RMI and JMS

Jaroslav Dytrych

Faculty of Information Technology Brno University of Technology
Božetěchova 1/2. 612 66 Brno - Královo Pole
dytrych@fit.vutbr.cz

7 October 2020

Content

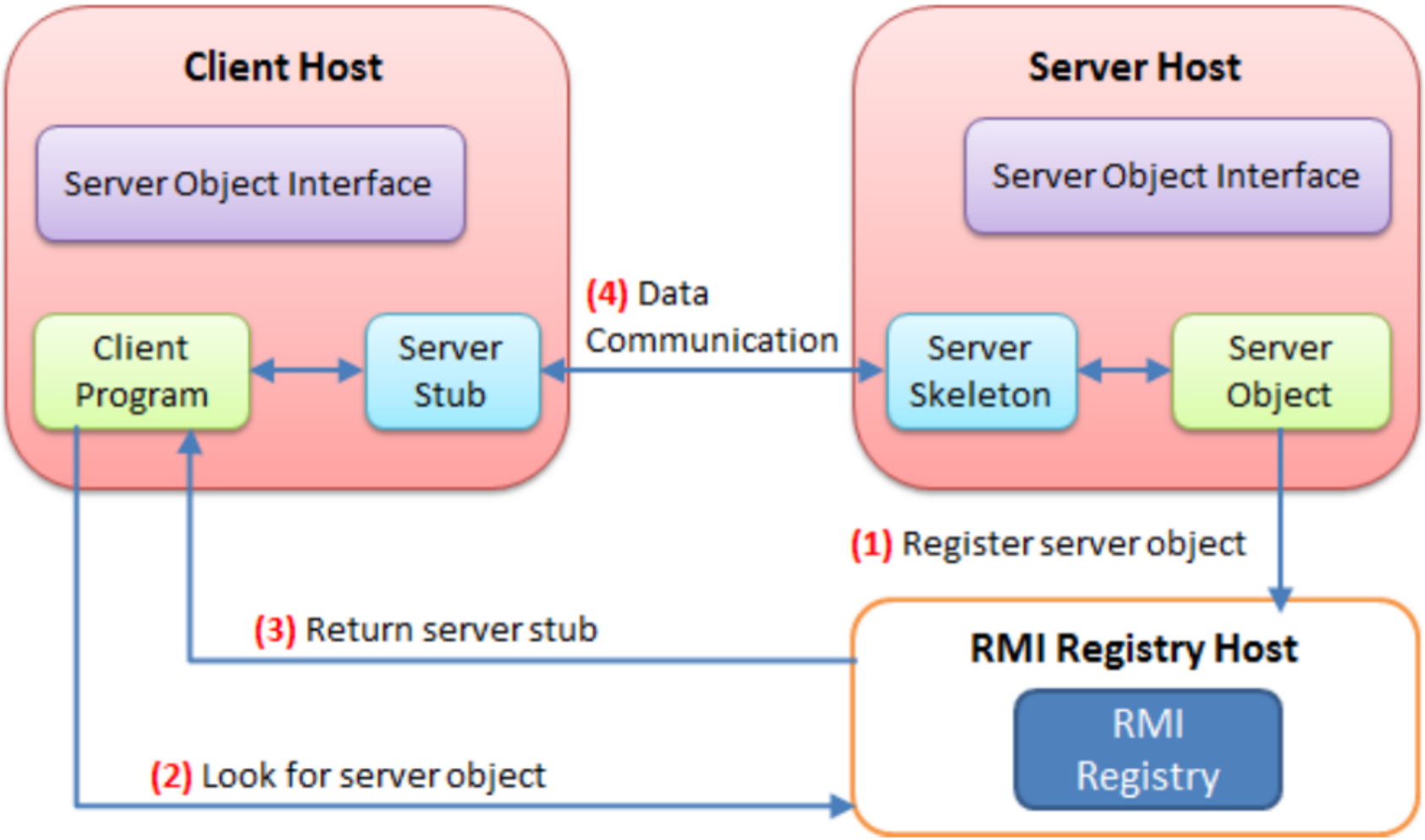
- RMI (Remote Method Invocation)
- JMS (Java Message Service)

RMI (Remote Method Invocation)

- RMI
  - Remote method invocation
  - Distributed Java
  - TCP/IP transport layer
  - Allow code that defines behavior and code that implements behavior to remain separate and to run on separate JVMs
  - Part of JDK since Java 1.1
  - Code running on one JVM may call method on other JVM
  - Client/Server architecture

- RMI uses a protocol called Java Remote Method Protocol
  - JRMP is proprietary
- For increased interoperability RMI later used the Internet Inter-ORB Protocol (IIOP)
  - IIOP is CORBA's communication protocol using TCP/IP as the transport. CORBA is Common Object Request Broker Architecture
  - language neutral protocol
  - standard way to make method calls to remote objects
  - `PortableRemoteObject` instead of `UnicastRemoteObject`
  - `-iiop` parameter of the `rmic` compiler
  - In JDK 11, the Java EE and CORBA modules were removed. These modules were deprecated for removal in JDK 9.
- RMI is all about remote calls at runtime.
  - It's not about compilation against a remote class.

RMI architecture



RMI features

- Simplifies communication with remote applications
  - local method calls
- Supports security
  - on both server and client side
- RMI layers
  - Stub
    - client side
    - creates marshall stream from client requests
    - demarshalling server response
    - object references are accessible via stub
  - Skeleton
    - server side
    - transforms marshall stream to method call

RMI principles

- RMI uses proxy design pattern
  - An object in one context is represented by another (the stub) in a separate context.
  - The stub knows how to forward method calls between the participating objects.
- A naming or directory service is run on a well-known host and port number
  - usually port 1099
- RMI includes RMI registry
  - which is actually a naming service
  - may be created directly in java or by "`rmiregistry`" program available in JDK
- Stubs and skeletons are generated
  - Static stubs and skeletons can be created by `rmic` program
    - Deprecated
  - Skeletons and stubs should be generated dynamically
    - 5 ways, e.g. subclassing `UnicastRemoteObject` and calling it's constructor (`super()`).

RMI Registry

- The RMI registry is a simple server-side bootstrap naming facility that enables remote clients to obtain a reference to an initial remote object.
- It can be started with the `rmiregistry` command which produces no output and is typically run in the background.
- Before you execute `rmiregistry`, you must make sure that the shell in which you will run `rmiregistry` either has no CLASSPATH environment variable set or has a CLASSPATH that does not include the path to any classes that you want downloaded to clients of your remote objects.
- From JDK 7 Update 21, the RMI property `java.rmi.server.useCodebaseOnly` is set to true by default. When set to false, it allows one side of an RMI connection to specify a network location (URL) from which the other side should load Java classes. If it is set to true, classes are loaded only from preconfigured locations, such as the locally-specified `java.rmi.server.codebase` property or the local CLASSPATH, and not from codebase information passed through the RMI request stream.

RMI naming

- `Naming` static class
  - `Bind` // binds the specified name to a remote object
  - `List` // returns an array of the names bound in the registry
  - `Lookup` // returns a reference, a stub, for the remote object
  - `Rebind` // rebinds the specified name to a new remote object
  - `Unbind` // destroys the binding for the specified name
- `LocateRegistry` static class
  - may create new registry
  - naming methods are available
- `UnicastRemoteObject`
  - also static class, which can export any object to be accessible on registry
  - Extend it or use `exportObject(Remote, PORT)`

Server implementation

- Shared proxy object

```
public interface Message extends Remote {
    int add(int a, int b) throws RemoteException;
}
```

- Shared proxy must be implemented

```
public class MessageImpl extends UnicastRemoteObject
                         implements Message {

    public MessageImpl() throws RemoteException {
    }
    @Override
    public int add(int a,int b) throws RemoteException {
        return a+b;
    }
}
```

- Registry is created (if not already running)

```
Registry registry = LocateRegistry.createRegistry(1099);
```

- Service is bind to given name

```
// create a new service named myMessage
registry.rebind("myMessage", new MessageImpl());
```

- Shared proxy object

```
public interface Message extends Remote {
    int add(int a, int b) throws RemoteException;
}
```

- Remote call

```
Registry myRegistry =
    LocateRegistry.getRegistry("127.0.0.1", 1099);

Message impl = (Message) myRegistry.lookup("myMessage");
System.out.println(impl.add(3, 5));
```

Server callback

- With RMI also server may initiate communication
- Communication object must implement proxy (which extends `java.rmi.Remote`)
  - This object then may be referenced via stub
- Object export via `UnicastRemoteObject`
  - Another JVM is running → use different port
  - `UnicastRemoteObject.exportObject(Remote,PORT)`
- Asynchronous messages
  - server is the origin of communication

RMI security

- SSL or any other mechanism may be used
- Initialize security manager
  - `System.setSecurityManager(new RMISecurityManager());`
  - Applets typically run in a container that already has a security manager, so there is generally no need for applets to set a security manager.
- By default, the `RMISecurityManager` restricts all code in the program from establishing network connections.
  - Naming doesn't work by default (creating registry manually approach does)

```
java -Djava.security.manager -Djava.security.policy=
policy-file MyClass
grant
{
  permission java.net.SocketPermission
  "*:1024-65535", "connect";
}
```

- Package `javax.security.manager`
- Individual for every application
- Restricts what stubs can do
  - `resolve`
  - `accept`
  - `connect`
  - `listen`
- Host can be defined by following way

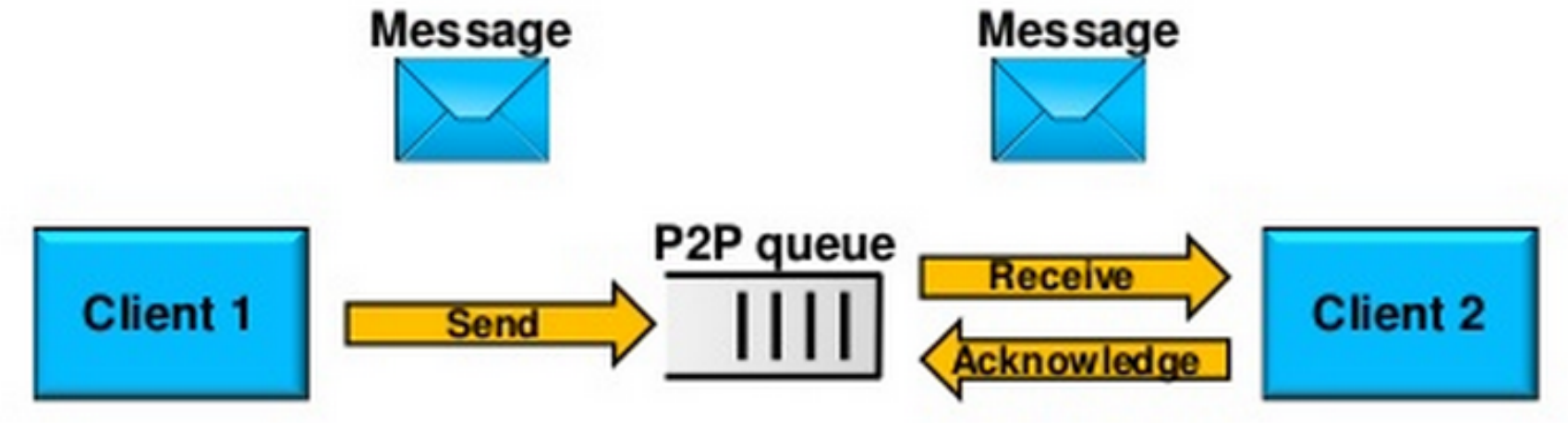
```
host = (hostname | IPv4address | iPv6reference) [:portrange]
portrange = portnumber | -portnumber | portnumber-[portnumber]
```

- RMI
  - http://docs.oracle.com/javase/tutorial/rmi/
  - http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/enhancements-7.html
  - https://docs.oracle.com/javase/8/docs/api/java/rmi/server/UnicastRemoteObject.html
  - https://docs.oracle.com/en/java/javase/15/docs/specs/rmi/index.html
  - https://docs.oracle.com/en/java/javase/11/migrate/index.html
- API
  - https://docs.oracle.com/javase/8/docs/api/
  - https://docs.oracle.com/en/java/javase/15/docs/api/index.html

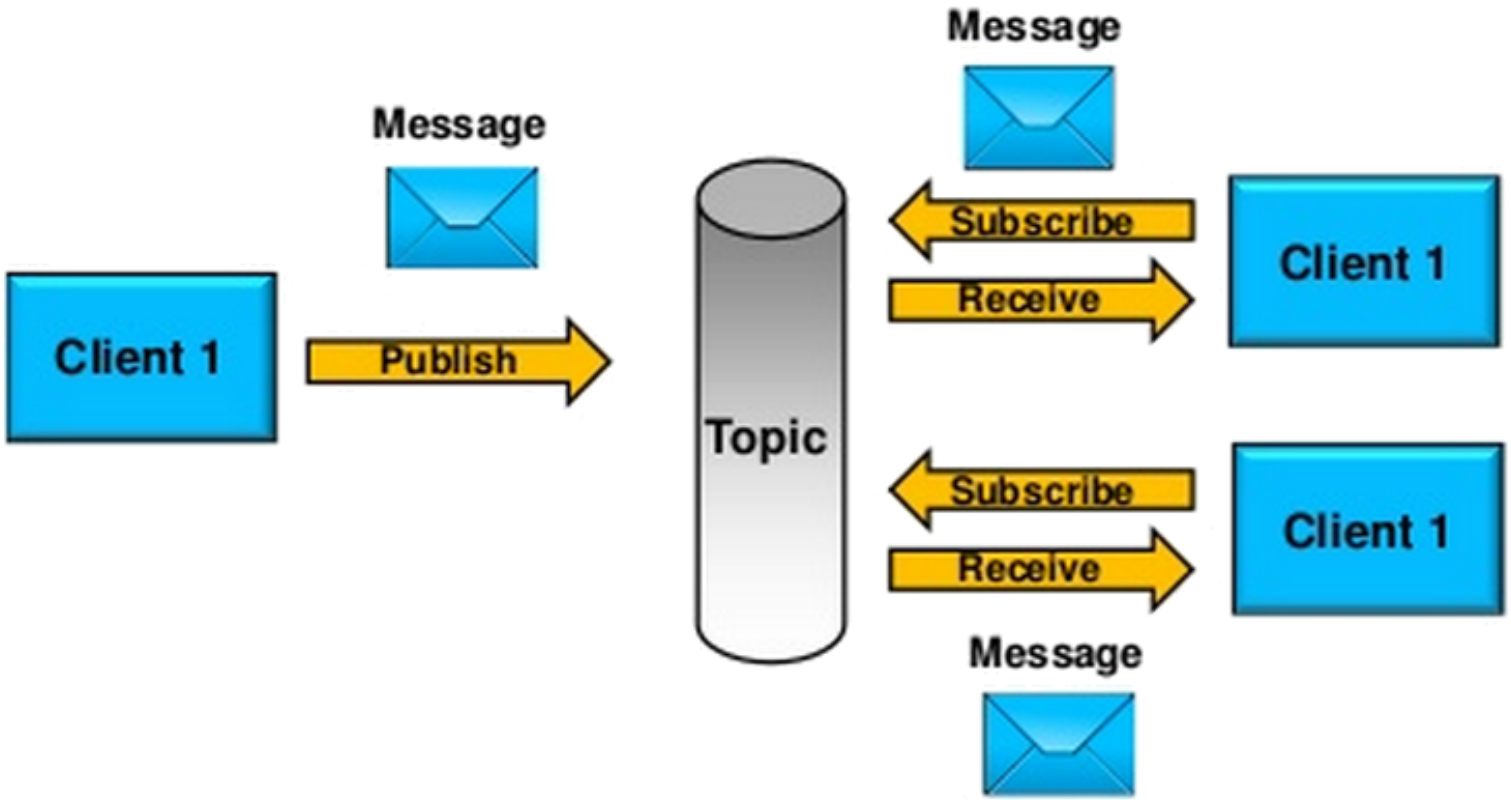
JMS (Java Message Service)

- JMS
  - Java Message Service
  - asynchronous message exchange between Java applications
  - JMS implementations are called JMS providers.
  - Different providers are not interoperable.
  - Reliable delivery.
- Providers
  - Open Message Queue (Part of GlassFish)
  - JBossMQ / JBossMessaging (Red Hat)
  - WebSphere MQ (IBM)
  - ActiveMQ (Apache project)
  - RabbitMQ (Pivotal Software)
  - ZeroMQ (iMatix Corporation)

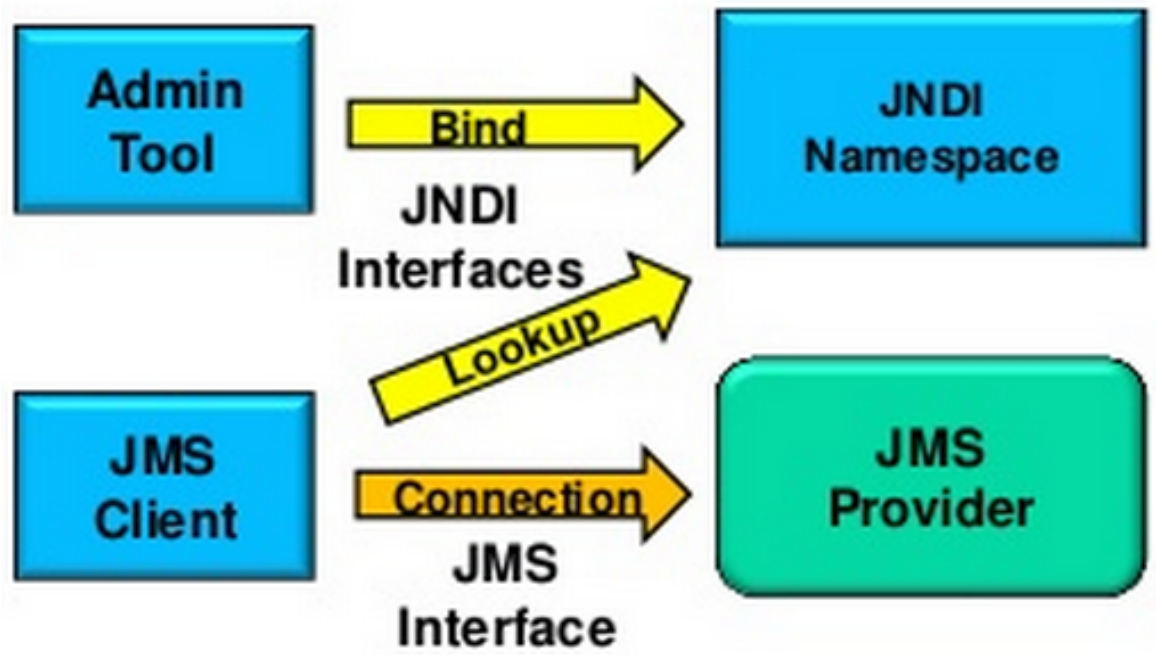
- P2P
  - Point to point domain
  - Each message has only one consumer.
  - No timing (client 1 may send a message before client 2 is started, and yet message will be delivered).
  - Each queue may have more senders.
  - Only one receiver can process the message (only once).



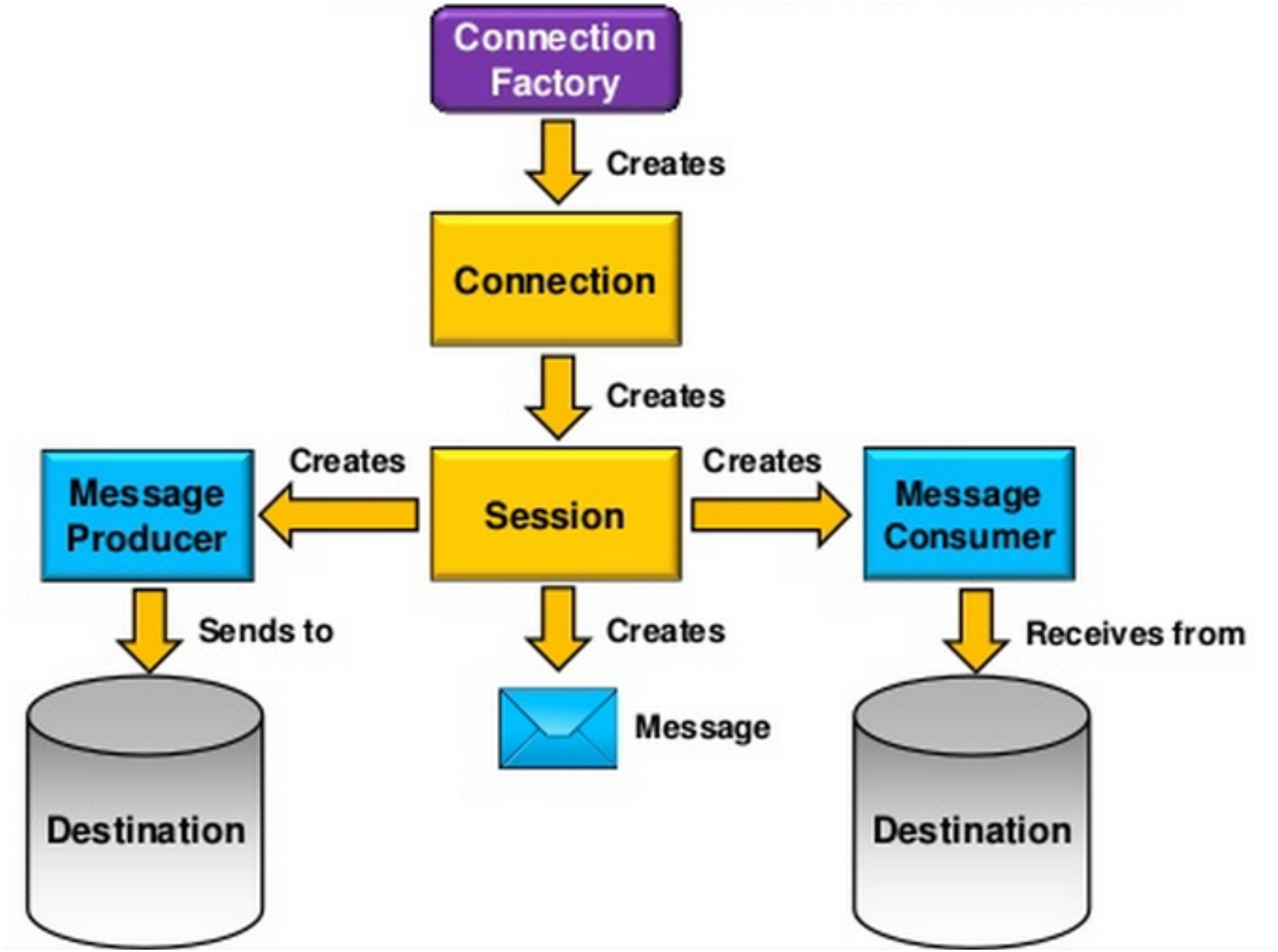
- PubSub
  - Publish-Subscribe domain
  - Multiple publishers/subscribers
  - Weak timing (no messages delivered before subscription)
  - Durable subscriptions available (survive reboot)



- JMS is an interface specification
  - Providers implement queues and topics
- Heavy use of JNDI (Java Naming and Directory Interface)
  - Connection factories (Topic or Queue factory)
  - Destinations (channels of communication)



JMS programing model



JMS programming model

- Connection factory
  - managed by JMS provider
  - `TopicFactory`
  - `QueueFactory`
- Destination
  - also managed by JMS provider
  - `Topic` and `Queue` channels (and interfaces)
  - configured on application server (not in application)
- Topic
  - Many to many (PubSub)
- Queue
  - Many to one (P2P)
  - When message is retrieved, it is deleted from the queue.

- Session
  - context to deliver and consume message
  - created from connections (factories)
  - lifecycle start and end defined
- Consumer and Producer
  - created by sessions
  - Session exists for each producer and consumer.

- `MapMessage`
  - for sending of key-value pairs
  - also sending of objects

```
MessageProducer producer = session.createProducer(queue);
MapMessage m = session.createMapMessage();
m.setIntProperty("Id", 987654321);
m.setStringProperty("name", "Widget");
m.setDoubleProperty("price", 0.99);
List<String> colors = new ArrayList<String>();
colors.add("red");
colors.add("green");
colors.add("white");
m.setObject("colours", colors);
Producer.send(m);
```

- Consumer receives `MapMessage` Object
- Getters
  - `getInt("key1")`
  - `getString("key2")`
  - `getObject("key3")`
  - `getMapNames()`

- `TextMessage`
  - simple string messages

```
queueConnectionFactory = (QueueConnectionFactory)
    jndiContext.lookup("QueueConnectionFactory");
queue = (Queue) jndiContext.lookup("myQueue");
queueConnection =
    queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
QueueSender queueSender = queueSession.createSender(queue);
TextMessage textMessage = queueSession.createTextMessage();
textMessage.setText("My message");
queueSender.send(textMessage);
```

  - Receiver

```
queueReceiver = queueSession.createReceiver(queue);
textMessage = (TextMessage) queueReceiver.receive();
String message = textMessage.getText();
```

JMS messages

- `ObjectMessage`
  - used to send serializable objects
  - `setObject(Serializable Object)`
  - `getObject()`
- `StreamMessage`
  - for sending of binary primitives
  - `getABC`, `setABC`
  - where ABC is primitive java type (`Integer`, `String`, ...)
  - It is possible to read different type than was written (conversion table exists).
  - `null` can be dangerous
    - Read is treated as calling the primitive's corresponding `valueOf(String)` with a `null` value.
    - `char` does not support a String conversion, attempting to read a `null` value as a `char` must throw a `NullPointerException`.

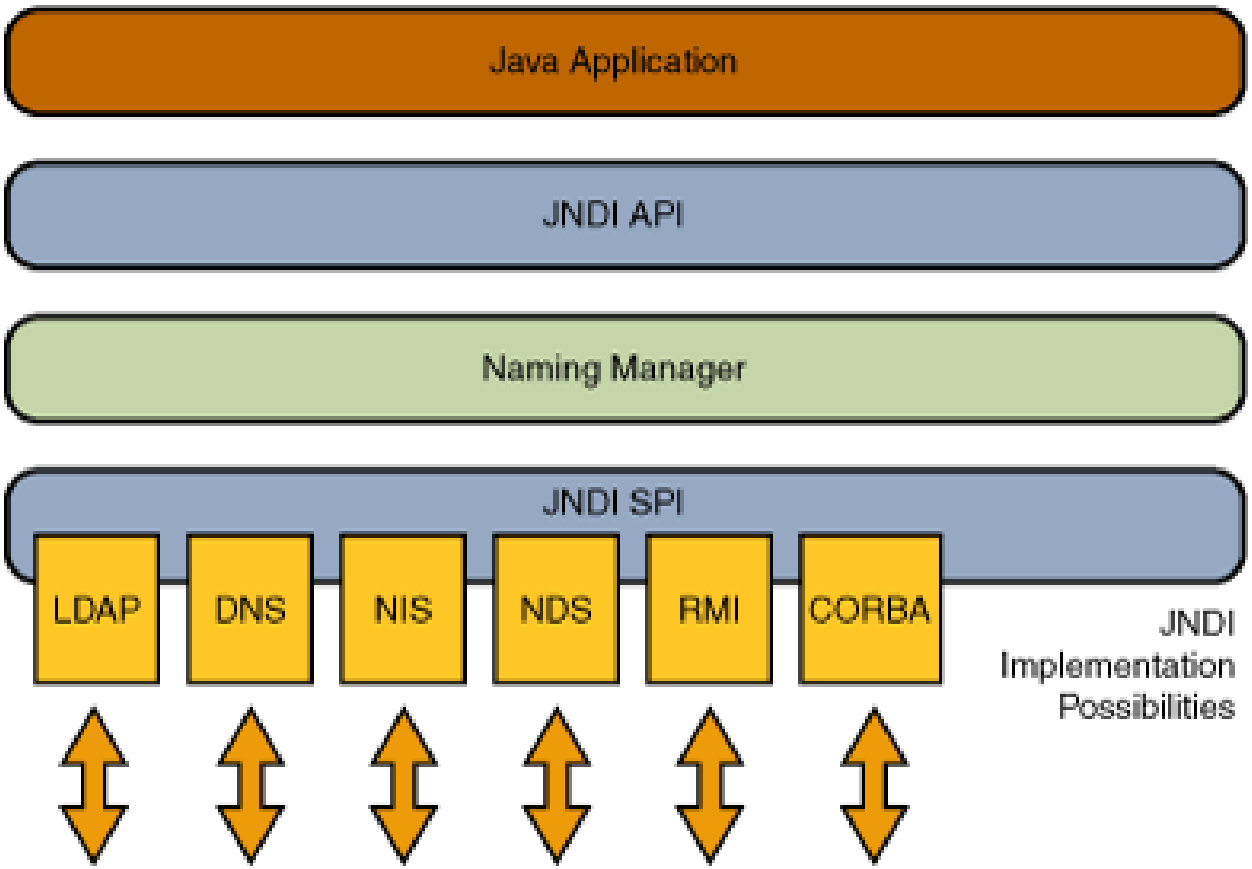
- `BytesMessage`
  - contains stream of uninterpreted bytes
  - based on `DataInputStream` and `DataOutputStream`
  - binary data
  - Methods – corresponding read/write calls
    - `writeDouble`
    - `writeBytes`
    - `writeUTF`
    - `readDouble`
    - `readBytes`
    - `readUTF`
    - `...`

- Java Naming and Directory Interface
- JMS is tightly coupled to JNDI
- Provider
  - Queue name lookups
  - an instance implementing the JNDI interface specification and services name lookups.
  - returns answers to name lookup requests.
- Initial context
  - starting point for name lookups
  - Different providers need to be parametrized with different properties.

- Association
  - associate name with object (create binding)
- Find
  - locate object specified by the name
- Context
  - set of bindings to object names
  - similar to e. g. filesystem
- Naming system
  - set of connected contexts
  - LDAP (Lightweight Directory Access Protocol)
- Namespace
  - all names in naming system
  - for example all DNS names

JNDI architecture

- Defines only interface for client accesses
- Common API for service providers
- JNDI SPI (service provider interface) allows to use different naming service providers
- Custom naming service may be developed by implementing JNDI SPI



- Access to directory services through common API
- Directories are structured trees of informations
- Directory services
  - LDAP (Lightweight Directory Access Protocol)
  - DNS
  - NIS (Network Information Service) (Oracle)
  - Microsoft Active Directory
  - IBM Lotus Notes/Domino

Message consumptions

- Asynchronous
  - A client can register a *message listener* with a consumer.
  - Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage()` method.
- Synchronous
  - A subscriber or a receiver explicitly fetches the message from the destination by calling the `receive` method.
  - The `receive` method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.

JMS features

- Additional features turned off by default
- Message acknowledgement
- Message priorities
- Persistent delivery mode
- Control of message expiration
- Durable subscriptions
- Message transactions

Message acknowledgements

- In non-transacted sessions
  - client receives the message
  - message is processed
  - acknowledgement is sent
- Acknowledgement modes
  - Auto-acknowledgement
    - An acknowledgement is sent after a successful return from `receive` function, or when listener successfully returns.
    - `Session.AUTO_ACKNOWLEDGE`
  - Client acknowledgement
    - Explicit call of message's `acknowledge` method.
    - `Session.CLIENT_ACKNOWLEDGE`
  - Lazy client acknowledgement
    - reduces JMS overhead
    - acknowledgment each time it has received a fixed number of messages, or when a fixed time interval has elapsed since the last acknowledgment (10 messages and 7 seconds)
    - Broker does not acknowledge receipt of the client acknowledgment.
    - You have no way to confirm that your acknowledgment has been received; if it is lost in transmission, the broker may redeliver the same message more than once, so duplicates may occur.
    - `Session.DUPS_OK_ACKNOWLEDGE`

Persistent delivery mode

- Persistent delivery mode
  - default mode
  - A message sent with this delivery mode is logged to stable storage when it is sent.
  - Messages can survive provider crashes.
  - Persistent delivery needs more performance.
  - Persistent delivery needs more storage.
- Non persistent delivery mode
  - does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails
  - may improve performance, but you should use it only if your application can afford to miss messages.
  - Can be enabled using:
    ```
    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    ```

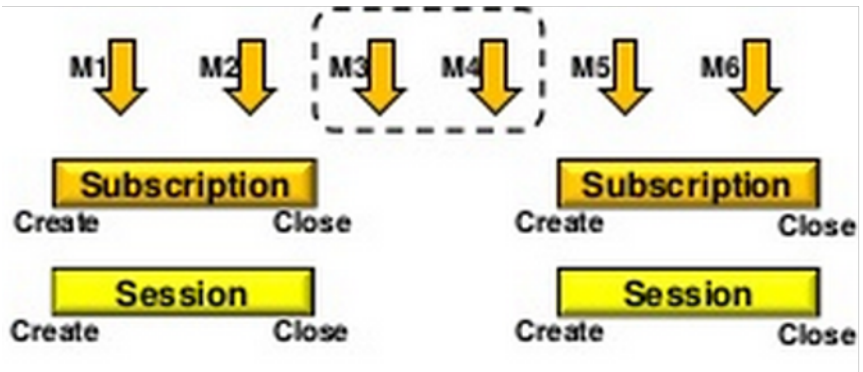
Priorities, expirations

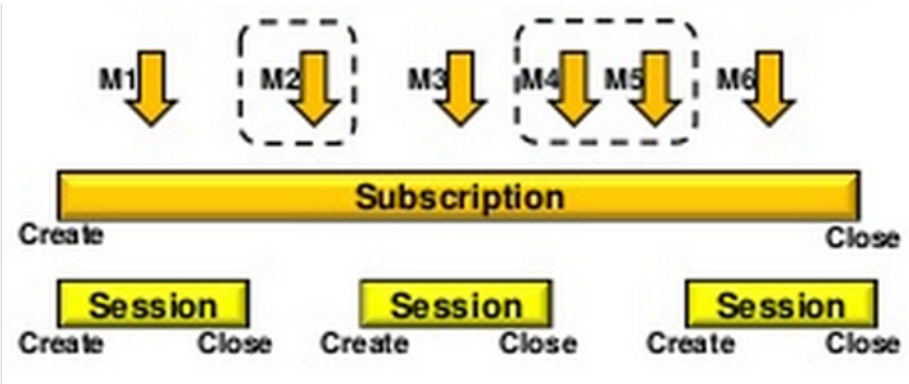
- Priorities
  - 10 levels
  - 0 – lowest priority
  - 4 – default priority
  - 9 – highest priority
  - Queues and Topics may grow big
  - `producer.setPriority(7);`
- Expiration
  - By default, messages never expire.
  - Expiration may be useful when using priorities.
  - TTL may be set to every message in milliseconds.
  - `producer.setTimeToLive(10000);`
- `topicPublisher.publish(message,`

  `DeliveryMode.NON_PERSISTENT, 8, 10000);`

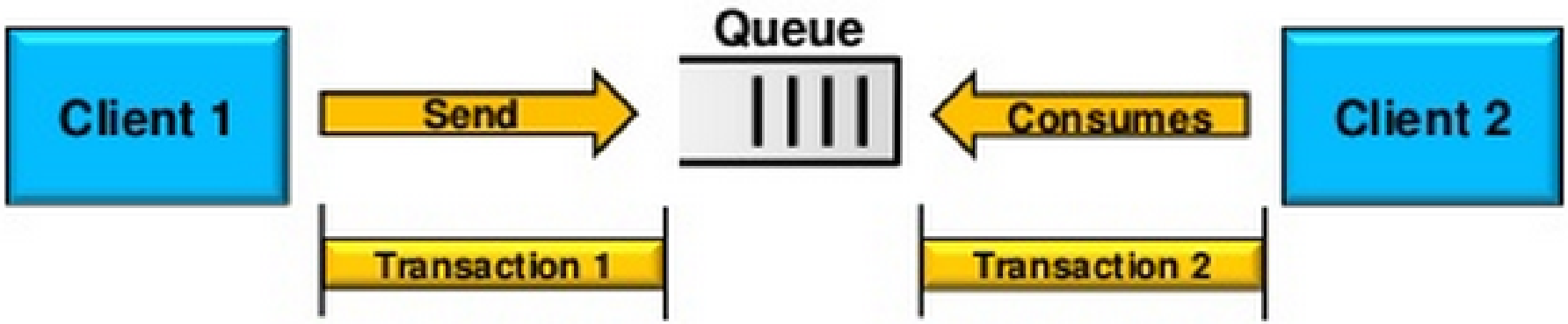
Durable subscriptions, transactions

- Default subscriptions are non-persistent
  - After each reboot the receiver must subscribe again.
  - Messages that arrived during the reboot will not be delivered.
- Durable subscriptions are persistent
  - After reboot the receiver do not need to subscribe again.
  - Messages that arrived during reboot will be delivered after a new session is created.
- Subscription is not session
  - In first case, messages M3 And M4 are not delivered.
  - In second case, messages M2, M4, M5 are received when user starts new session.

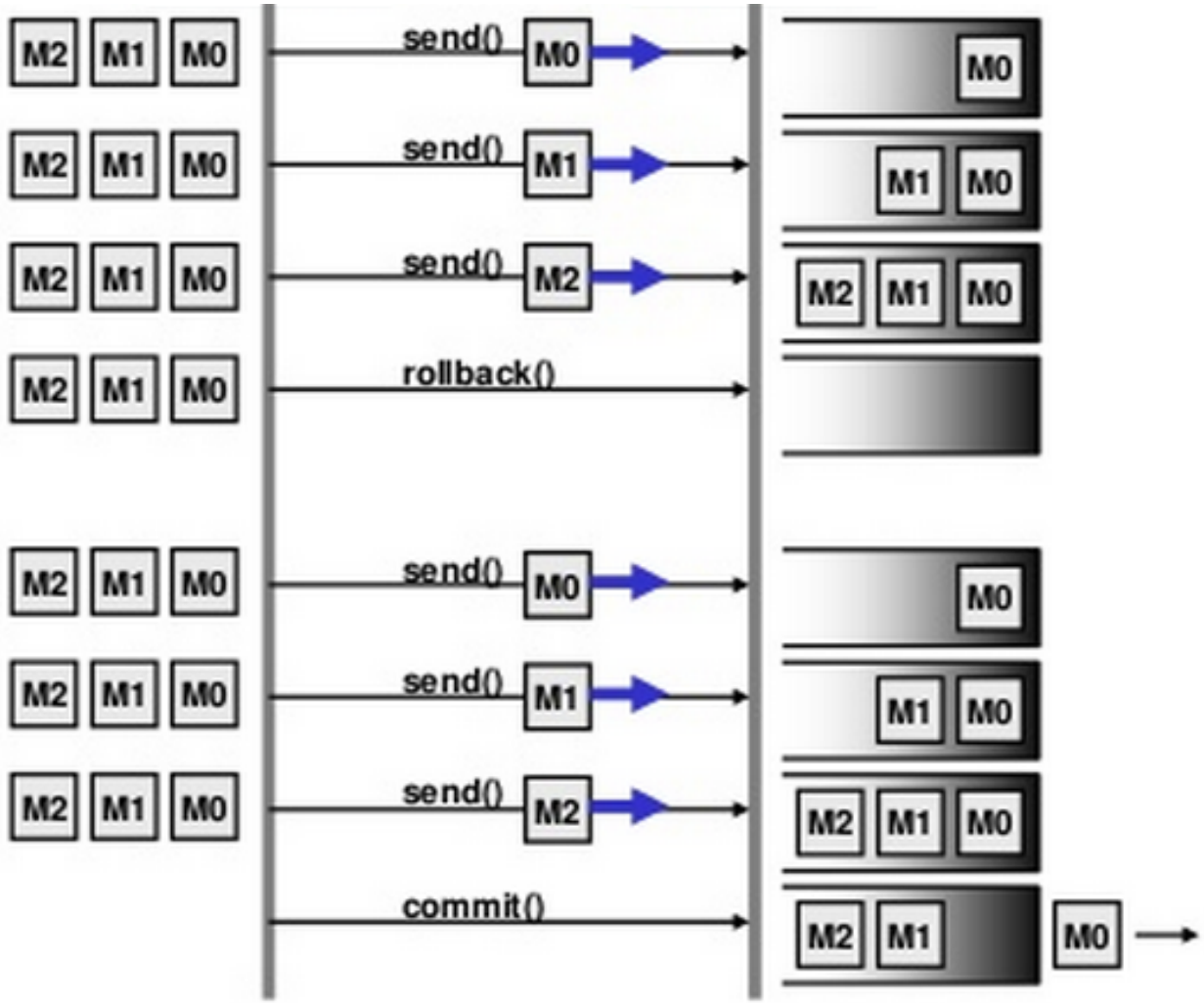




- Transactions allows grouping of operations into an atomic unit of work.
- During rollback all produced messages are destroyed, consumed messages are recovered.
- Commit means that all messages are sent and consumed messages acknowledged.
- Transactions cannot be combined with request-reply mechanism.



- Rollback
  - After rollback, all "buffered" messages are destroyed.
- Commit
  - After commit, messages begins to be retrieved.



References

- JMS Concepts
  - `https://docs.oracle.com/javaee/7/tutorial/jms-concepts.htm`
- JMS Examples
  - `https://docs.oracle.com/javaee/7/tutorial/jms-examples.htm`
- Tutorial
  - `https://www.javatpoint.com/jms-tutorial`

Thank you for your attention!