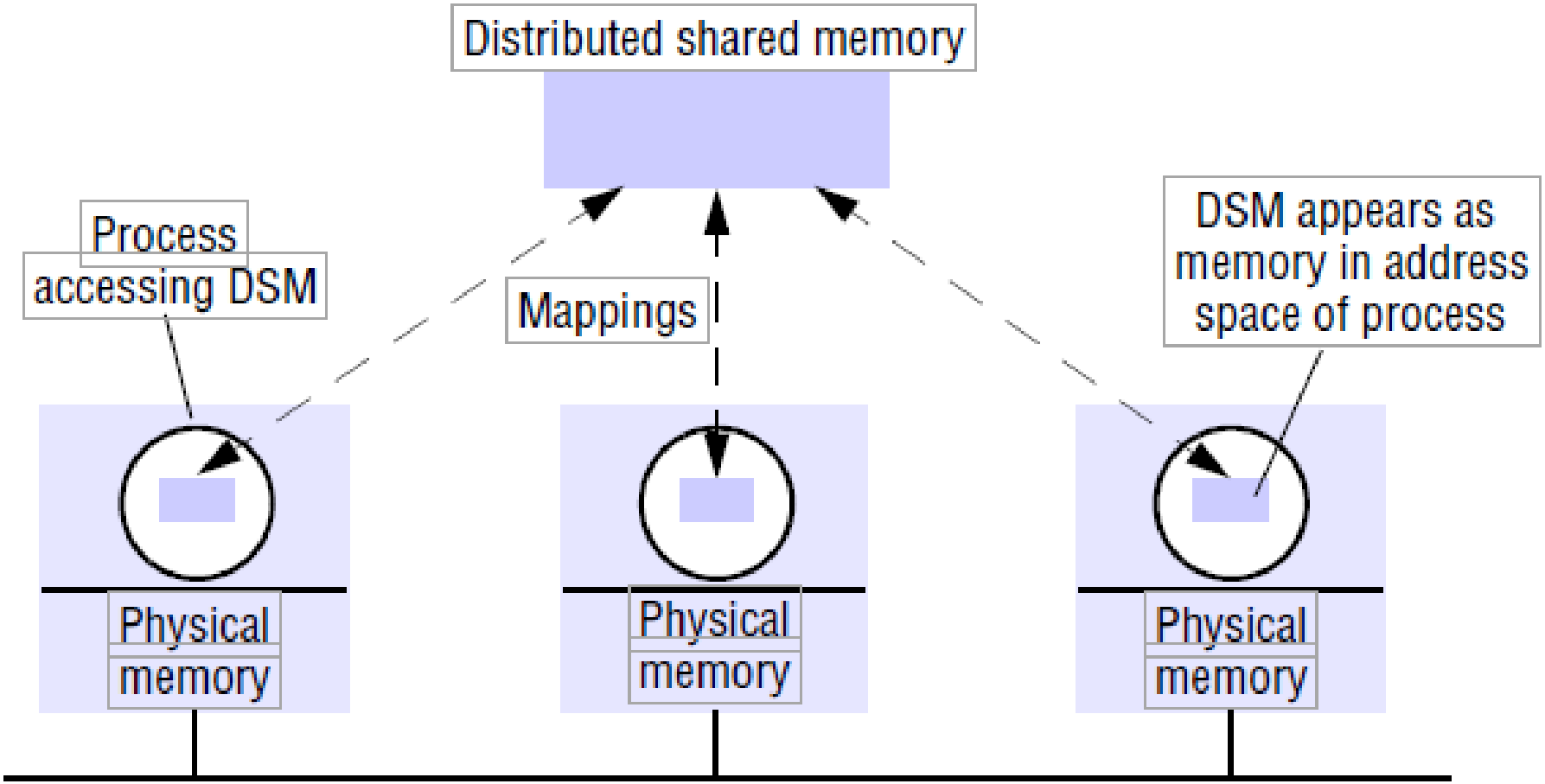
Freie Universität Berlin

- Shared memory techniques were developed principally for parallel computing

- **Distributed shared memory** (DSM) is an abstraction used for sharing data between computers that do not share physical memory
  - distributed shared memory operates at the level of reading and writing bytes and is accessed by address

- **tuple spaces** offer a higher-level perspective in the form of semi-structured data and are associative, offering a form of **content-addressable memory**

Distributed Shared Memory



Message Passing vs. DMS

Freie Universität Berlin

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 non-overlapping lifetimes

Tuple Spaces

The programming model

Tuple Spaces

Freie Universität Berlin

- **tuple space** programming model, processes communicate through a tuple space – a **shared collection of tuples**
- Tuples in turn consist of a sequence of one or more **typed data fields** such as <"sid", 1964> and <4, 9.8, "Yes">
- Processes share data by accessing the same tuple space
  - ***write*** operation, ***read*** operation, ***take*** operation (returns and removes a tuple)*

    * In Linda: out, rd and in
    - reading or removing a tuple from tuple space, a process provides a **tuple specification**, e.g. take(<String, integer>) or take(<String, 1958>), and the tuple space returns any tuple that matches that specification
    - read and take operations both block until there is a matching tuple in the tuple space
  - processes have to replace tuples in the tuple space instead of modifying them -> **tuples are immutable**

Properties of Tuple Spaces

Freie Universität Berlin

- **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: e.g. scoping with multiple tuple spaces; distributed tuple spaces; more operations, tuples with object types rather than flat data types

Tuple Spaces

Implementation issues

Tuple Spaces – Programming Issues

- Centralized vs. Decentralized
- **Replication** – different strategies:
  - **State-machine approach**: Tuple spaces behaves like a state machine, maintaining state and changing this state in response to events received from other replicas or from the environment
  - **Partitioning into Views and Tuple Sets**: updates are carried out in the context of the current view (the agreed set of replicas) and tuples are also partitioned into distinct tuple sets based on their associated logical names (designated as the first field in the tuple). The system consists of a set of workers carrying out computations on the tuple space, and a set of tuple space replicas.

Case Study: Java Spaces

Java Spaces

- Specification for tuple space communication developed by Sun [java.sun.com X, [java.sun.com VI]
  - implementations of JavaSpaces e.g. GigaSpaces [www.gigaspaces.com] and Blitz [www.dancres.org])

- Programming with Java Spaces:
  - create any number of instances of a space, where a space is a shared, persistent repository of objects
  - item in a JavaSpace is referred to as an **entry**: a group of objects contained in a class that implements *net.jini.core.entry.Entry*
  - API provides operations defined on JavaSpaces

Overview 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

Code Example - Entry

Java class AlarmTupleJS

```
import net.jini.core.entry.*;
public class AlarmTupleJS implements Entry {
        public String alarmType;
        public AlarmTupleJS() { }
        public AlarmTupleJS(String alarmType) {
                this.alarmType = alarmType;
        }
}
```

Code Example – Producer

Java class FireAlarmJS

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

Code Example – Consumer

Java class FireAlarmReceiverJS

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

Freie Universität Berlin

Indirect Communication

Summary

So, what have we learned today?

Freie Universität Berlin

- **Message queues**
- **Distributed Shared Memory**
- **Tuple Spaces**
- Indirect Communication Properties:
  - space uncoupling in that messages are directed to an intermediary and not to any specific recipient or recipients
  - time uncoupling dependent on the level of persistency
    - Message queues, distributed shared memory and tuple spaces all exhibit time uncoupling.
  - message queues offer a programming model that emphasizes *indirect communication* (through messages or events), whereas distributed shared memory and tuple spaces offer a more *state-based abstraction*

Summary of Indirect Communication Styles

Freie Universität Berlin

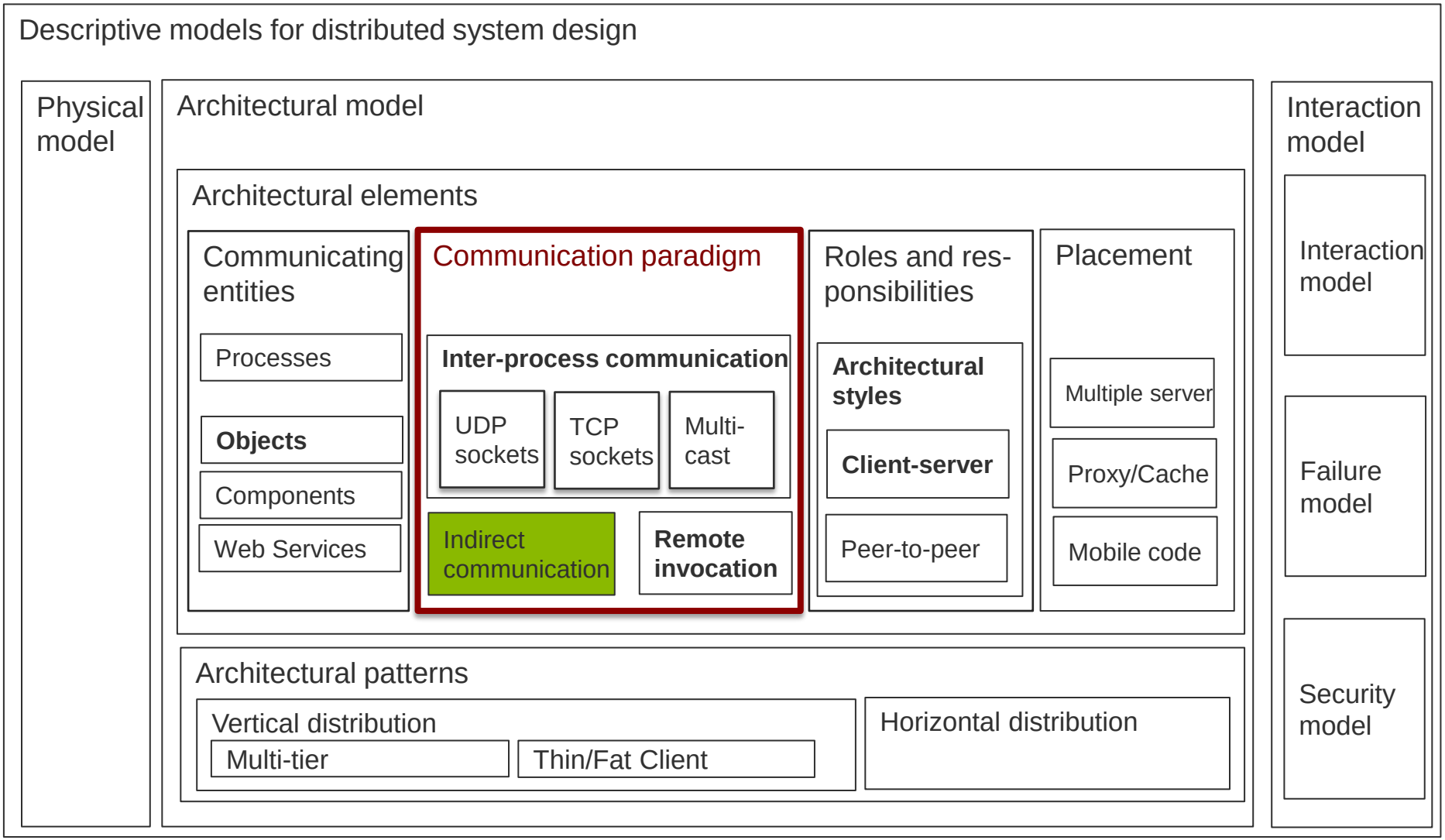
	Groups	Publish-subscribe systems	Message queues	DSM	Tuple spaces
Space-uncoupled	Yes	Yes	Yes	Yes	Yes
Time-uncoupled	Possible	Possible	Yes	Yes	Yes
Style of service	Communication-based	Communication-based	Communication-based	State-based	State-based
Communication pattern	1-to-many	1-to-many	1-to-1	1-to-many	1-to-many or 1-to-1
Main intent	Reliable distributed computing	Information dissemination, EAI, CEP	Information dissemination, EAI, CEP, transactions	Parallel and distributed computing	Parallel and distributed computing
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based	No	No	No

Questions

Freie Universität Berlin

- What are the characteristics and applications of message queues?
- Describe the programming model of message queues. What are the differences between receive, poll and notify? Describe the properties persistency, reliable delivery and transactional.
- Describe centralized and decentralized message queues. Explain the hub-and-spoke approach.
- Explain how distributed shared memory can be used for indirect communication. What are the differences to message passing approaches?
- What are tuple spaces and how can they be used for indirect communication?
- How do group communication systems, publish-subscribe systems. Distributed shared memory, message queues and tuple spaces differ in terms of main intent, space and time decoupling, style of service, communication pattern and scalability?

Our topics next week



Freie Universität Berlin

Next class

Distributed Event Based Systems

Complex Event Processing

References

George Coulouris, Jean Dollimore, Tim Kindberg: *Distributed Systems: Concepts and Design*. 5th edition, Addison Wesley, 2011.

Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114-131. DOI=10.1145/857076.857078 http://doi.acm.org/10.1145/857076.857078

Baldoni, R. & Virgillito, A., 2005. Distributed event routing in publish/subscribe communication systems: a survey. DIS Universita di Roma" La Sapienza" Tech Rep. Available at:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.1108&rep=rep1&type=pdf.

Selected slides from "Distributed Event Routing in Publish/Subscribe Systems", Roberto Baldoni, Sapienza University of Rome, 2009