Message Oriented Midleware (MOM)

September 26, 2024

Roadmap

Message-based communication

Asynchronous Communication (MOM)
- Concept
- Java Message Service
- Implementation

Further Reading

Roadmap

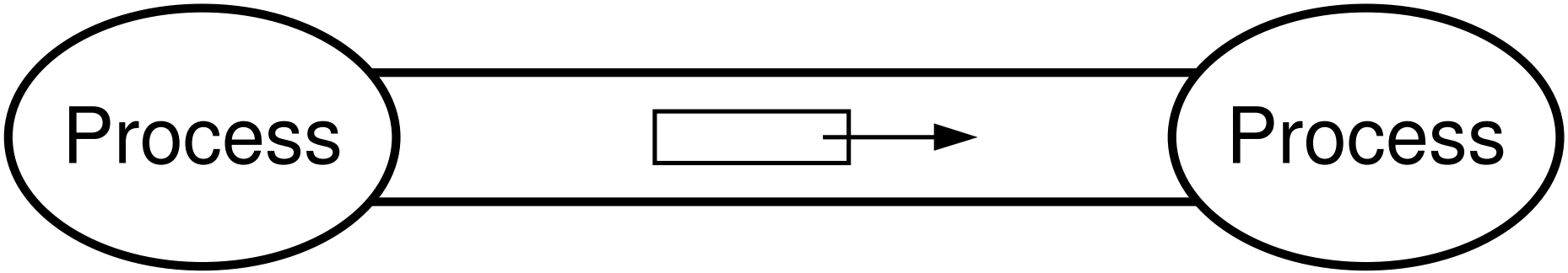
Message-based communication

- Asynchronous Communication (MOM)
  - Concept
  - Java Message Service
  - Implementation
- Further Reading

Distributed System

Definition A distributed system consists of a **collection of** distinct **processes** which are spatially separated and **which communicate with one another by exchanging messages**. (L. Lamport, "Time, Clocks and the Order of Events in a Distributed System", CACM)

- "A system is distributed if **the message transmission delay is not negligible** compared to the time between events in a single process."

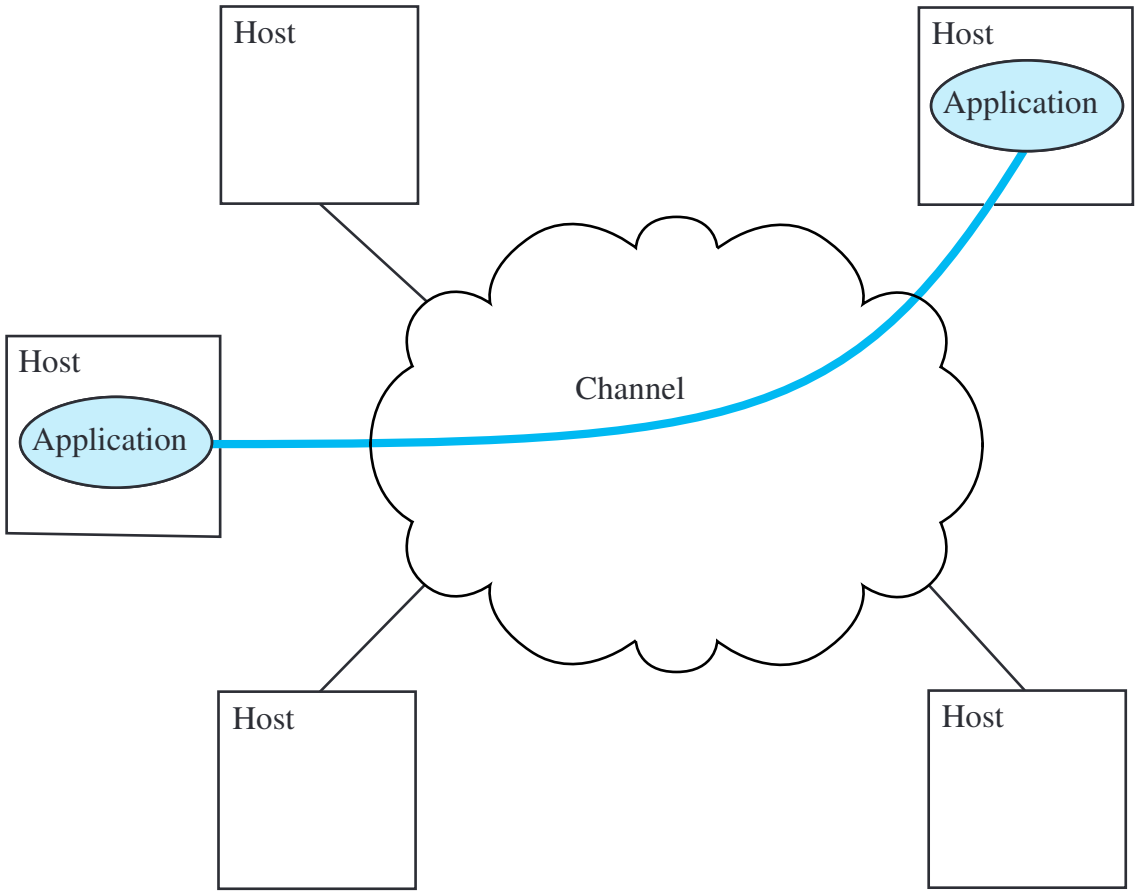


What is a message? is an **atomic** bit string

- Its format and its meaning are specified by a communications protocol

Message-based Communication and Networking

- The transport of a message from its source to its destination is performed by a computer network.



Internet Protocols

Application	Specific communication services
Transport	Communication between 2 (or more) processes.
Network	Communication between 2 computers not directly connected with each other.
Interface	Communication between 2 computers directly connected.

- On the Internet, the properties of the communication channel provided to an application depend on the transport protocol used (UDP or TCP):
  - The design of a distributed application depends on the properties provided by the chosen transport protocol

Summary of the Properties of the Internet Transport Protocols

Property	UDP	TCP
Abstraction	Message	Stream
Connection-based	N	Y
Reliability (loss & duplication)	N	Y
Order	N	Y
Flow control	N	Y
Number of recipients	1 | n	1

- The abstraction provided by TCP stems from the API, or is it intrinsic to the protocol?

TCP Reliability (Message loss)

What does this mean? Can we assume all data sent through a TCP connection will be delivered to the remote end?

TCP Reliability (Message loss)

What does this mean? Can we assume all data sent through a TCP connection will be delivered to the remote end?

What if bad things happen? E.g.:

- Networking hardware misconfiguration or failures
- Unplugged cables
- Damaged cables, e.g. by road works or shark bites

TCP Reliability (Message loss)

What does this mean? Can we assume all data sent through a TCP connection will be delivered to the remote end?

What if bad things happen? E.g.:

- Networking hardware misconfiguration or failures
- Unplugged cables
- Damaged cables, e.g. by road works or shark bites

What TCP guarantees is that the application will be notified if the local end is unable to communicate with the remote end

- Typically, `send()/write()` or `recv()/read()` will return an error code. E.g. `ENOTCONN` (the connection will be closed).
- TCP **cannot** guarantee that there is no data loss.

It is up to the application to deal with this

- A web browser may just report the problem to the user
- If the application does not interface with the user, to try to connect again is a possibility

  But TCP does not re-transmit data that was lost in other connections

TCP Reliability (Message duplication)

Why not always re-transmit messages that might have not been delivered?

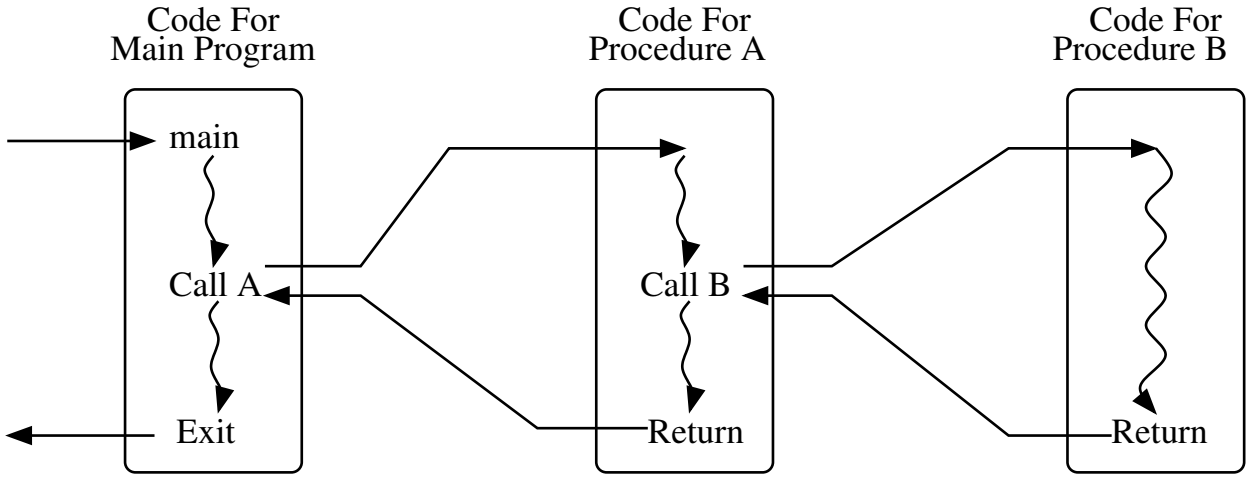
Issue The re-transmitted message may have been delivered before the connection was closed

- TCP is not able to filter data duplicated by the application
  - Only duplicated TCP segments
- This may be an issue. E.g. if the duplicated data is a request for a **non-idempotent** operation such as:
  - A credit/debit operation
  - A purchase order

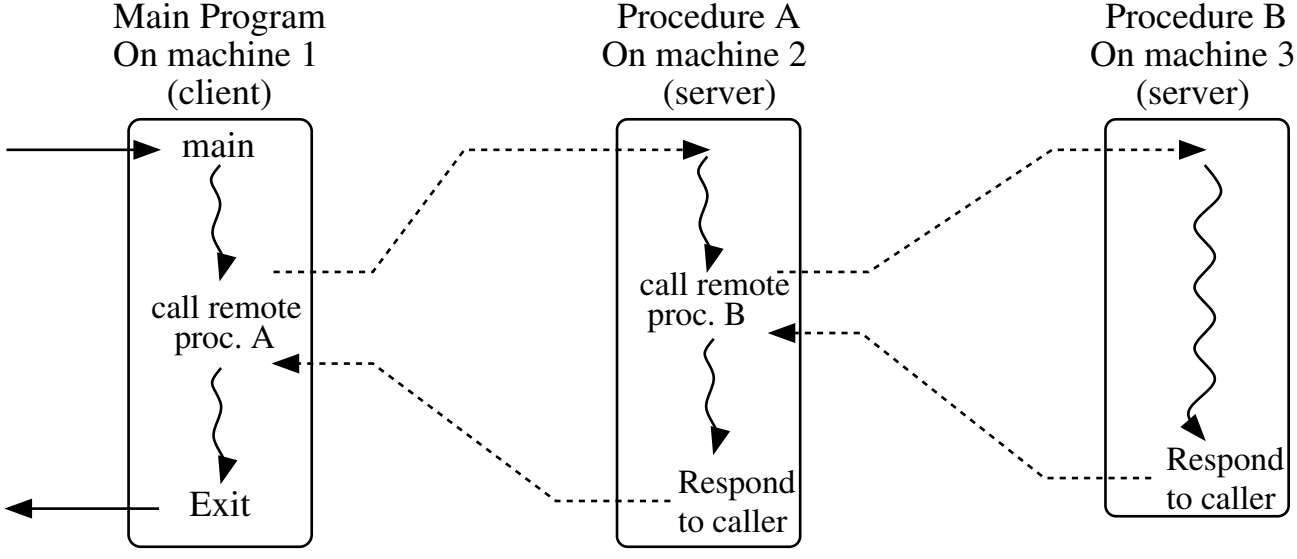
In this case the application may need to synchronize with the remote end to learn if there was some data loss in either direction

RPC: the Idea

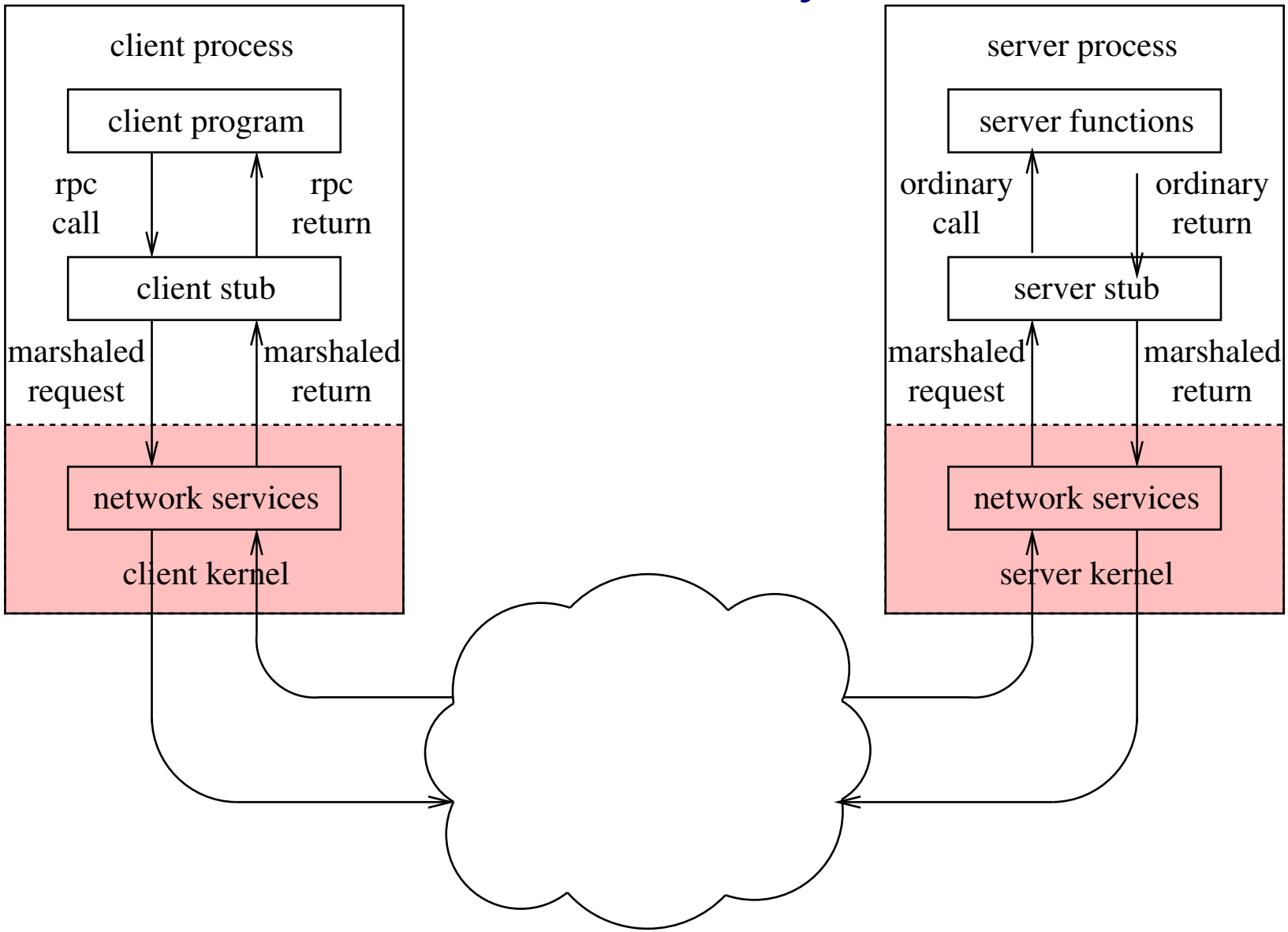
Local procedure call:



Remote procedure call:



Typical Architecture of an RPC System



Obs. RPC is typically implemented on top of the transport layer (TCP/IP)

Client Stub

Request

1. Assembles message: **parameter marshalling**
2. Sends message, via `write()/sendto()` to server
3. Blocks waiting for response, via `read()/recvfrom()`
   - Not in the case of **asynchronous RPC**

Response

1. Receives responses
2. Extracts the results (**unmarshalling**
3. Returns to client
   - Assuming **synchronous RPC**

Server Stub

Request

1. Receives message with request, via `read()/recvfrom()`
2. Parses message to determine arguments (**unmarshalling**)
3. Calls function

Response

1. Assembles message with the return value of the function
2. Sends message, via `write()/sendto()`
3. Blocks waiting for a new request

RPC

- RPC is a very useful communications paradigm
  - Programming distributed applications with (**non-asynchronous**) RPC would be almost as simple as programming non-distributed applications, if it were not for failures
    - How would failures affect the above time-diagrams?
- There are several more or less recent implementations:

  RPC libraries gRPC (Google), Avro (Apache Software Foundation (ASF)), Thrifty (originally Facebook, now ASF)

  Languages supporting RPC Java, Go, Erlang
- However it has its own limitations
  - RPC is not always the best approach
    - It is great for request-reply communication patterns, but even then there may be better alternatives

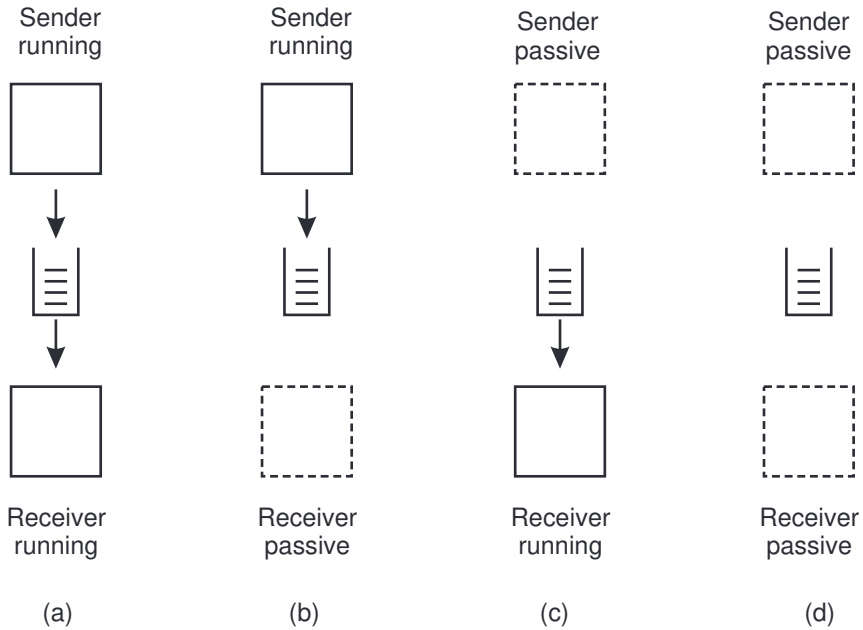
Asynchronous Communication

Problem: Communication using TCP/UDP (explicitly using messages or via RPC) require synchronization between sender and receiver

- Synchronization requires waiting, leading to wasted time

Solution: Use *asynchronous communication*

- The communicating parties need not be active simultaneously



Roadmap

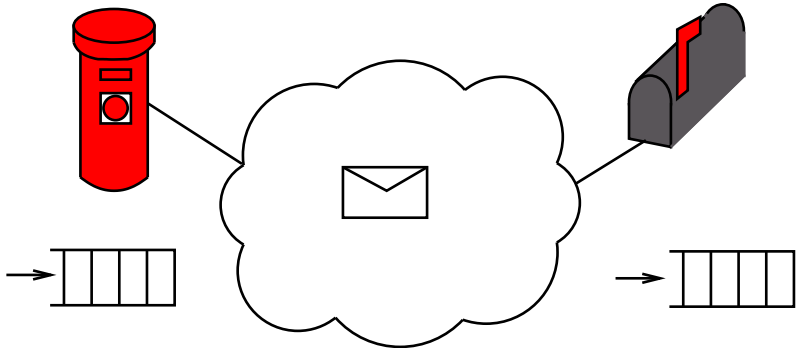
Message-based communication

Asynchronous Communication (MOM)
- Concept
- Java Message Service
- Implementation

Further Reading

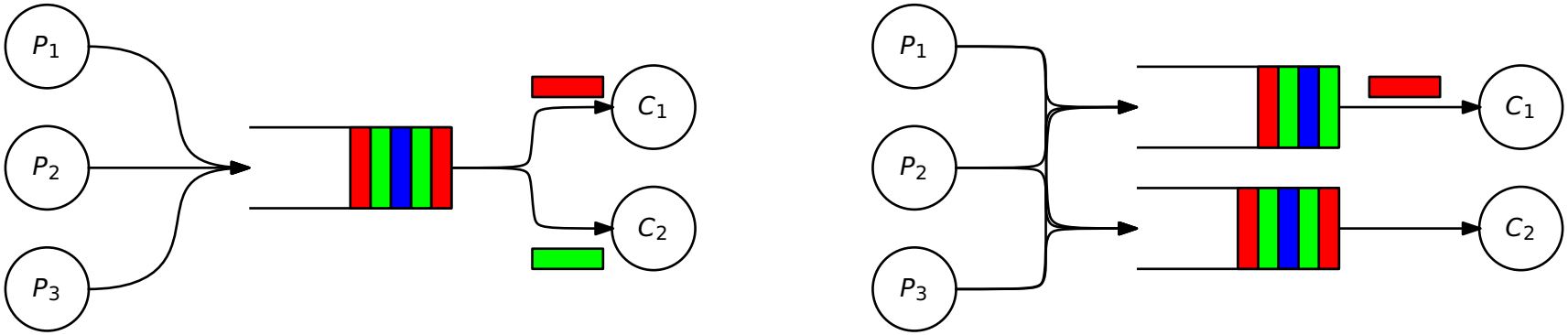
Message Oriented Middleware (MOM)

- **Asynchronous** message-based communication
  - Sender and receiver need not synchronize with one another to exchange messages
  - Communication service (middleware) stores the messages *as long as needed* to deliver them
- The service is close to that of the (snail) mail service:



- The service guarantees may vary:
  - order;
  - reliability;
- Some MOM provides also an abstraction similar to discussion fora/news groups
  - **publishers** may send messages
  - **subscribers** may receive messages.

MOM: Basic Patterns



Point-to-point The model is that of a **queue**.

- Several senders can put messages in a queue
- Several receivers can get messages from a queue

  But each message is delivered to at most one process (receiver)

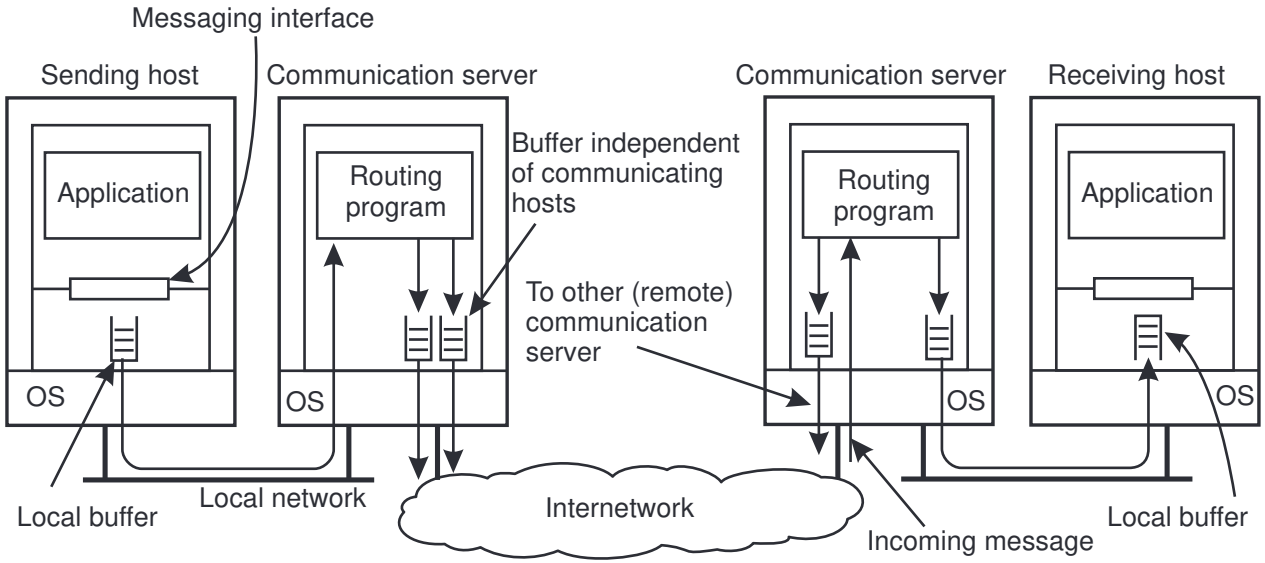
Publish-subscriber This is more like a discussion forum. Instead of queues we talk about **topics**

- Several **publishers** can put messages in a topic
- Several **subscribers** can get messages from a topic

  Unlike in queues a message may be delivered to more than one process (subscriber)

Messaging Service Implementation

- Asynchronous communication is provided by a messaging service



  - Other deployments, e.g. with a single communication server, are possible
  - At the lowest communication level, there must be synchronization between sender and receiver

Asynchronous Communication Applications

- This type of communication is appropriate for applications when the sender and receiver are **loosely coupled**. Some examples:

  Enterprise Application Integration
  Workflow applications
  Microservices
  Message based communication between people
  - *Email*, *SMS*;
  - Instant (real-time) messaging;

Java Message Service (JMS)

- JMS is an **API for MoM**, originally specified for J2EE (now Jakarta EE, because Oracle holds the Java trademark):
  - It allows Java applications to access MOM in a **portable** way
  - It provides a maximum common divisor of the functionality provided by well known MOM providers (IBM MQSeries, TIBCO)
- JMS is representative of the MOM functionalities that may be useful for developing enterprise applications
  - JMS can be integrated with the Java Transaction Service, and therefore take advantage of transactions

Note JMS has been replaced by Jakarta Messaging.

But In these transparencies we still use JMS

- Actually, the API still uses JMS
- Jakarta Messaging 3.1 Specification is mostly identical to JMS Specification v2.0 rev. A

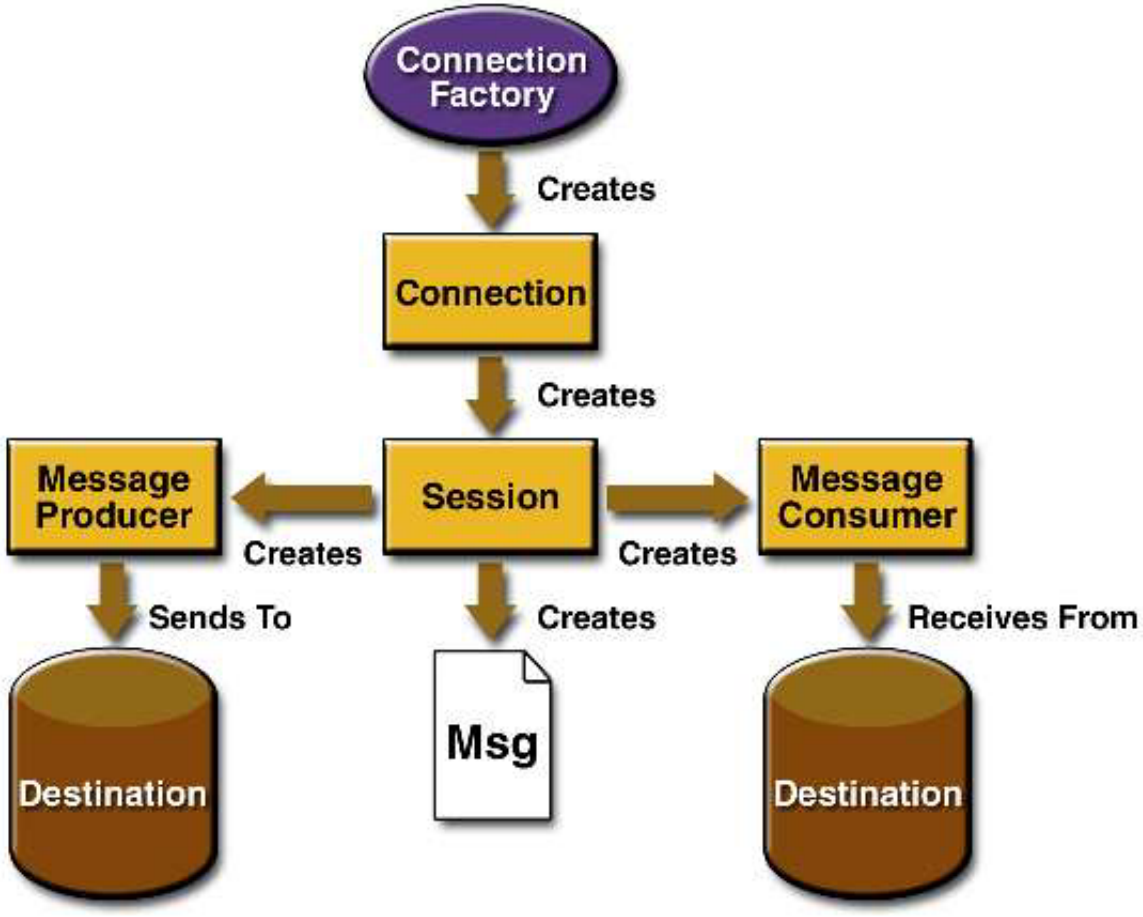
JMS Architecture and Model

- JMS supports two types of **destinations**:
  - Queues (for single-destination communication)
  - Topics (for multi-destination communication)
- JMS defines 2 fundamental components:

  JMS Provider i.e. the MOM service implementation;
  - It includes client-side libraries

  JMS Client i.e. an application that sends/receives messages to a **destination** via a **JMS provider**
- JMS specifies the API, and its semantics, that a JMS provider offers to a client
- To use the JMS, a client must first set up a **connection** to the provider
  - This is not a TCP connection, but it may be built on top of TCP
- Clients send/receive messages to/from **destinations** in the context of a **session**,
  - Sessions are created in the context of a connection

JMS Model



Source: Sun

- Each of these "boxes" corresponds to a Java type (i.e. class or interface) defined in JMS's classic API (specified in JMS 1.1)
  - Jakarta Messaging 3.1 provides a simplified API, which was introduced in JMS 2.0

JMS Messages

- JMS messages have 3 parts:

  Header: is a set of fields necessary for identifying and routing messages;
  - This set is defined in the JMS specification
    - `JMSDeliveryMode`, `JMSMessageId`, `JMSExpiration`, `JMSRedelivered`, `JMSPriority` are some of the 11 header fields

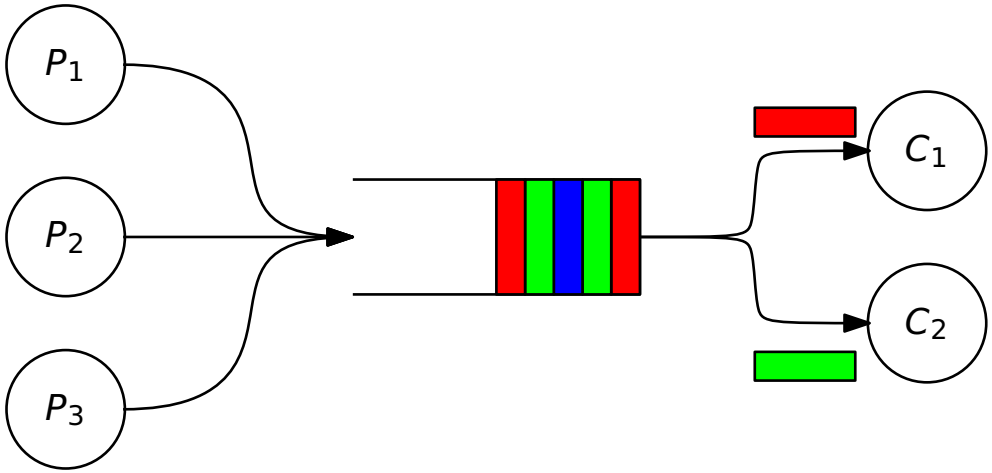
  Properties: these are optional fields that logically belong to the header – i.e. they are meta-data
  - A **property** is a `(key,value)` pair

    key is a string, which must obey some rules

    value can be of one of several primitive types, as well as `String` or `Object` classes
  - Properties are defined by the applications
    - Essentially they are an extension mechanism, allowing a client to add fields to the header

  Body: data to exchange. Can be typed.
- JMS does not specify the format of the messages on the wire
  - **JMS specifies an API not a protocol**

JMS Queues



- Match the queue model described above:
  - Several senders/producers can put messages in a queue
  - Several receivers/consumers can get messages from a queue

    But each message is delivered to at most one receiver/consumer
    This helps improve scalability
- Queues are long lived
  - are created by an administrator, not the clients
  - are always available to receive messages, even if there are no active receivers
    - this is critical for decoupling senders from receivers

  with exception of temporary queues (each JMS connection may have one temporary queue)

JMS Queues: Communication Semantics (1/2)

	Blocking	Non-Blocking	Asynchronous
`send()`	Y		via callback
`receive()`	Y	via timeout	via callback

`send()`

Blocking `send()` blocks until message is sent

- Client may have to synchronize with JMS server (see below)

Asynchronous callback is executed after sending the message or after synchronization with JMS server

`receive()` may have timeout argument (in blocking mode)

Non-blocking with 0 valued timeout (or via `receiveNoWait()`)

Asynchronous callback is executed upon message reception

JMS Queues: Communication Semantics (2/2)

Reliability depends mostly on the **delivery mode**, which may be set per message, e.g. in the `send()` call:

`PERSISTENT` ensures **once-and-only-once** semantics, i.e. the failure (?crash?) of the JMS provider must not cause a message to be lost or to be delivered twice.

- Requires the JMS server to store the message in non-volatile storage
- Requires the client to synchronize with the JMS server

`NON_PERSISTENT` ensures **at-most-once** semantics

- Message needs not survive a JMS server crash
- But JMS is expected to tolerate common network failures

Essentially, these alternatives provide different trade-offs between **reliability** and **performance**

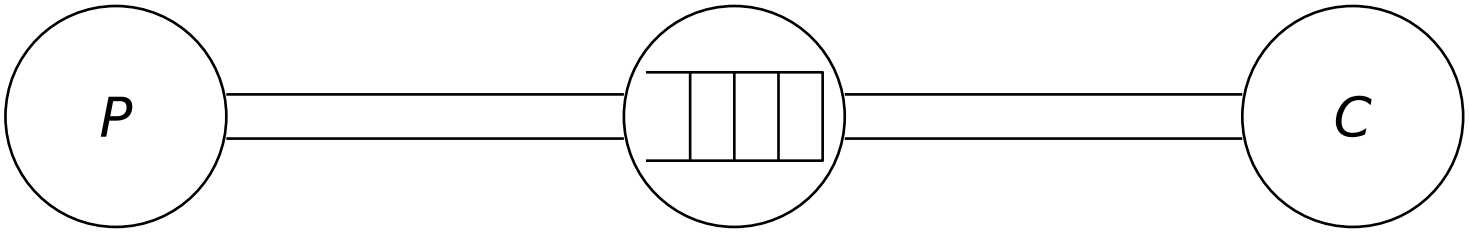
- If multiple clients consume messages from a given queue, then a client may not receive all messages.

JMS Queues: `PERSISTENT` Deliv. Implement. (1/5)

Implementing `PERSISTENT` delivery is not trivial

- A distributed system is characterized by **partial failures**

Let's assume for simplicity's sake, that there is just one server between producer and consumer



The channel between a client (either producer or consumer) and the JMS server **can loose messages**, even if the provider uses TCP for communication with the server. E.g.:

- The message producer sends the message, but communication problems cause the message to be lost and ...
  - **Producer must receive confirmation from JMS server**
- A similar scenario may happen in the channel between the JMS server and the message consumer
  - **The consumer must acknowledge the reception**, before the JMS server can dispose of the message

JMS Queues: `PERSISTENT` Deliv. Implement. (2/5)

Consumer acknowledgment behavior is set per **session**.

- Consumer acknowledgment is used to ensure that a message is delivered to one consumer

There are 3 (+1, as we'll explain below) modes:

`AUTO_ACKNOWLEDGE` the JMS session, i.e. the provider, automatically acknowledges upon a successful return from either `receive()` or the reception callback

`DUPS_OK_ACKNOWLEDGE` the JMS session lazily acknowledges the delivery of messages.

- The provider may deliver a message, without sending an ACK to the server

`CLIENT_ACKNOWLEDGE` it is up to the client to acknowledge the delivery of messages.

- The provider does not send ACK, the client does it whenever it sees fit, by calling the `acknowledge` method of `Message`
  - Acknowledgment of a message, implicitly acknowledges previously received messages

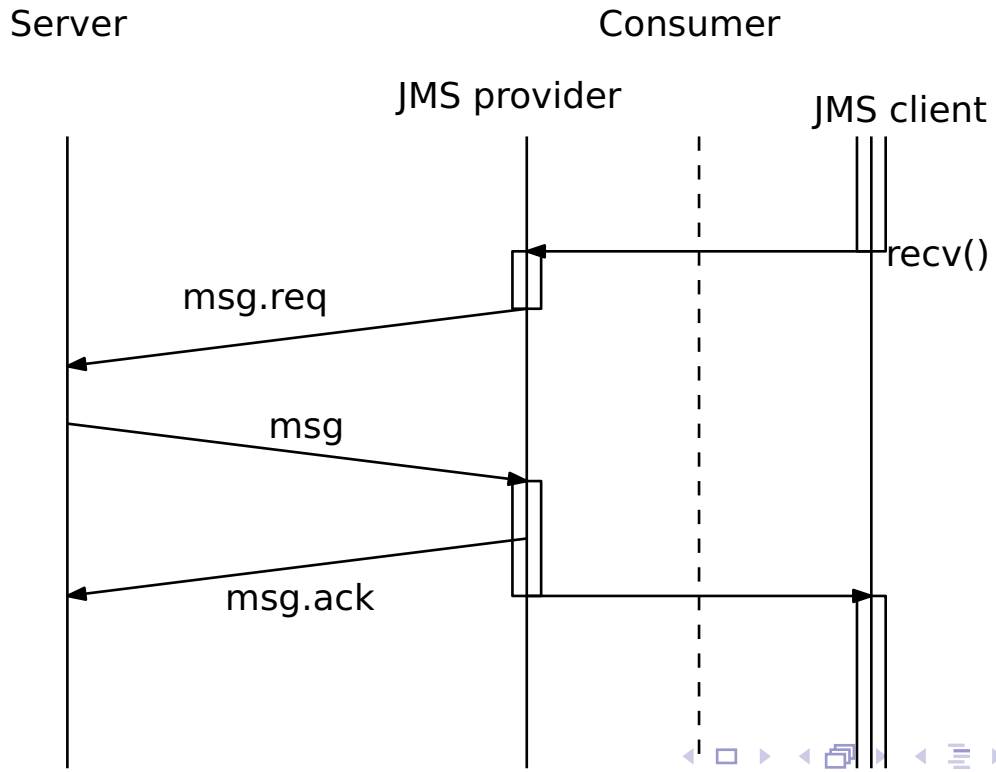
JMS Queues: `PERSISTENT` Deliv. Implement. (2/5)

Consumer acknowledgment behavior is set per **session**.

- Consumer acknowledgment is used to ensure that a message is delivered to one consumer

There are 3 (+1, as we'll explain below) modes:

`AUTO_ACKNOWLEDGE` the JMS session, i.e. the provider, automatically acknowledges upon a successful return from either `receive()` or the reception callback



JMS Queues: `PERSISTENT` Deliv. Implement. (2/5)

Consumer acknowledgment behavior is set per **session**.

- Consumer acknowledgment is used to ensure that a message is delivered to one consumer

There are 3 (+1, as we'll explain below) modes:

`AUTO_ACKNOWLEDGE` the JMS session, i.e. the provider, automatically acknowledges upon a successful return from either `receive()` or the reception callback

`DUPS_OK_ACKNOWLEDGE` the JMS session lazily acknowledges the delivery of messages.

- The provider may deliver a message, without sending an ACK to the server

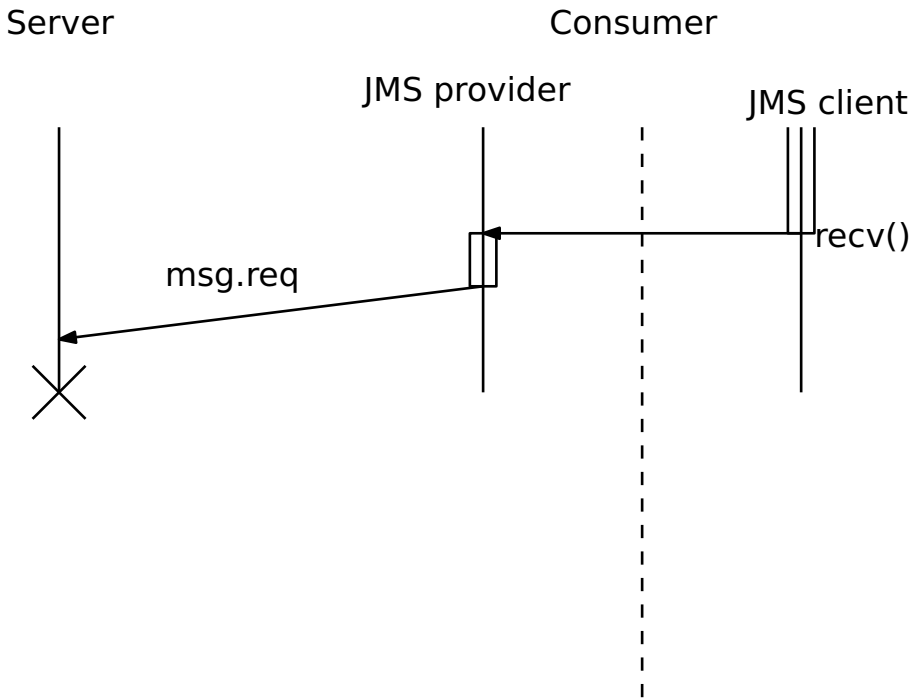
`CLIENT_ACKNOWLEDGE` it is up to the client to acknowledge the delivery of messages.

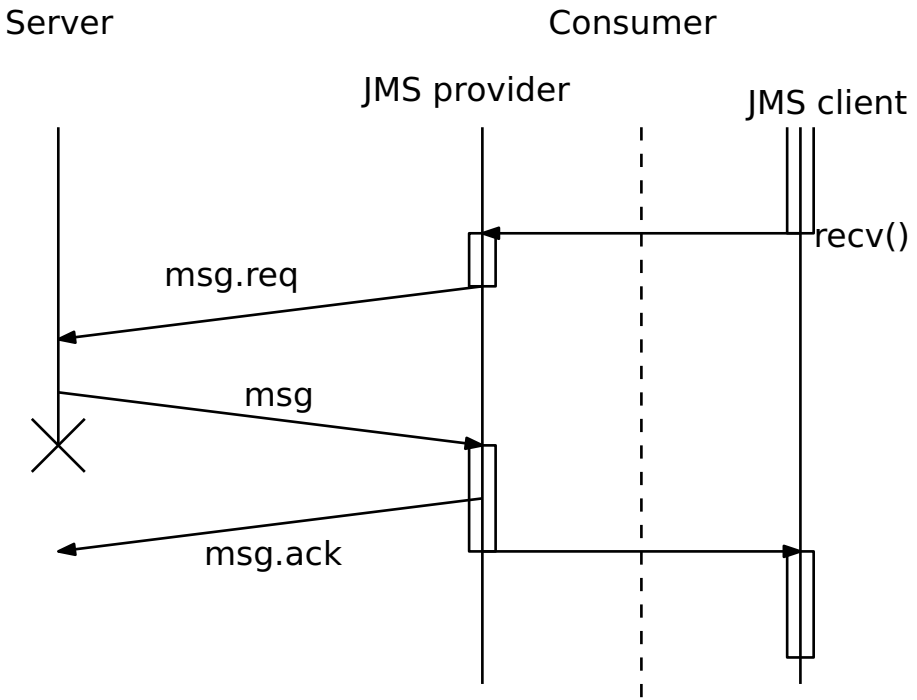
- The provider does not send ACK, the client does it whenever it sees fit, by calling the `acknowledge` method of `Message`
  - Acknowledgment of a message, implicitly acknowledges previously received messages

JMS Queues: PERSISTENT Deliv. Implement. (3/5)

Issue: upon recovery of the server there may be uncertainty wrt message delivery. Before the failure, some messages

- May have not been sent
- Or their acknowledgment may have been lost

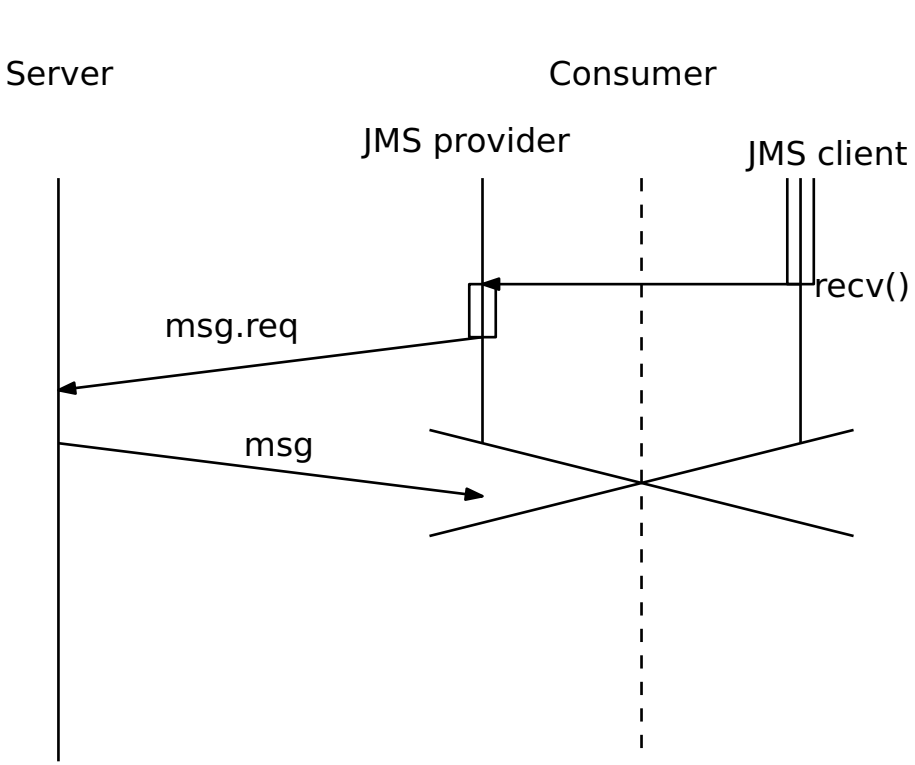


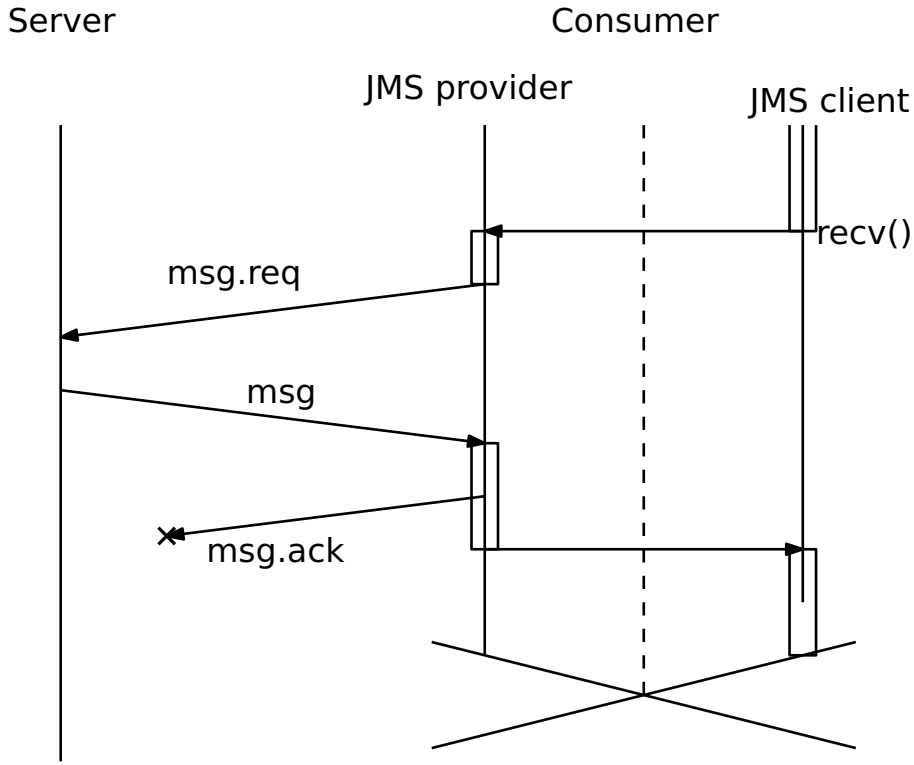


JMS Queues: PERSISTENT Deliv. Implement. (4/5)

Upon failure of the consumer the provider may be uncertain about message delivery. Some messages

- May have not been delivered
- Or their acknowledgment may have been lost





- Also possible (with AUTO_ACKNOWLEDGE): JMS server receives ACK, but consumer crashes before processing message

JMS Queues: `PERSISTENT` Deliv. Implement. (5/5)

Upon recovery the provider may be uncertain about message delivery. Some messages

- May have not been delivered
- Or their acknowledgment may have been lost

Solution: the provider resends messages whose delivery it is uncertain about with:

`JMSRedelivered` header field set;
`JMSXDeliveryCount` header property with the appropriate value

- It should be incremented every time the provider may have delivered the message
- For a redelivered message its value must be larger or equal to 2.

Consumer learn about messages that it **may** have delivered before the failure

- The application may have more information than the JMS provider to deal with this uncertainty

JMS Queues: Once-and-only-once Guarantees

- JMS' **once-and-only-once** delivery is not **exactly once**
  - If a message may be a duplicate, the consumer is notified.
- Apache's Kafka claims to support **exactly once** delivery
  - Neha Narkhede, *Exactly-Once Semantics Are Possible: Here's How Kafka Does It*
- But Kafka has the advantage that its **topics** are logs (implemented on non-volatile storage) whose records (messages/events) may persist for a long time (configurable)
  - A consumer can read a record, even if it subscribes to a topic long after that record was added to the topic
  - Each consumer has an offset pointing to the next message in a topic (more or less), and it can reset it to another value.
- Exactly once delivery on Kafka is built on top of:

  Idempotent send operations with no duplication on one partition
  - A Kafka topic may have several partitions, each of which is a log

  Transactions to write atomically to different partitions
  - Writing happens not only on a send, but also on a receive (as it modifies the partition's offset)

JMS Queues: Session-based Transactions

- Session-based transactions are the fourth session mode. I.e., it (`SESSION_TRANSACTED`) cannot be used with the other session modes, e.g. `AUTO_ACKNOWLEDGE`
- All messages sent/received within a session execute in the scope of a **transaction**
  - The key property here is **atomicity**
- To terminate the **current** transaction (and start the next one), a JMS client must call

  `commit()` so that all messages sent are added to the destination queue(s), and to acknowledge the delivery of all messages received.

  `rollback()` to cancel the sending of all messages sent in the scope of the current transaction, as well as the delivery of all messages received (what does this mean?)

JMS Queues: Distributed Transactions

Session-based transactions just ensure atomicity in the sending/receiving of messages

- The uncertainty regarding the delivery of messages may still occur
  - E.g., if there is an error in `commit()`

Distributed transactions provide stronger guarantees especially wrt processing

- Does not avoid uncertainty about message delivery in some failure scenarios
- Requires the use of the Java TransactionsAPI
- A JMS server should support the `XAResource` API, i.e. play a role similar to a DB.

JMS Queues: Message Order

Order JMS also provides some order guarantees.

- Messages sent in the context of a session to a queue are delivered in the sending order
  - This applies only to messages with the same delivery mode, e.g. `NON_PERSISTENT` message may be delivered ahead of an earlier `PERSISTENT` message
- However, it makes no guarantees wrt messages sent by different sessions

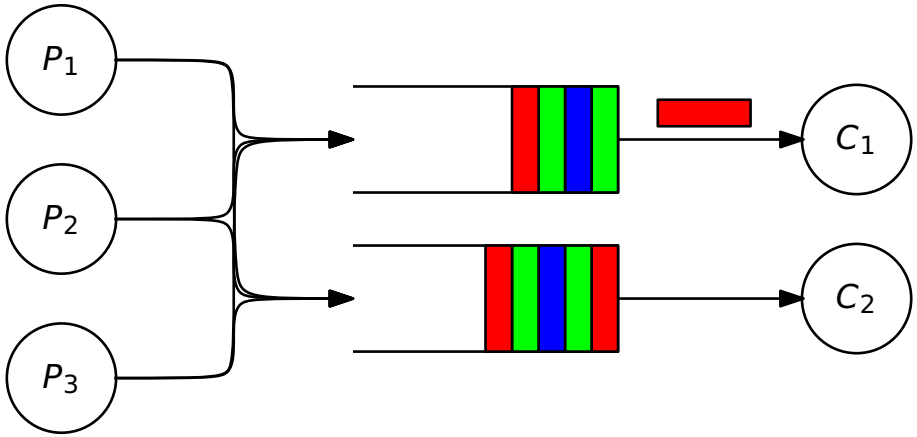
In addition, these guarantees are affected by other JMS **features**:

- Higher **priority** messages may jump-ahead of lower priority
- Messages with later **delivery time** may be delivered after messages with an earlier delivery time
- Message **selectors** (provided as arguments of `receive()`) may also affect the order in which messages are delivered

Also note that:

- If multiple clients consume messages from a queue, then a client may not receive all messages.

JMS Topics



- Support the publish-subscribe pattern, as defined above:
  - Several **publishers** can put messages in a topic
  - Several **subscribers** can get messages from a topic

  Unlike in queues a message is delivered to more than one process (subscriber)
- Topics are long lived (like queues)
  - are created by an administrator, not the clients
  - are always available to receive messages, even if there are no active receivers
    - this is critical for decoupling senders from receivers

  with exception of temporary topics (each JMS connection may have one temporary topic)

JMS Topics: sending and receiving messages

- Sending/receiving messages to/from a topic use the same API as that used for queues:

	Blocking	Non-Blocking	Asynchronous
`send()`	Y		via callback
`receive()`	Y	via timeout	via callback

`send()`

- Sender may specify (both for topics and queues)

  Earliest delivery time message cannot be delivered before (via a `MessageProducer`'s `setDeliveryDelay` method)

  Latest delivery time message should be dropped (via `send()`'s `TTL` argument)

`receive()`

- Receiver may filter messages (both from topics or queues) using a

  Message selector condition on the values of header fields and message properties, passed as argument when a `MessageConsumer` is created

JMS Topic Subscription

- Consumers use **subscriptions** to receive (all) messages sent to the respective topics
  - If a message selector is specified, some messages may be skipped
- A subscription may be:

  Unshared can have only one active consumer at a time
  Shared can have more than one active consumer
  - Each message is delivered to only one consumer
  - This helps improve scalability
- Furthermore, a subscription can be:

  Durable once created it exists until explictly deleted
  Non-Durable exists only while there is an active consumer
  - But the topic continues to exist
- Subscription identification (different from topic identification):

	Unshared	Shared
Non-durable	–	Name [+ Client id]
Durable	Name + Client id	Name [+ Client id]

JMS Topic Subscription and Reliability

- Message reliability depends both on the message's delivery mode and on the durability of the subscription:

	Non-durable	Durable
`NON_PERSISTENT`	at-most-once (missed if inactive)	at-most-once
`PERSISTENT`	once-and-only-once (missed if inactive)	once-and-only-once

  - Durable subscriptions provide same guarantees as queues
- Like for queues, no duplication guarantees do not hold on session recovery (JMS 2.0 Rev. A, Sec. 6.2.11 & Sec. 6.2.12)
  - It is up to the client/application to filter duplicates
- Asynchronism of subscribers and publishers and communication latency (JMS 2.0 Rev. A, Sec. 4.2.3):
  - a message sent after a subscription may not be delivered;
  - a message sent before a subscription may be delivered.

Note Section numbers in Jakarta Messaging 3.1 are the same

JMS Topic Message Consumption Order

- The general guarantees for message delivery order are similar to those for queues:
  - Messages sent by a session to a topic are delivered in the sending order
    - Remember this applies only to messages with same delivery mode
  - But these guarantees are affected by other JMS features:
    - A message may jump-ahead of another with lower priority
    - The delivery time of a message may also change the delivery order
    - And so do message selectors.
- Furthermore, the order of delivery in a subscription may not match that in another subscription of the same topic (JMS 2.0 Rev. A, Sec. 6.2.9.1)
  - Message delivery order is time dependent, and is outside of control of the client application.

JMS ...

JMS is not a service but an API

- Oracle's J2EE implementation comprises a *JMS provider*.
- Open Message Queue is a reference implementation of a JMS provider
- There are several other MoM that support JMS, e.g. IBM MQS and Amazon's SQS

JMS does not support

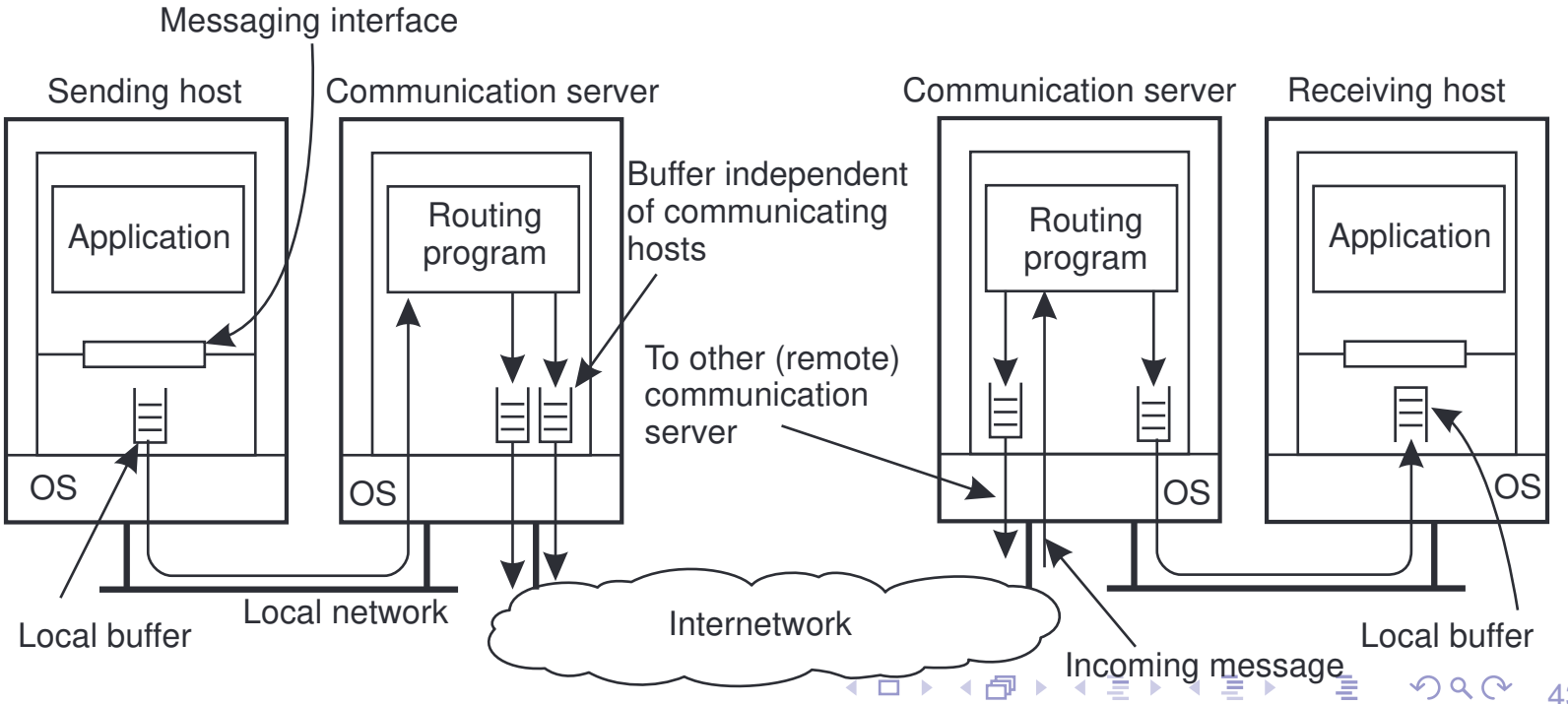
- Fault-tolerance/load balancing, i.e. does not specify how clients implementing a critical service cooperate
- Error notification, i.e. messages for reporting problems or system events to clients.
- JMS Provider administration
- Security – i.e. it does not offer an API to manage security attributes of exchanged messages

JMS promotes the portability of Java applications that use MOM

- A client that uses the JMS API, can use any conformant JMS provider

JMS: Portability vs. Interoperability

- JMS is an API
  - It allows the **portability** of applications that use MOM
- JMS is not a protocol
  - JMS does not guarantee **interoperability**
    - I.e. that a JMS provider can communicate with another JMS provider
- This may be a limitation when we need to integrate different JMS providers



Message Queuing Protocols

AMQP Advanced Message Queuing Protocol, is an open-standard protocol first aproved by OASIS and later by ISO/IEC

MTTQ at some point it was the acronym of Message Queuing Telemetry Transport, i.e. a protocol designed for industrial applications, is also an OASIS transport

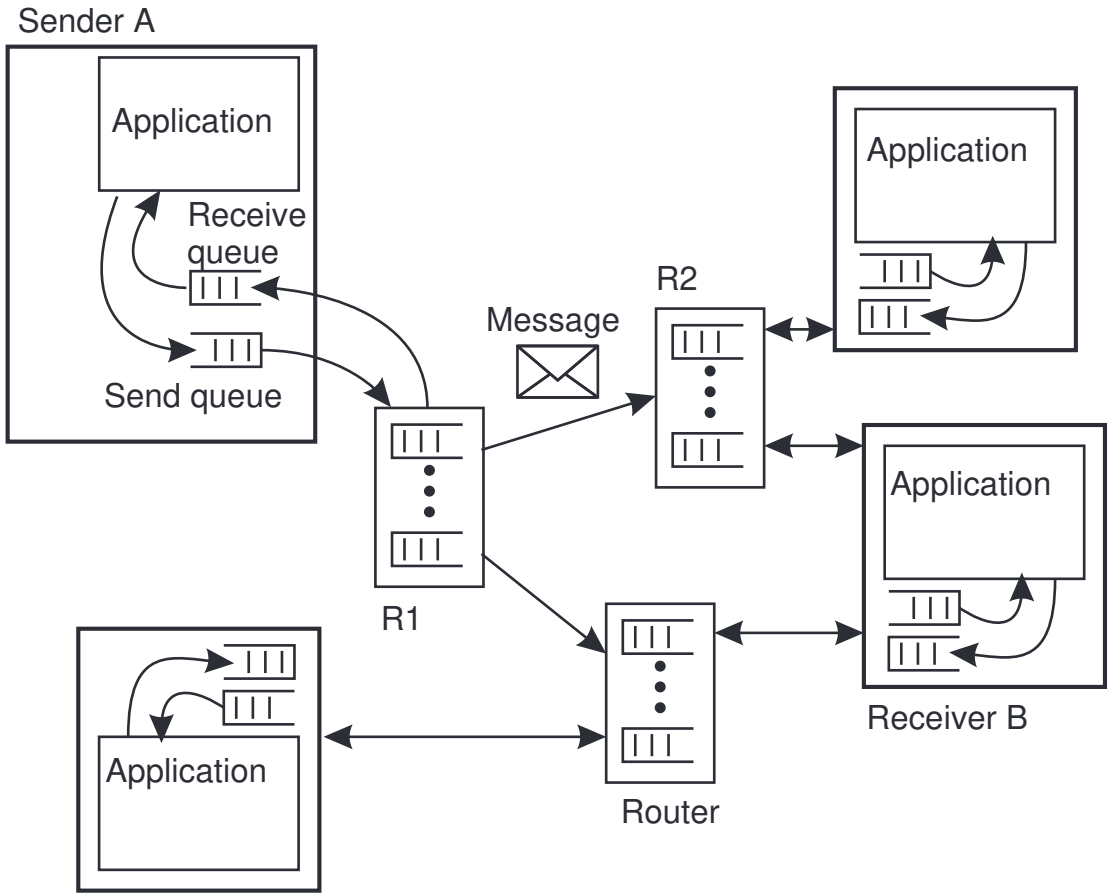
- Nowadays, it is being proposed for IoT applications

OpenWire is a public protocol used by Apache ActiveMQ (which provides a JMS API)

- But ActiveMQ also supports AMQP, MTTQ and other protocols

Architecture

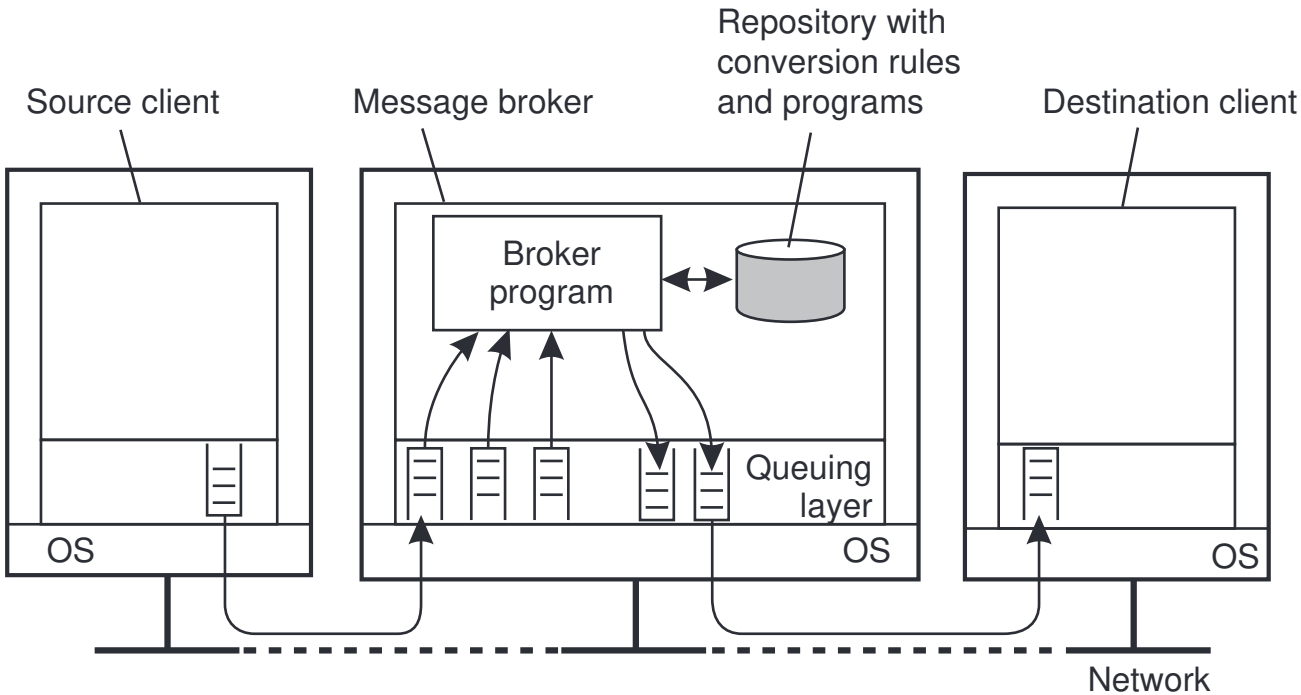
- Larger scale systems may use **message relays** to route messages to their destinations
  - E.g. if applications/services run on different data centers



- This architecture is very similar to that of SMTP, although nowadays almost every e-mail message just traverses two servers

Message Brokers

- MOM is often used for enterprise application integration. Sometimes:
  - These applications may have been designed independently
  - The syntax of the messages used by each of them may be different from one another
- **Message brokers** convert the format of the messages used by one application to the format used by another application
  - Strictly, they are not part of the communication service



Roadmap

Message-based communication

Asynchronous Communication (MOM)
- Concept
- Java Message Service
- Implementation

Further Reading

Further Reading

- van Steen and Tanenbaum, *Distributed Systems, 3rd Ed.*
  - Section 4.3 *Message-oriented communication*
- Jakarta, *Jakarta Messaging v3.1*
  - Essentially identical to:
    - Oracle, *JMS Specification v2.0 rev. A*