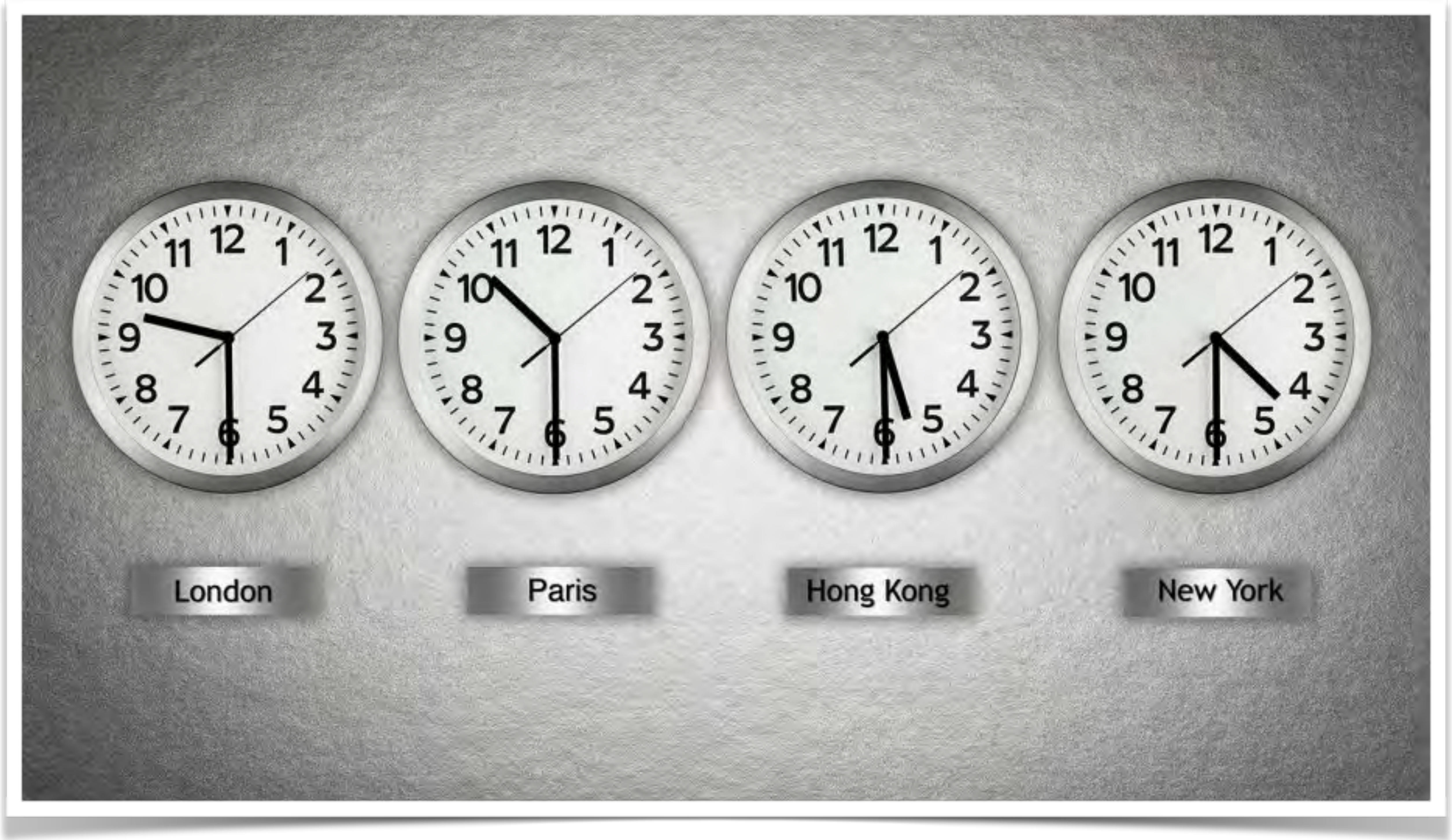
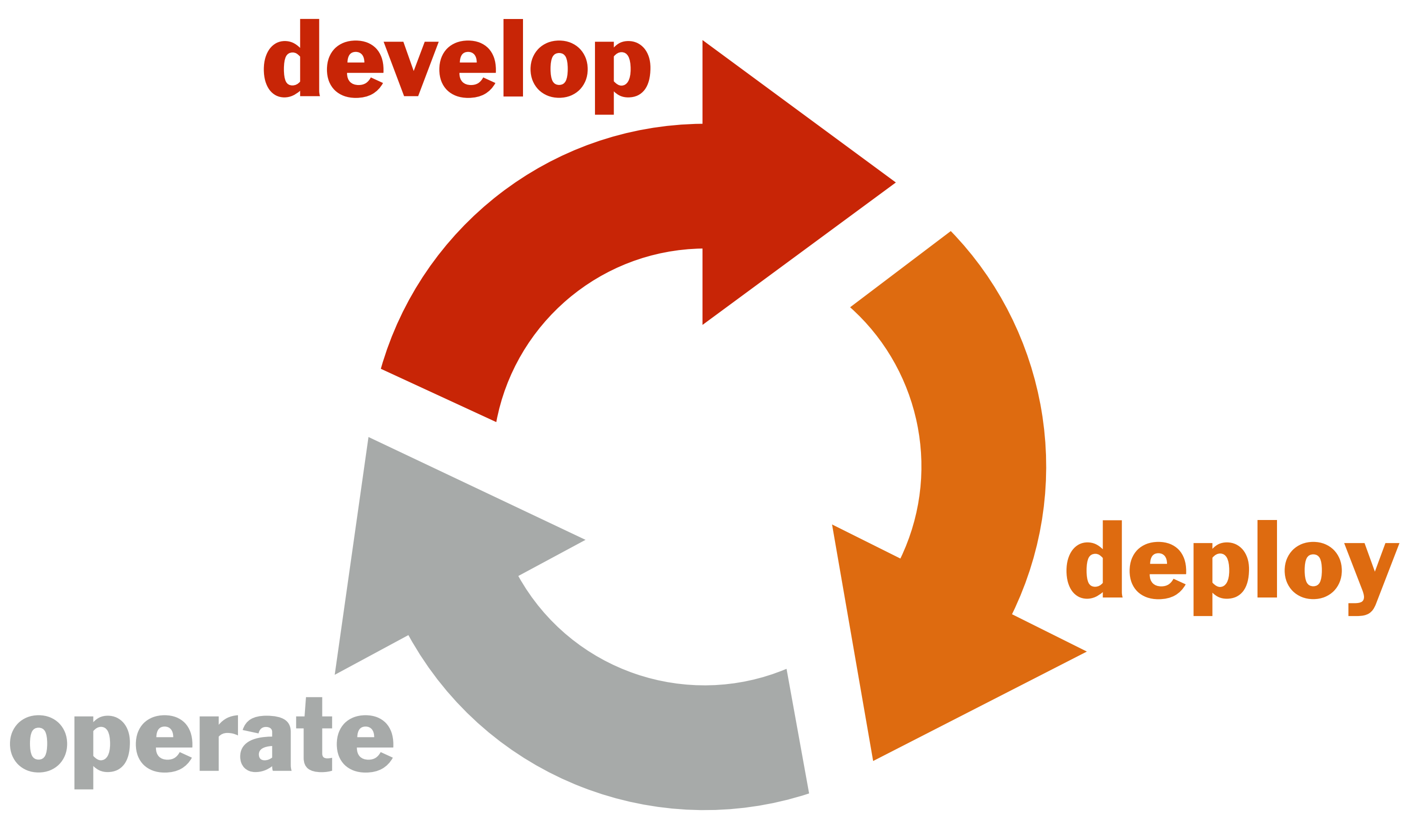
asynchronous



interactions

learning objectives



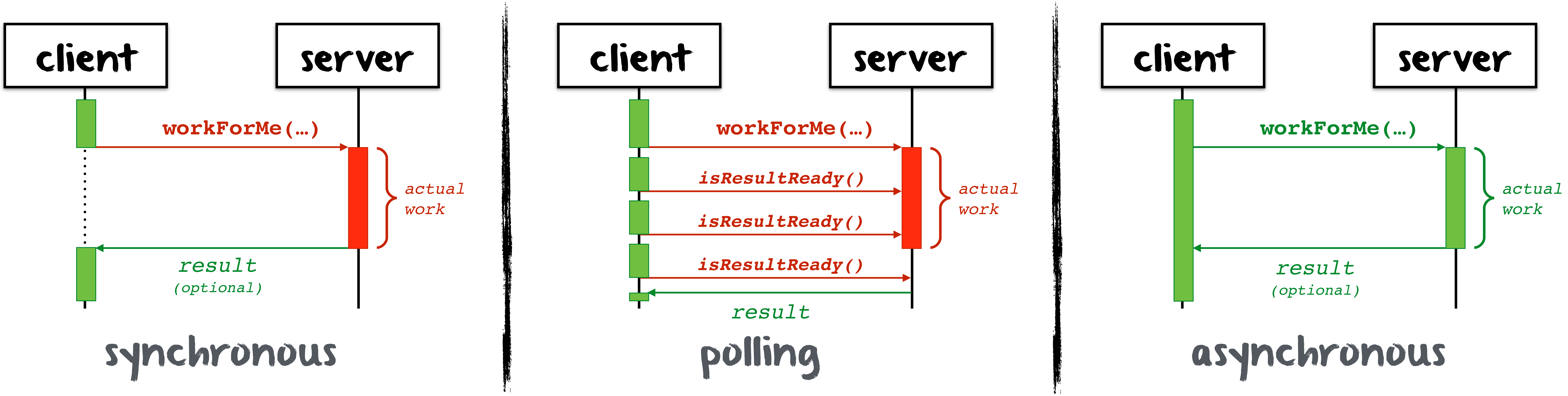
- learn what asynchronous interactions are
- learn about asynchronous methods in java
- learn about tcp/udp sockets and web sockets
- learn about message-oriented middleware and jms*

*java messaging service

what is an asynchronous interaction?

no blocking of the client until the server is done

no polling by the client when a result is expected from the server



asynchronous interactions allow to achieve time decoupling

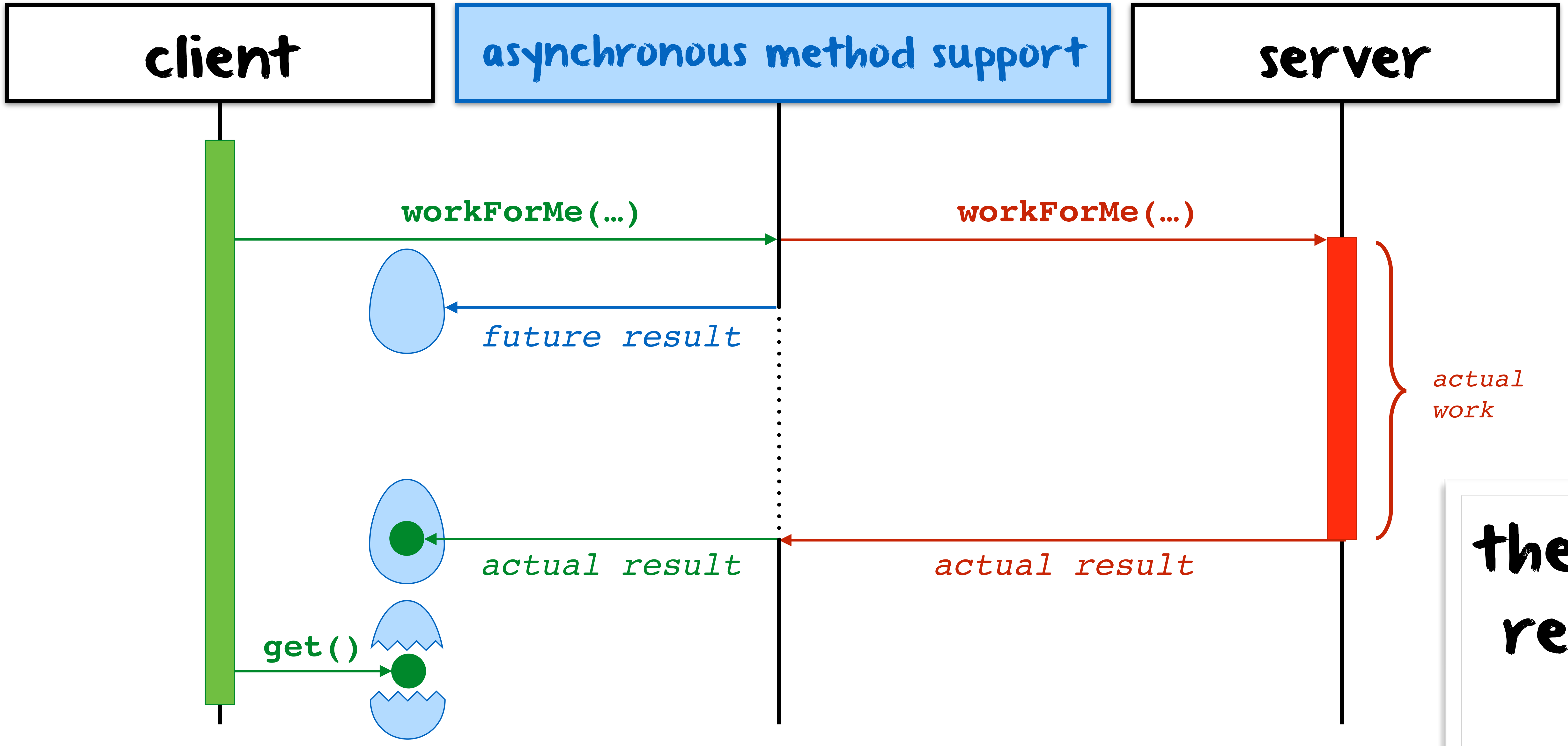
asynchronous methods

a session bean can implement asynchronous methods

they rely on the notion of future objects

these objects are also called promises

the container returns the control to the client before the method is actually invoked in background



the client can try to get the result but might be blocked if it is not ready yet

asynchronous methods

server side

```
@Remote
public interface PortfolioRemote {
 public Future<Double> computeValue();
}
```

an asynchronous method must return **void** or a **Future<V>** object

```
@Stateful
public class Portfolio implements PortfolioRemote {
  @Resource
  SessionContext context;
  :
  @Asynchronous
  public Future<Double> computeValue() {
    double value = ...;  // Processor-intensive computation
    return new AsyncResult<Double>(value);
  }
}
```

if it returns **void**, it cannot declare exceptions

the client can use the **Future<V>** object to retrieve the actual result *or to cancel the invocation*

asynchronous methods

client side

```
@Remote
public interface PortfolioRemote {
 public Future<Double> computeValue();
}
```

```
Future<Double> value = myPortfolio.computeValue();
:
System.out.println("Portfolio is worth $" + value.get());
```

some time passes by...

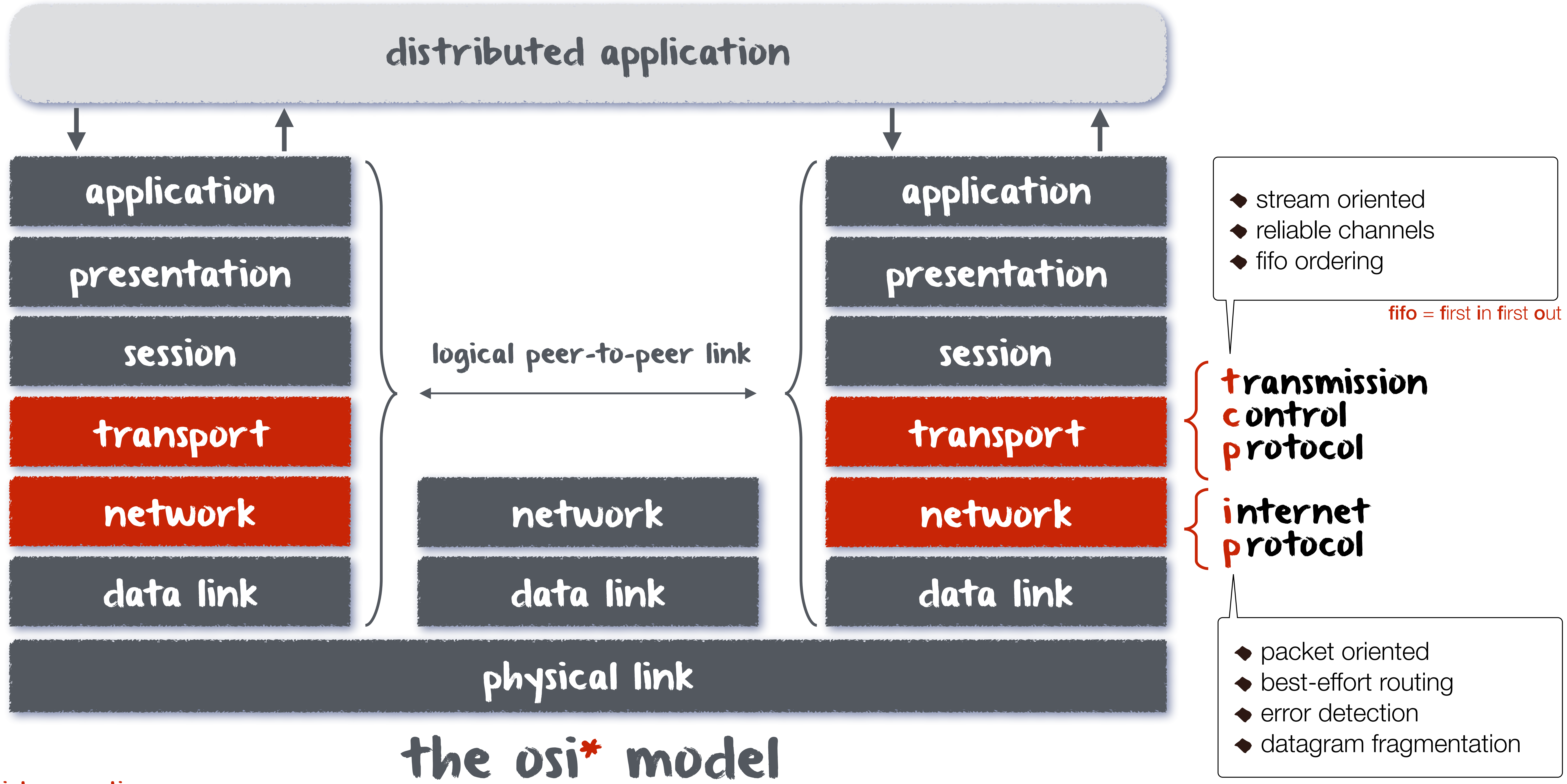
```
Future<Double> value = myPortfolio.computeValue();
try {
  System.out.println("Portfolio is worth $" + value.get(5, TimeUnit.SECONDS));
} catch (TimeoutException ex) {
  value.cancel(true);
  System.err.println("Timeout: operation was cancelled");
}
```

server side

```
@Asynchronous
public Future<Double> computeValue() {
  if (context.wasCancelCalled()) {
    System.err.println("Call to computeValue() was cancelled");
    return null;
  }
  double value = ...;  // Processor-intensive computation
  return new AsyncResult<Double>(value);
}
```

asynchronous messaging

using sockets



the osi* model

*open systems interconnection

asynchronous messaging
using sockets

internet protocol

an **ip address** is used by the ip protocol
to address computers and routers

an ip **v4** address consists of 32-bits (4 bytes) and is often written in **dotted decimal format**, e.g., 130.223.171.8

Class	First byte	Networks			Hosts		Address format	
A	1→126	$2^7 - 2$	=	126	$2^{24} - 2$ =	16'777'214	net id	host id
B	128→191	2^{14}	=	16'384	$2^{16} - 2$ =	65'534	net id	host id
C	192→223	2^{21}	=	2'097'152	$2^8 - 2$ =	254	net id	host id
D	224→239						multicast	
E	240→247						reserved	

asynchronous messaging

using sockets

internet protocol

ip v4 address

Class	Format
A	0NNNNNNN.HHHHHHHH.HHHHHHHH.HHHHHHHH
B	10NNNNNN.NNNNNNNN.HHHHHHHH.HHHHHHHH
C	110NNNNN.NNNNNNNN.NNNNNNNN.HHHHHHHH
D	1110MMMM.MMMMMMMM.MMMMMMMM.MMMMMMMM
E	1111RRRR.RRRRRRRR.RRRRRRRR.RRRRRRRR

N network ID bits

H host ID bits

M multicast address bit

R reserved bits

asynchronous messaging

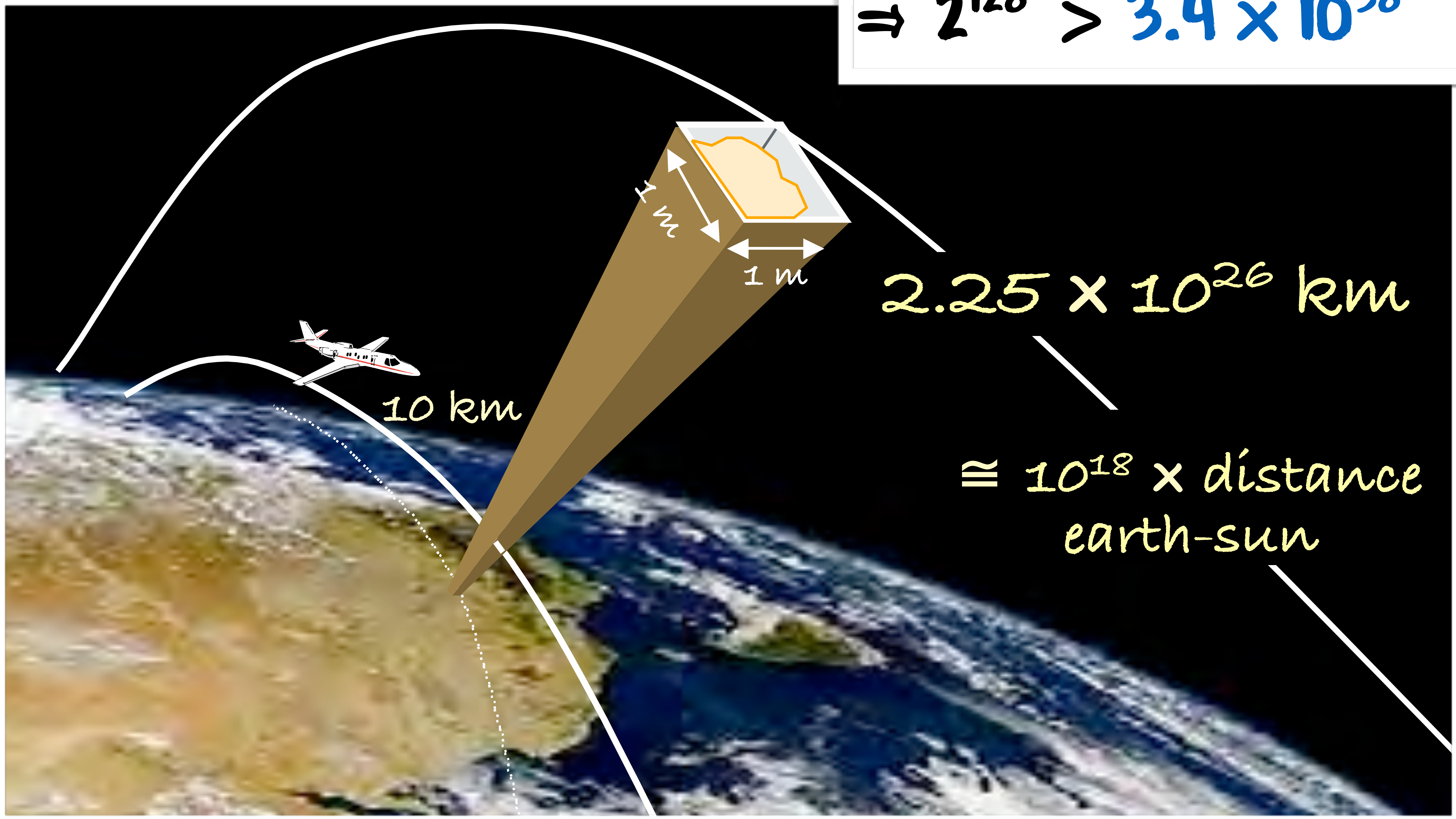
using sockets

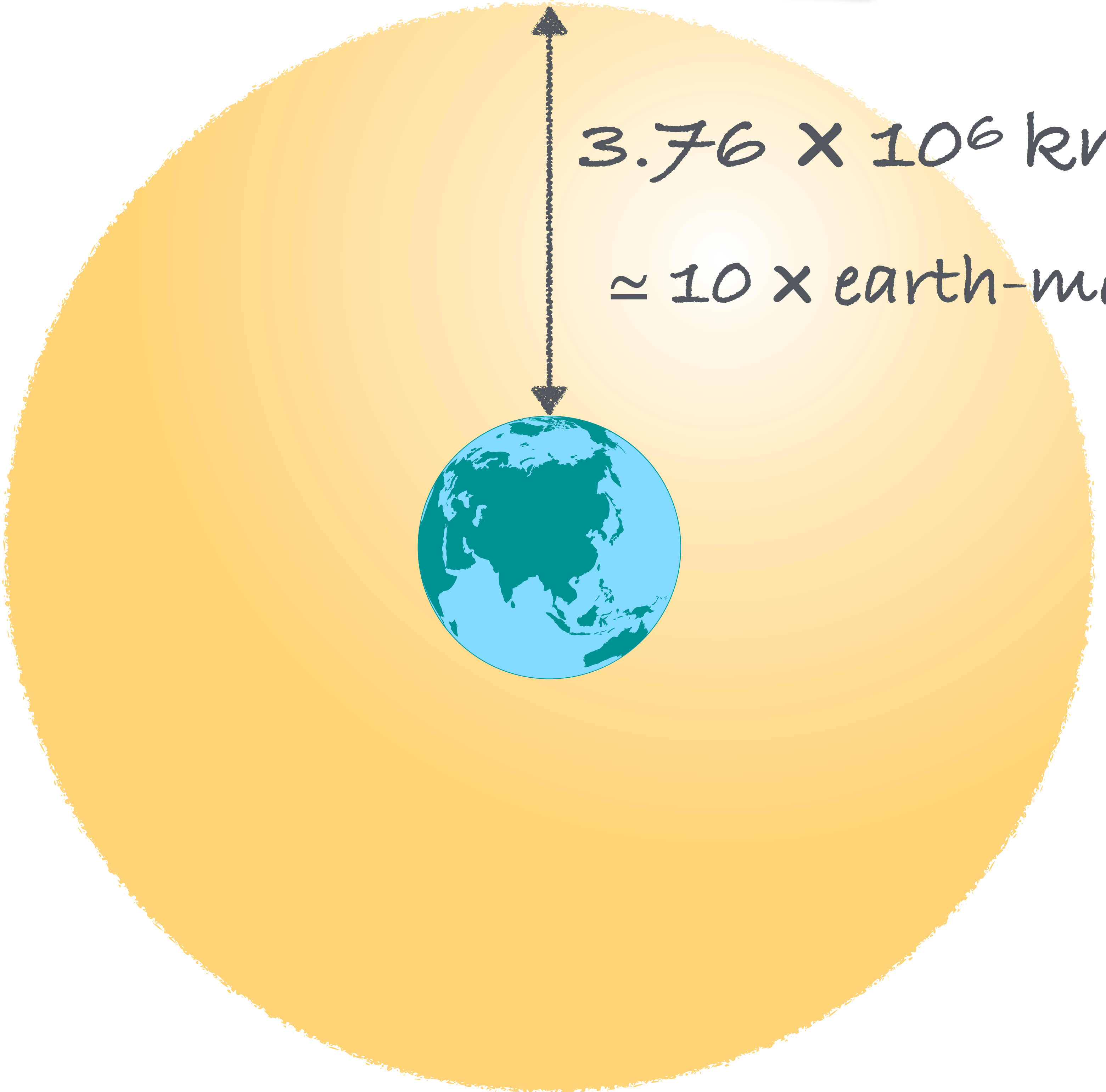
internet protocol

ip v6 address

addresses encoded on 128 bits

$\Rightarrow 2^{128} > 3.4 \times 10^{38}$ available addresses





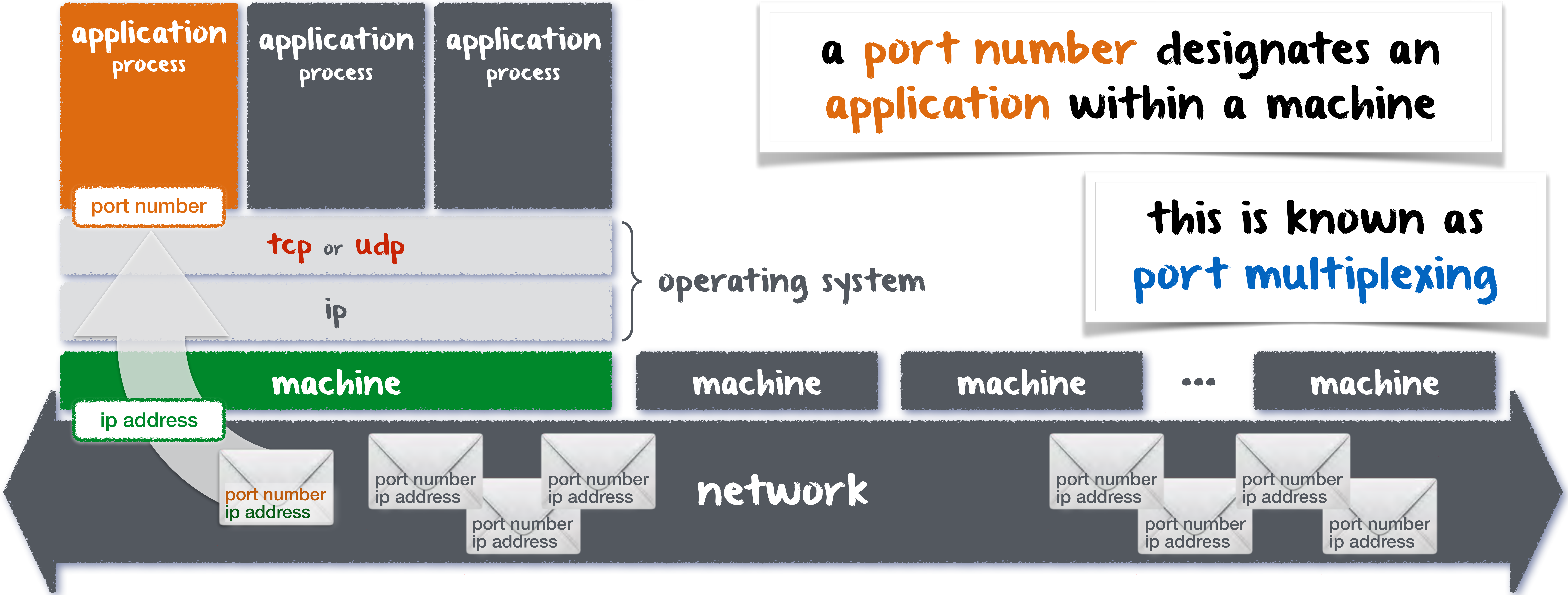
asynchronous messaging
using sockets

addressing applications

an ip address designates a machine

a port number designates an application within a machine

this is known as port multiplexing



asynchronous messaging
using sockets

sockets are programming abstractions representing bidirectional communication channels between processes

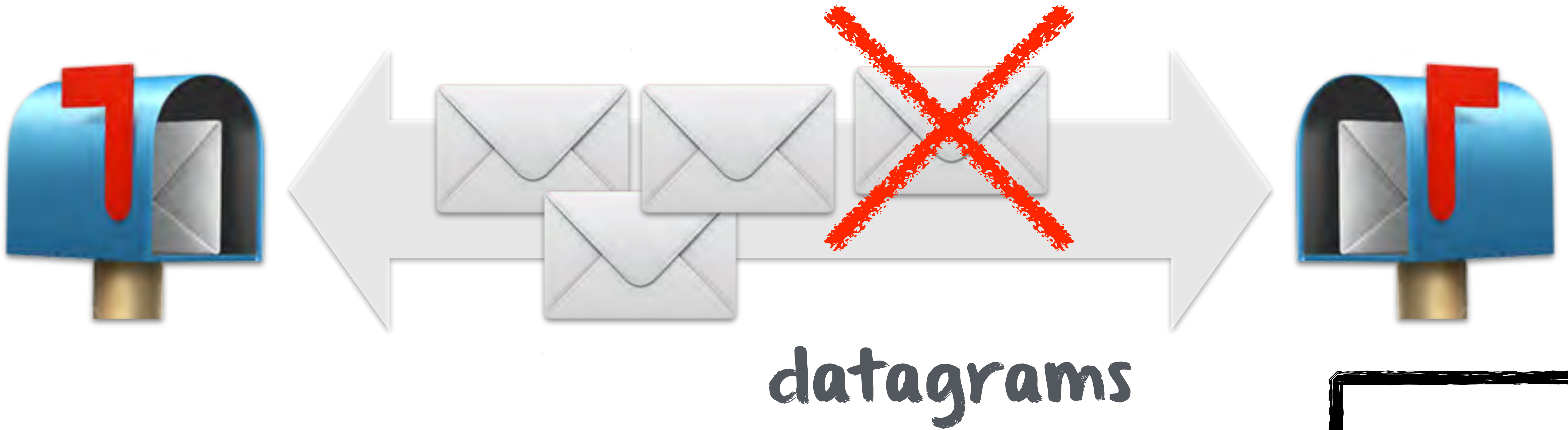
there exists two types of sockets, namely tcp sockets and upd sockets

in java, sockets are instances of various classes found in the `java.net` package

asynchronous messaging

using sockets

transmission
control
protocol



user
datagram
protocol

tcp and udp exhibit dual features

	connection oriented	reliable channels	fifo ordering	message oriented
TCP	YES	YES	YES	NO
UDP	NO	NO	NO	YES

asynchronous messaging
using sockets

tcp sockets

since tcp is connection-oriented, we have two classes for tcp sockets in java

client

```java
public class Socket {
    ⋮
    public Socket(String host, int port) {…}
    public OutputStream getOutputStream() {…}
    public InputStream getInputStream() {{…}
    public void close() {…}
    ⋮
}
```

server

```java
public class ServerSocket {
    ⋮
    public ServerSocket(int port) {…}
    public Socket accept() {…}
    ⋮
}
```

this captures the asymmetry when establishing a communication channel

asynchronous messaging

using sockets

tcp

sockets

server

```java
public class DictionaryServer {
    private static Map dico = Map.of("inheritance", "héritage", "distributed", "réparti");

    public static void main(String[] args) {
        ServerSocket connectionServer = null;
        Socket clientSession = null;
        PrintWriter out = null;
        BufferedReader in = null;
        try {
            connectionServer = new ServerSocket(4444);
            clientSession = connectionServer.accept();
            out = new PrintWriter(clientSession.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(clientSession.getInputStream()));
            String word, mot;

            while ((word = in.readLine()) != null) {
                mot = (String) dico.get(word);
                if (mot == null) {
                    mot = "sorry, no translation available for \"" + word + "\" !";
                }
                out.println(mot);
            }
            out.close();  in.close();  connectionServer.close();  clientSession.close();
        } catch (IOException e) {
            System.out.println(e);
            System.exit(1);
        }
    }
}
```

asynchronous messaging
using sockets

tcp
sockets
client

```java
public class DictionaryClient {
    public static void main(String[] args) {
        Socket mySession = null;
        PrintWriter out = null;
        BufferedReader in = null;
        BufferedReader stdIn = null;
        try {
            if (args.length < 1) {
                System.out.println("Hostname missing.");
                System.exit(1);
            }
            mySession = new Socket(args[0], 4444);
            out = new PrintWriter(mySession.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(mySession.getInputStream()));
            stdIn = new BufferedReader(new InputStreamReader(System.in));
            String fromServer, fromUser;

            System.out.println("Go on, ask the dictionary server!");
            while (!(fromUser = stdIn.readLine()).equals("quit")) {
                out.println(fromUser);
                fromServer = in.readLine();
                System.out.println("-> " + fromServer);
            }
            out.close();  in.close();  stdIn.close();  mySession.close();
        } catch (UnknownHostException e) {
            System.err.println("Host Unknown: " + args[0]);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("No connection to: " + args[0]);
            System.exit(1);
        }
    }
}
```

asynchronous messaging
using sockets

the concept of streams
(unix and java)

streams offer a unified programming abstraction for reading and writing data

streams can encapsulate various types of data sources, e.g., files, byte arrays in memory, sockets, etc.

streams can encapsulate other streams to stack up processing of the data

in java, streams are instances of various classes found in the `java.io` package

asynchronous messaging

using sockets

the concept of streams

(unix and java)

```
Socket clientSession= connectionServer.accept();
BufferedReader in= new BufferedReader(new InputStreamReader(clientSession.getInputStream()));
```

data source

byte stream

character stream

buffered character stream

printer and writer classes are **special streams** manipulating **only characters**

standard operating systems-level input and output streams are also accessed via java streams (`System.in` & `System.out`)

asynchronous messaging

using sockets

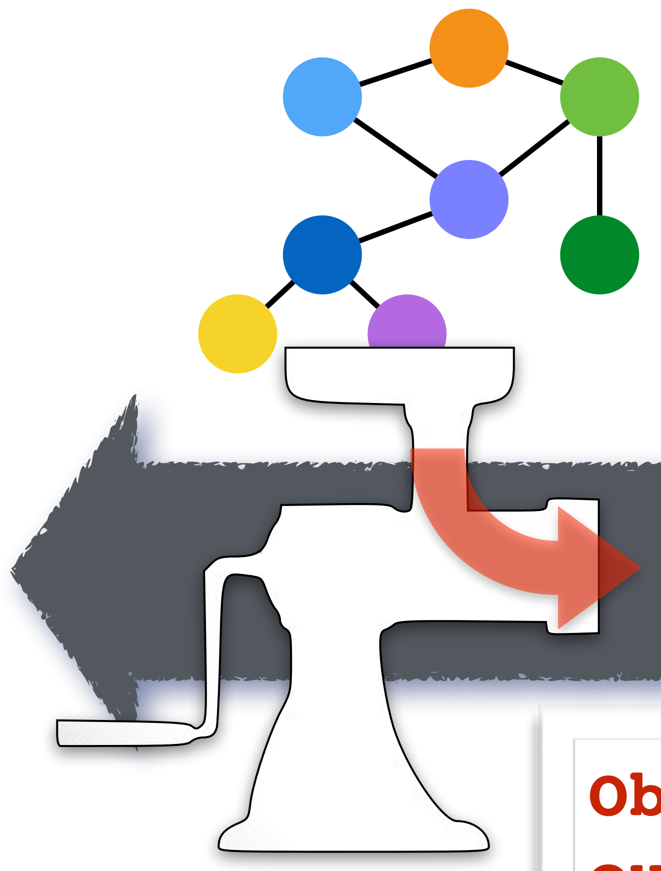
the concept of object streams

fact — the network knows nothing about objects, only bytes

problem — so how can we send an object graph across the network?

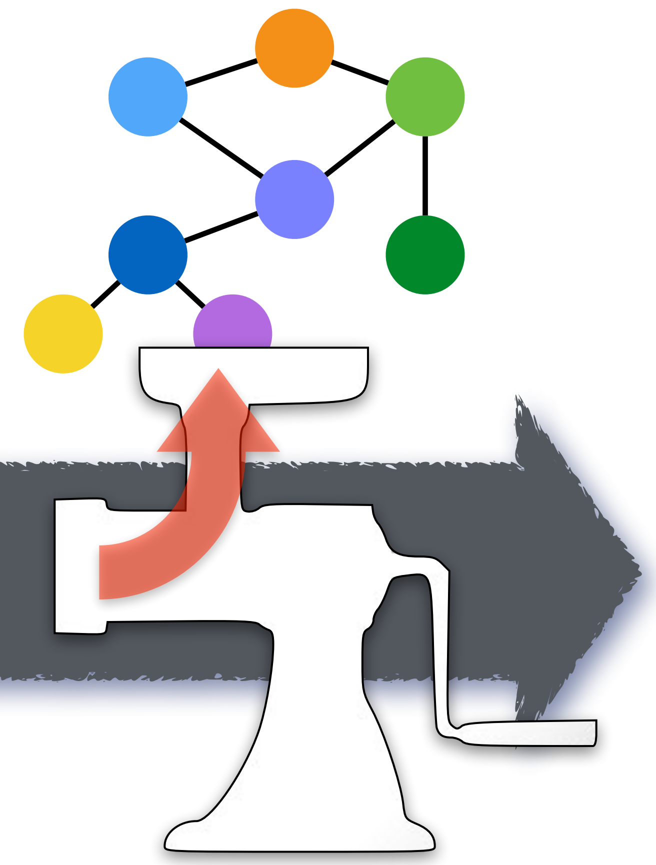
solution — any java object can be encoded into a stream of bytes and recreated from it by implementing the `java.io.Serializable` interface

this process is known as **object serialization**



sender

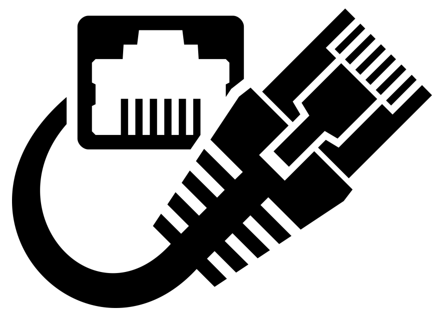
1011011101011001010101010110010111001001101



receiver

```
ObjectOutputStream out = new ObjectOutputStream(sessionWithServer.getOutputStream());
out.writeObject(senderCollection);
```

```
ObjectInputStream in = new ObjectInputStream(sessionWithClient.getInputStream());
Collection receiverCollection = (Collection) in.readObject();
```



asynchronous messaging

using sockets

udp sockets

since udp is **connectionless**, we have only one class for udp sockets in java

```java
public class DatagramSocket {
    ⋮
    public DatagramSocket() {…}
    public DatagramSocket(int port) {…}
    public void send(DatagramPacket packet) {…}
    public void receive(DatagramPacket packet) {…}
    public void close() {…}
    ⋮
}
```

```java
public class DatagramPacket {
    ⋮
    public DatagramPacket(…) {…}
    public byte[] getData() {…}
    public InetAddress getAddress() {…}
    ⋮
}
```

the `DatagramPacket` class is key when working with udp sockets

it captures the **message-oriented** nature of udp sockets

asynchronous messaging
using sockets

udp sockets

```
public class QuoteServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = null;
        BufferedReader in = null;
        socket = new DatagramSocket(4445);
        in = new BufferedReader(new FileReader("one-liners.txt"));
        String quote = null;
        boolean moreQuotes = true;

        while (moreQuotes) {
            byte[] buf = new byte[256];
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);
            quote = in.readLine();
            if (quote == null) {
                moreQuotes = false;
                buf = ("No more, bye!").getBytes();
            } else { buf = quote.getBytes(); }
            InetAddress address = packet.getAddress();
            int port = packet.getPort();
            packet = new DatagramPacket(buf, buf.length, address, port);
            socket.send(packet);
        }
        socket.close();
    }
}
```

Life is wonderful. Without it we'd all be dead.
Daddy, why doesn't this magnet pick up this floppy disk?
Give me ambiguity or give me something else.
I.R.S.: We've got what it takes to take what you've got!
We are born naked, wet and hungry. Then things get worse.
Make it idiot proof and someone will make a better idiot.
He who laughs last thinks slowest!
Always remember you're unique, just like everyone else.
"More hay, Trigger?" "No thanks, Roy, I'm stuffed!"
A flashlight is a case for holding dead batteries.
Lottery: A tax on people who are bad at math.
Error, no keyboard - press F1 to continue.
There's too much blood in my caffeine system.
Artificial Intelligence usually beats real stupidity.
Hard work has a future payoff. Laziness pays off now.
"Very funny, Scotty. Now beam down my clothes."
Puritanism: The haunting fear that someone, somewhere may be happy.
Consciousness: that annoying time between naps.
Don't take life too seriously, you won't get out alive.
I don't suffer from insanity. I enjoy every minute of it.
Better to understand a little than to misunderstand a lot.

asynchronous messaging

using sockets

udp sockets

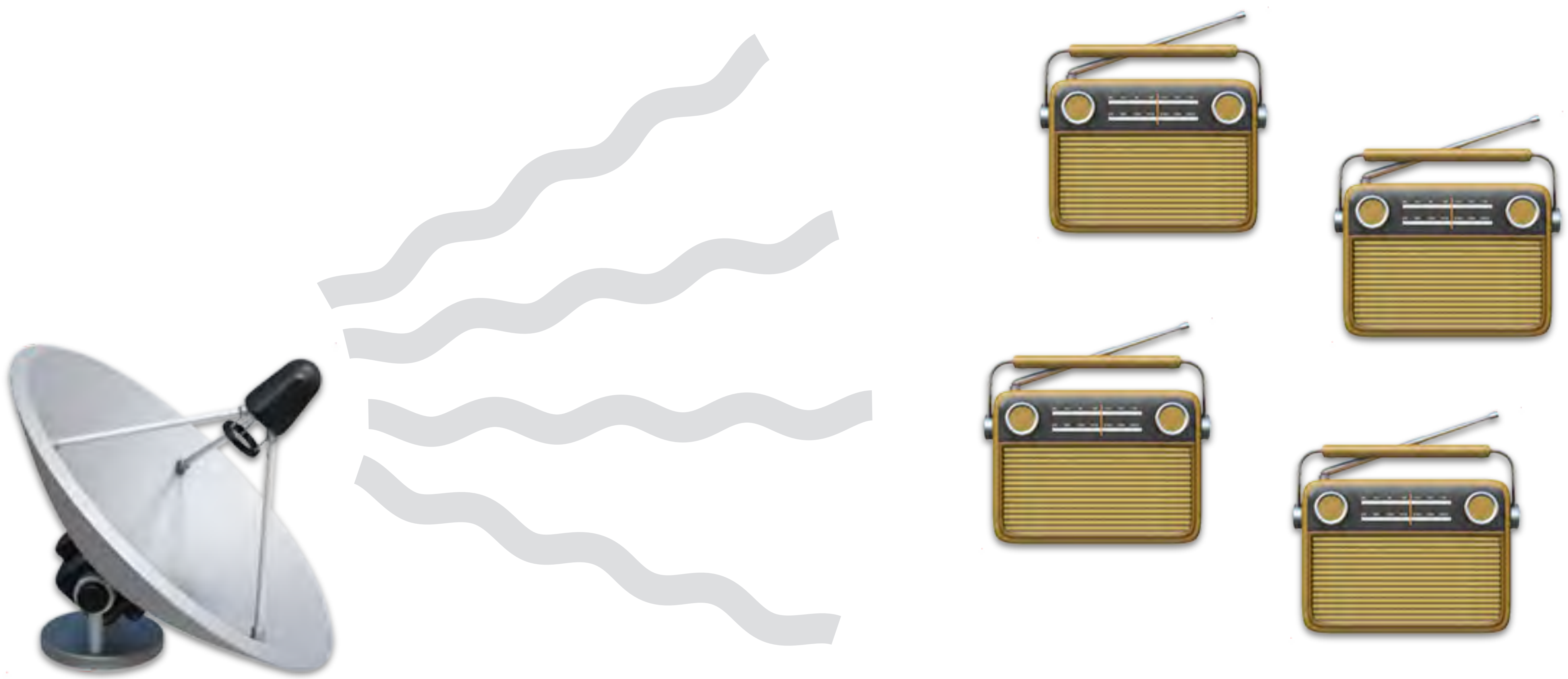
```java
public class QuoteClient {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) { System.out.println("Missing hostname"); System.exit(1); }
        DatagramSocket socket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Go on, ask for a quote by typing return!");
        while (!stdIn.readLine().equals("quit") ) {
            byte[] buf = new byte[256];
            DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 4445);
            socket.send(packet);
            packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);
            String received = new String(packet.getData()).trim();
            System.out.println("-> " + received);
        }
        socket.close();
    }
}
```

asynchronous messaging
using sockets

user
datagram
protocol

one-to-one communication

one-to-many communication



udp multicast

a multicast address lies in range [224.0.0.0 , 239.255.255.255] and defines a multicast group

in java, udp multicast is accessed via `MulticastSocket`, a subclass of `DatagramSocket`

asynchronous messaging
using sockets

udp multicast

```
public class MulticastQuoteSender {
   public static void main(String[] args) throws Exception {
        MulticastSocket socket = null;
        BufferedReader in = null;
        socket = new MulticastSocket();
        InetSocketAddress group = new InetSocketAddress(InetAddress.getByName("228.0.0.4"), 9000);
        NetworkInterface networkInterface = NetworkInterface.getByName("en0");
        socket.setTimeToLive(1);
        in = new BufferedReader(new FileReader("one-liners.txt"));
        String quote = null;
        boolean moreQuotes = true;

        while (moreQuotes) {
            Thread.currentThread().sleep(2000);
            byte[] buf = new byte[256];
            quote = in.readLine();
            if (quote == null) {
                moreQuotes = false;
                buf = ("No more, bye!").getBytes();
            } else {
                buf = quote.getBytes();
            }
            System.out.println("About to multicast: " + new String(buf));
            DatagramPacket packet = new DatagramPacket(buf, buf.length, group);
            socket.send(packet);
        }
        socket.close();
    }
}
```

maximum number of routers a multicast packet can go through before being deleted

asynchronous messaging

using sockets

udp multicast

```java
public class MulticastQuoteReceiver {

    public static void main(String[] args) throws Exception {
        try (MulticastSocket socket = new MulticastSocket(9000)) {
            InetSocketAddress group = new InetSocketAddress(InetAddress.getByName("228.0.0.4"), 9000);
            NetworkInterface netInterface = NetworkInterface.getByName("en0");
            socket.joinGroup(group, netInterface);
            while (true) {
                byte[] buf = new byte[256];
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                System.out.print("Waiting for the next quote: ");
                socket.receive(packet);
                String received = new String(packet.getData());
                System.out.println(received.trim());
                if (received.contains("bye")) {
                    break;
                }
            }
            socket.leaveGroup(group, netInterface);
            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

tuning in

tuning out

asynchronous messaging

using sockets

udp multicast

creating a multicast route

```
garbi — -bash — ttys004
wallace-palace:~ garbi$ sudo route -nv add -net 228.0.0.4 -interface en0
u: inet 228.0.0.4; u: link en0:f0.18.98.74.5f.ff; u: inet 255.255.255.255; RTM_ADD: Add Route: len 136, pid: 0, seq 1, errno 0, flags:<UP,STATIC>
locks:  inits:
sockaddrs: <DST,GATEWAY,NETMASK>
 228.0.0.4 en0:f0.18.98.74.5f.ff 255.255.255.255
add net 228.0.0.4: gateway en0
wallace-palace:~ garbi$
```

deleting the multicast route

```
garbi — -bash — ttys004
wallace-palace:~ garbi$ sudo route -v delete -inet 228.0.0.4
u: inet 228.0.0.4; RTM_DELETE: Delete Route: len 108, pid: 0, seq 1, errno 0, flags:<UP,GATEWAY,HOST,STATIC>
locks:  inits:
sockaddrs: <DST>
 228.0.0.4
delete host 228.0.0.4
wallace-palace:~ garbi$
```

asynchronous messaging

using web sockets

unlike http, which is a request-response protocol, the web socket protocol is message-oriented and offers full-duplex channels

web sockets are similar to tcp sockets but they offer streams of messages, rather than streams of bytes

most web browsers and servers support web sockets via two uri* schemes

`ws://host:port/...` for unencrypted streams

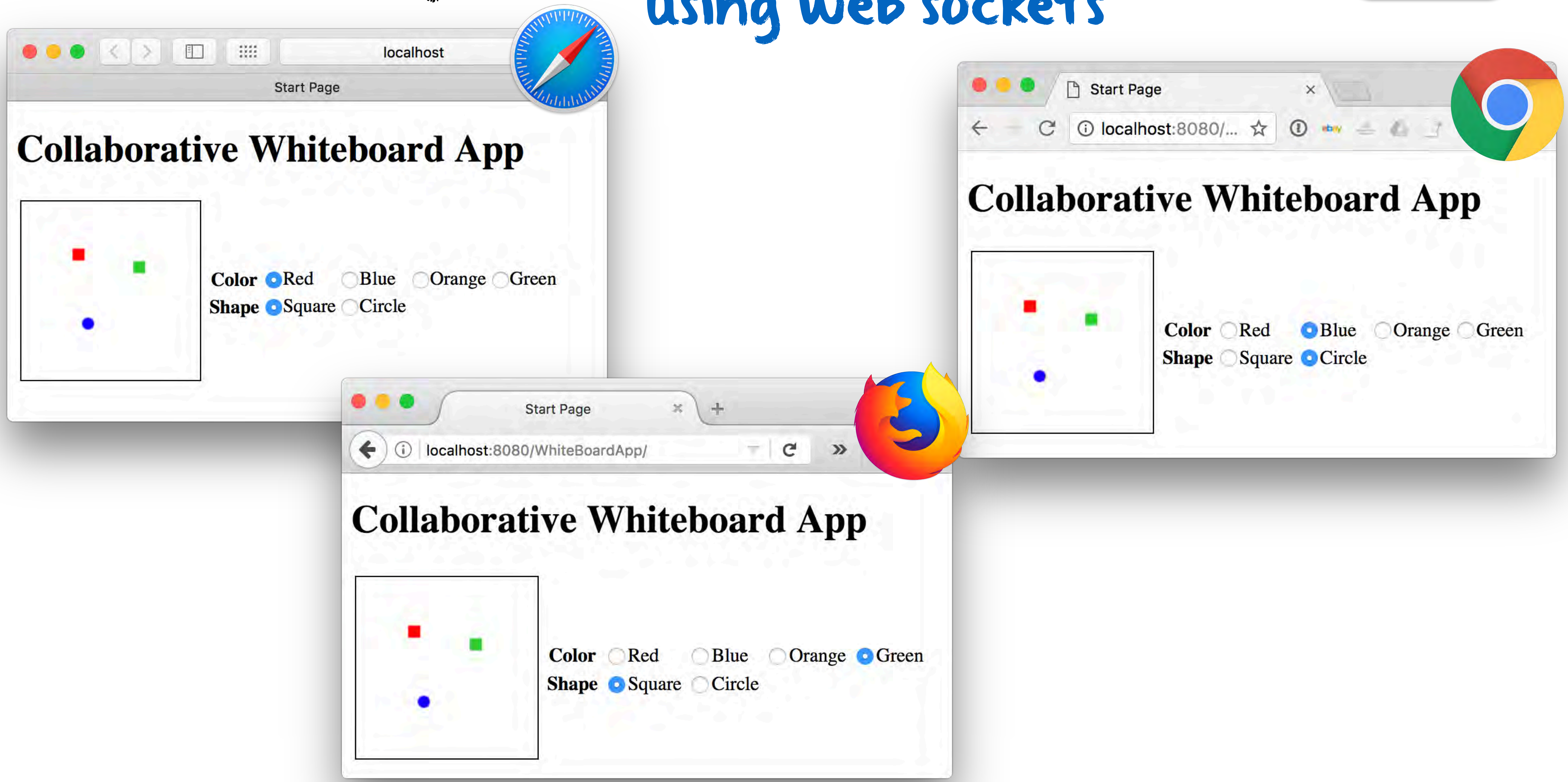
`wss://host:port/...` for encrypted streams

the web socket protocol is based on tcp and totally independent from http, except for the handshake phase, which done via an http request interpreted by the server as an upgrade request

*uniform resource identifier

asynchronous messaging

using web sockets



asynchronous messaging

using web sockets

web client

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Start Page</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1>Collaborative Whiteboard App</h1>

    <table>
      <tr>
        <td>
          <canvas id="myCanvas" width="150" height="150" style="border:1px solid #000000;"></canvas>
        </td>
        <td>
          <form name="inputForm">
            <table>
              <tr>
                <th>Color</th>
                <td><input type="radio" name="color" value="#FF0000" checked="true">Red</td>
                <td><input type="radio" name="color" value="#0000FF">Blue</td>
                <td><input type="radio" name="color" value="#FF9900">Orange</td>
                <td><input type="radio" name="color" value="#33CC33">Green</td>
              </tr>
              <tr>
                <th>Shape</th>
                <td><input type="radio" name="shape" value="square" checked="true">Square</td>
                <td><input type="radio" name="shape" value="circle">Circle</td>
                <td> </td>
                <td> </td>
            </table>
      </tr>
    </table>
    <script type="text/javascript" src="websocket.js"></script>
    <script type="text/javascript" src="whiteboard.js"></script>
  </body>
</html>
```

client code in javascript

index.html

asynchronous messaging

using web sockets

web client

web socket
in javascript

```
var wsUri = "ws://" + document.location.host + document.location.pathname + "whiteboardendpoint";
console.log("wsURI: " + wsUri)
var websocket = new WebSocket(wsUri);

function sendText(json) {
  console.log("sending text: " + json);
  websocket.send(json);
}

websocket.onerror = function (evt) {
  onError(evt)
};

websocket.onmessage = function (evt) {
  onMessage(evt)
};

function onError(evt) {
  writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
}

function onMessage(evt) {
  console.log("received: " + evt.data);
  drawImageText(evt.data);
}
```

websocket.js

asynchronous messaging

using web sockets

web client

❶

```javascript
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
canvas.addEventListener("click", defineImage, false);

function getCurrentPos(evt) {
  var rect = canvas.getBoundingClientRect();
  return {
    x: evt.clientX - rect.left,
    y: evt.clientY - rect.top
  };
}
```

❷

```javascript
function defineImage(evt) {
  var currentPos = getCurrentPos(evt);

  for (i = 0; i < document.inputForm.color.length; i++) {
    if (document.inputForm.color[i].checked) {
      var color = document.inputForm.color[i];
      break;
    }
  }

  for (i = 0; i < document.inputForm.shape.length; i++) {
    if (document.inputForm.shape[i].checked) {
      var shape = document.inputForm.shape[i];
      break;
    }
  }

  var json = JSON.stringify({
    "shape": shape.value,
    "color": color.value,
    "coords": {
      "x": currentPos.x,
      "y": currentPos.y
    }
  });
  drawImageText(json);
  sendText(json);
}
```

❸

```javascript
function drawImageText(image) {
  console.log("drawImageText");
  var json = JSON.parse(image);
  context.fillStyle = json.color;
  switch (json.shape) {
    case "circle":
      context.beginPath();
      context.arc(json.coords.x, json.coords.y, 5, 0, 2 * Math.PI, false);
      context.fill();
      break;
    case "square":
    default:
      context.fillRect(json.coords.x, json.coords.y, 10, 10);
      break;
  }
}
```

whiteboard.js

web server

asynchronous messaging

using web sockets

```java
@ServerEndpoint(value = "/whiteboardendpoint", encoders = {FigureEncoder.class}, decoders = {FigureDecoder.class})
public class MyWhiteboard {

  private static Set<Session> peers = Collections.synchronizedSet(new HashSet<Session>());

  @OnMessage
  public void broadcastFigure(Figure figure, Session session) throws IOException, EncodeException {
    System.out.println("broadcastFigure: " + figure);
    for (Session peer : peers) {
      if (!peer.equals(session)) {
        peer.getBasicRemote().sendObject(figure);
      }
    }
  }

  @OnOpen
  public void onOpen(Session peer) {
    peers.add(peer);
  }

  @OnClose
  public void onClose(Session peer) {
    peers.remove(peer);
  }
}
```

```java
public class Figure {
  private JsonObject json;

  public Figure(JsonObject json) {
    this.json = json;
  }
 public JsonObject getJson() {
    return json;
  }
 public void setJson(JsonObject json) {
    this.json = json;
  }

  @Override
  public String toString() {
    StringWriter writer = new StringWriter();
    Json.createWriter(writer).write(json);
    return writer.toString();
  }
}
```

asynchronous messaging

using web sockets

web server

```java
public class FigureDecoder implements Decoder.Text<Figure> {

  @Override
  public Figure decode(String s) throws DecodeException {
    JsonObject jsonObject = Json.createReader(new StringReader(s)).readObject();
    return new Figure(jsonObject);
  }

  @Override
  public boolean willDecode(String s) {
    try {
      Json.createReader(new StringReader(s)).readObject();
      return true;
    } catch (JsonException ex) {
      ex.printStackTrace();
      return false;
    }
  }

  @Override
  public void init(EndpointConfig config) {
    System.out.println("init");
  }

  @Override
  public void destroy() {
    System.out.println("destroy");
  }
}
```

```java
public class FigureEncoder implements Encoder.Text<Figure> {

  @Override
  public String encode(Figure figure) throws EncodeException {
    return figure.getJson().toString();
  }

  @Override
  public void init(EndpointConfig config) {
    System.out.println("init");
  }

  @Override
  public void destroy() {
    System.out.println("destroy");
  }
}
```

asynchronous messaging

using message-oriented middleware

in addition to time decoupling, message-oriented middleware also achieve space decoupling

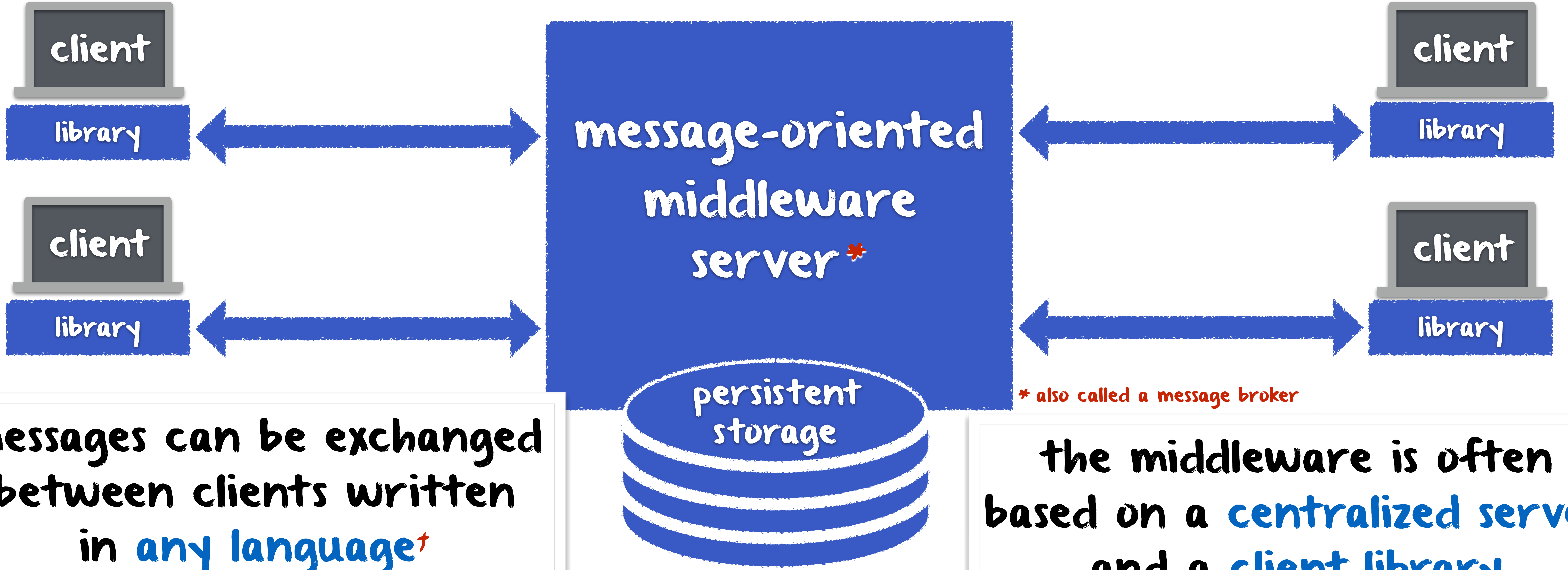
time decoupling ⇒ asynchrony

space decoupling ⇒ anonymity

a message-oriented middleware is a software layer acting as a kind of "middle man" between distributed clients (of the middleware)

asynchronous messaging

using message-oriented middleware



* also called a message broker

messages can be exchanged between clients written in **any language**†

† assuming a library exists for that language

the middleware is often based on a **centralized server** and a **client library**

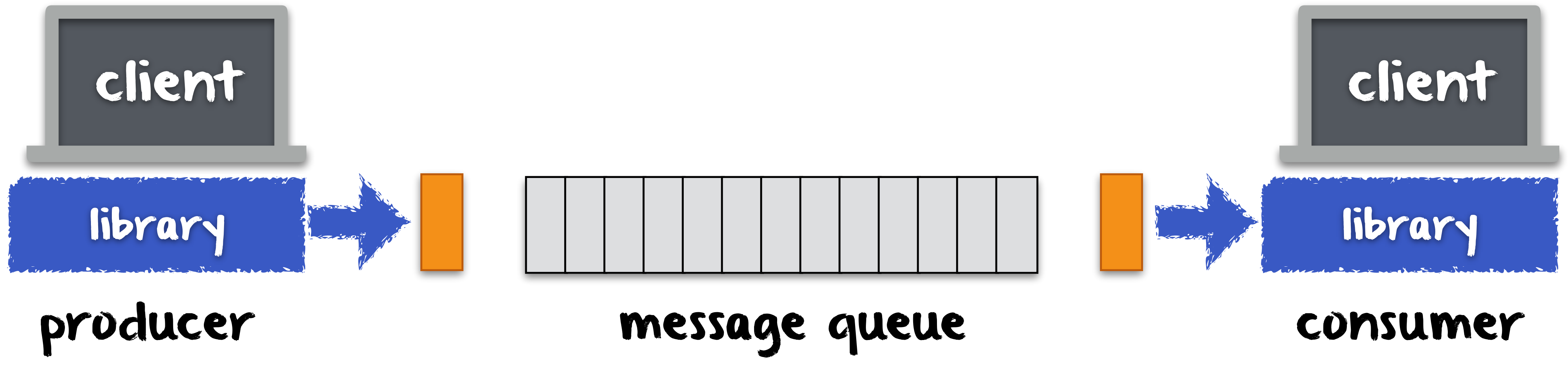
many software providers offer such middleware products, e.g., IBM, Oracle, Apache

asynchronous messaging

using message-oriented middleware

point-to-point model

one-to-one communication where producers send messages and each message is consumed by one and only one consumer



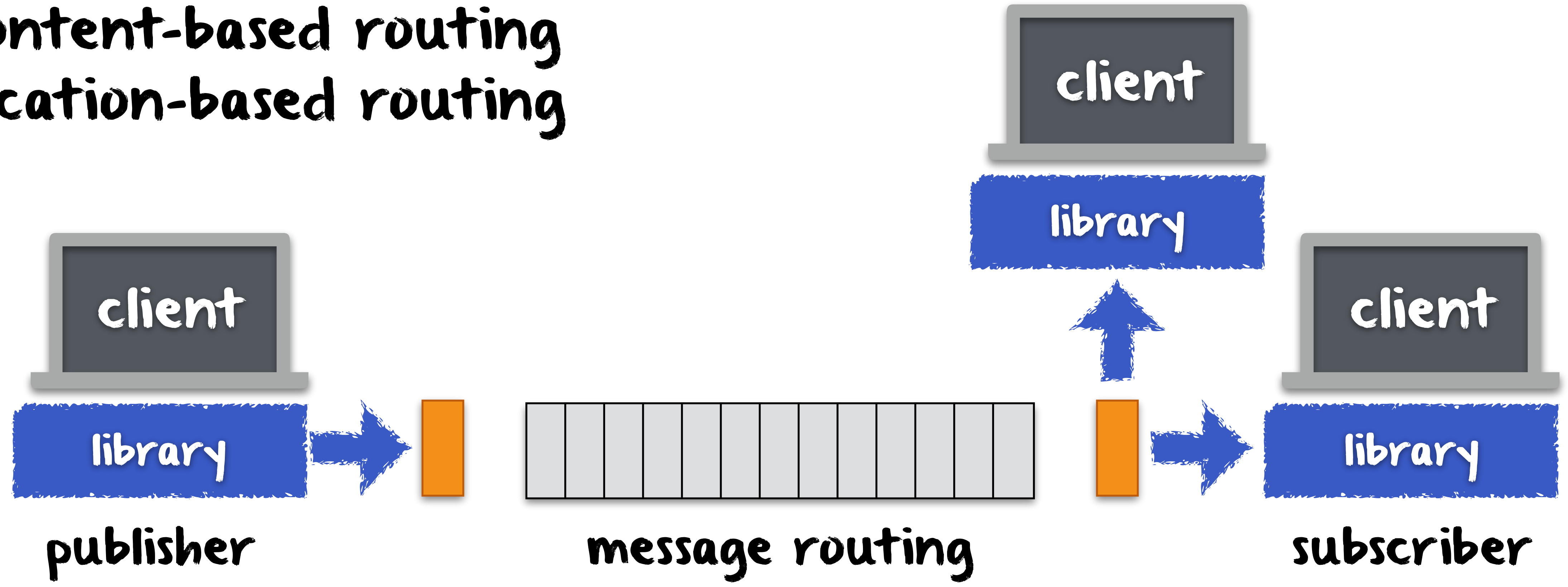
- messages are kept in a queue until consumed
- this model can be used for load-balancing but then the fifo* ordering of is no longer guaranteed

*fifo order = first in first out

publish/subscribe model

one-to-many communication where producers publish messages and all consumers that have subscribed receive them

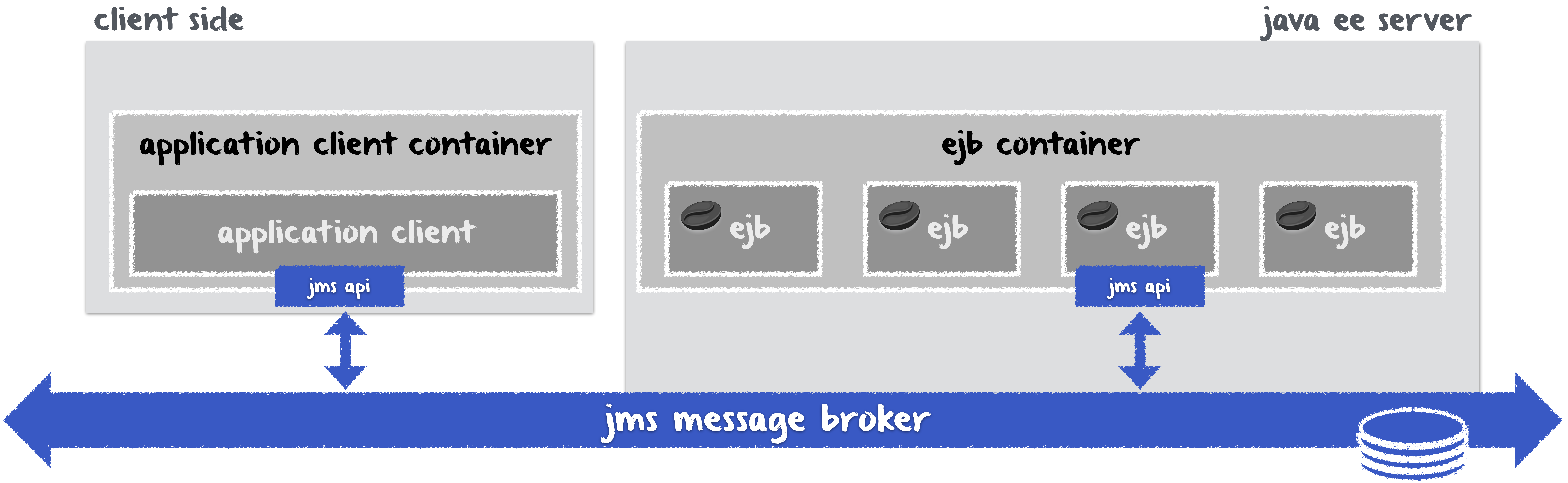
- there exists various message routing criteria
  - topic-based routing
  - content-based routing
  - location-based routing
  - ...



asynchronous messaging

using message-oriented middleware

the java messaging service (jms) defines the standard asynchronous messaging api* of the java ee platform



asynchronous messaging

using message-oriented middleware

the jms api is essentially an **interface-based specification** that is encapsulating **existing implementations** from software providers

when using jms, we can distinguish **three key phases**

1. **development** phase
2. **deployment** phase
3. **execution** phase

Package javax.jms

The Java Message Service (JMS) API provides a common way for Java programs to create, send, receive and read an enterprise messaging system's messages.

See: Description

Interface Summary

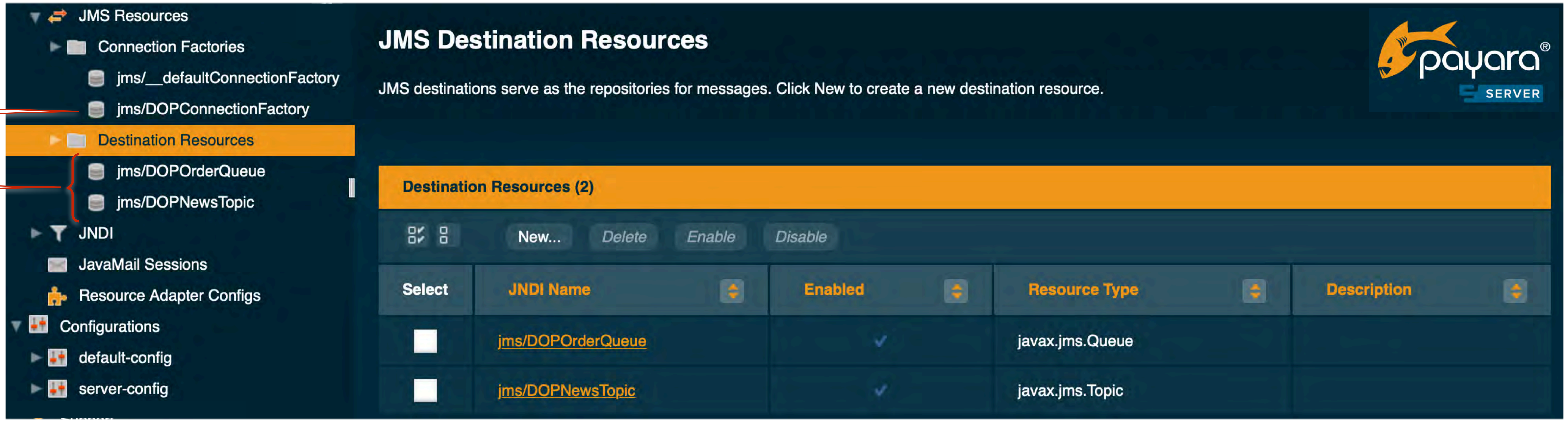
Interface	Description
BytesMessage	A `BytesMessage` object is used to send a message containing a stream of uninterpreted bytes.
CompletionListener	A `CompletionListener` is implemented by the application and may be specified when a message is sent asynchronously.
Connection	A `Connection` object is a client's active connection to its JMS provider.
ConnectionConsumer	For application servers, `Connection` objects provide a special facility for creating a `ConnectionConsumer` (optional).
ConnectionFactory	A `ConnectionFactory` object encapsulates a set of connection configuration parameters that has been defined by an administrator.
ConnectionMetaData	A `ConnectionMetaData` object provides information describing the `Connection` object.
DeliveryMode	The delivery modes supported by the JMS API are `PERSISTENT` and `NON_PERSISTENT`.
Destination	A `Destination` object encapsulates a provider-specific address.
ExceptionListener	If a JMS provider detects a serious problem with a `Connection` object, it informs the `Connection` object's `ExceptionListener`, if one has been registered.
JMSConsumer	A client using the simplified JMS API introduced for JMS 2.0 uses a `JMSConsumer` object to receive messages from a queue or topic.
JMSContext	A `JMSContext` is the main interface in the simplified JMS API introduced for JMS 2.0.
JMSProducer	A `JMSProducer` is a simple object used to send messages on behalf of a `JMSContext`.
MapMessage	A `MapMessage` object is used to send a set of name-value pairs.
Message	The `Message` interface is the root interface of all JMS messages.
MessageConsumer	A client uses a `MessageConsumer` object to receive messages from a destination.
MessageListener	A `MessageListener` object is used to receive asynchronously delivered messages.
MessageProducer	A client uses a `MessageProducer` object to send messages to a destination.
ObjectMessage	An `ObjectMessage` object is used to send a message that contains a serializable object in the Java programming language ("Java object").
Queue	A `Queue` object encapsulates a provider-specific queue name.
QueueBrowser	A client uses a `QueueBrowser` object to look at messages on a queue without removing them.
QueueConnection	A `QueueConnection` object is an active connection to a point-to-point JMS provider.
QueueConnectionFactory	A client uses a `QueueConnectionFactory` object to create `QueueConnection` objects with a point-to-point JMS provider.
QueueReceiver	A client uses a `QueueReceiver` object to receive messages that have been delivered to a queue.

asynchronous messaging

using message-oriented middleware

execution

1. a producer creates and send messages via the jms api, specifying a destination
2. a jms-compliant middleware routes those messages to the specified destination
3. a consumer receives messages via the jms api specifying the same destination



deployment

1. start the jms-compliant middleware broker
2. create the destination referenced by the producer and consumer code
3. package the library implementing the jms api on the producer and consumer code

asynchronous messaging

using message-oriented middleware

development

point-to-point model

⇓

destination = **queue**

```
public class OrderProducer {

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

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

    public static void main(String[] args) throws IOException {
        JMSContext context = connectionFactory.createContext();
        JMSProducer producer = context.createProducer();
        boolean moreOrders = true;
        while (moreOrders) {
           String order = askForOrder(3);
            if (order != null) {
                System.out.println("Sending order to " + order);
                producer.send(queue, order);
                if (order.toLowerCase().contains("quit")) {
                    moreOrders = false;
                }
            } else {
                moreOrders = false;
            }
        }
        System.out.println("Bye bye!");
    }
```

asynchronous messaging

using message-oriented middleware

```java
public class OrderConsumer implements MessageListener {

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

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

    private static boolean stopReceiving = false;

    public static void main(String[] args) throws InterruptedException {
       JMSContext context = connectionFactory.createContext();
       JMSConsumer consumer = context.createConsumer(queue);

       System.out.println("I am now ready to receive orders");
       MessageListener listener = new OrderConsumer();
       consumer.setMessageListener(listener);
       for (int i = 0; i < 60; i++) {
            Thread.sleep(1000);
            System.out.print(".");
            if (stopReceiving) {
                break;
            }
        }
        System.out.println("\nBye bye!");
    }
    ⋮
```

non-blocking version

development

point-to-point model

⇓

destination = queue

```java
   ⋮
  @Override
   public void onMessage(Message message) {
        System.out.println();
        String order = "quit";
        try {
            order = ((TextMessage) message).getText();
            System.out.println("I received the order to " + order);
        } catch (JMSException ex) {
            System.err.println("Error when trying to receive message: "
                                  + ex.getMessage());
        }
        System.out.println("------");
        if (order.toLowerCase().contains("quit")) {
            stopReceiving = true;
        }
    }
}
```

asynchronous messaging

using message-oriented middleware

```
public class OrderConsumer {

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

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

    public static void main(String[] args) throws InterruptedException {
        JMSContext context = connectionFactory.createContext();
        JMSConsumer consumer = context.createConsumer(queue);

        System.out.println("I am now ready to receive orders");

        while (true) {
            String order = consumer.receiveBody(String.class);
            System.out.println("The order is to " + order);
            if (order.toLowerCase().contains("quit")) {
                break;
            }
        }
        System.out.println("\nBye bye!");
    }
}
```

development

point-to-point model

⇓

destination = queue

blocking version

blocking call

blocking call until timeout expires

```
String order = consumer.receiveBody(String.class, 1000);
```

asynchronous messaging

using message-oriented middleware

development

publish/subscribe model

⇓

destination = topic

```java
public class NewsPublisher {

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

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

    public static void main(String[] args) throws IOException, InterruptedException {
        JMSContext context = connectionFactory.createContext();
        JMSProducer producer = context.createProducer();
        int numberOfNews;

        do {
            numberOfNews = askForNumberOfNews(3);

            for (int i = 0; i < numberOfNews; i++) {
                String news = getNextNews();
                System.out.println("Sending news: " + news);
                producer.send(topic, news);
                Thread.sleep(2000);
            }
        } while (numberOfNews > 0);

        System.out.println("No more news to send. Bye bye!");
        producer.send(topic, "quit");
    }
}
```

asynchronous messaging

using message-oriented middleware

```java
public class NewsSubscriber implements MessageListener {

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

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

    private static boolean stopReceiving = false;

    public static void main(String[] args) throws InterruptedException {
        JMSContext context = connectionFactory.createContext();
        JMSConsumer consumer = context.createConsumer(topic);

        MessageListener listener = new NewsSubscriber();
        consumer.setMessageListener(listener);
        for (int i = 0; i < 60; i++) {
            Thread.sleep(1000);
            System.out.print(".");
            if (stopReceiving) {
                break;
            }
        }
        System.out.println("\nBye bye!");
    }
    :
```

development

publish/subscribe model

⇓

destination = topic

```java
    :
    @Override
    public void onMessage(Message message) {
        System.out.println();
        String news = "quit";
        try {
            news = ((TextMessage) message).getText();
            System.out.println("I received the following news: " + news);
        } catch (JMSException ex) {
            System.err.println("Error when trying to receive message: "
                                 + ex.getMessage());
        }
        System.out.println("------");
        if (news.toLowerCase().contains("quit")) {
            stopReceiving = true;
        }
    }
}
```

non-blocking version

asynchronous messaging

using message-oriented middleware

development with message-driven enterprise beans

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "jms/DOPOrderQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue")
})
public class OrderConsumer implements MessageListener {

    @Override
    public void onMessage(Message message) {
        try {
            System.out.println(this + " received the order to " + ((TextMessage) message).getText());
        } catch (JMSException ex) {
            System.err.println("Error when trying to receive message: " + ex.getMessage());
        }
        System.out.println("------");
    }
}
```

point-to-point

publish/subscribe

```
@MessageDriven(activationConfig = {
   @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "jms/DOPNewsTopic"),
   @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic")
})
public class NewsSubscriber implements MessageListener {

    @Override
    public void onMessage(Message message) {
                System.out.println();
        try {
            String news = ((TextMessage) message).getText();
            System.out.println(this + " received the following news: " + news);
        } catch (JMSException ex) {
            System.err.println("Error when trying to receive message: " + ex.getMessage());
        }
        System.out.println("------");
    }
}
```

non-blocking version

by definition

asynchronous messaging

using message-oriented middleware

header

properties

body

a jms message is composed of three parts:

- a **header** holding required fields for the message broker, e.g., priority, time-to-live, etc.
- a list of **optional properties** acting as meta-data for the message selection mechanism
- a **body** containing the actual data of the message (what that is)

a jms message can be of various types, based on what data is in its body

- `TextMessage message = context.createTextMessage("Hello world!");`
- `ObjectMessage message = context.createObjectMessage(someSerializableObject);`
- `MapMessage message = context.createMapMessage();`
- `Message message = context.createMessage();`

a consumer can select messages in term of their properties (meta-data)

consumer side

```
String selector = "name LIKE 'Max' OR (age > 18 OR address LIKE 'Lausanne')";
consumer = context.createConsumer(topic, selector);
```

producer side

```
Message message = session.createMessage();
message.setStringProperty("name", "Bob");
message.setIntProperty("age", 30);
message.setStringProperty("address", "Lausanne");
```

asynchronous messaging

using message-oriented middleware

parameterized quality of service

compared to other asynchronous messaging solutions, a message-oriented middleware offers flexible quality of service expressed in terms of:

- message ordering, priorities & time-to-live
- acknowledgement modes & transactions
- durable subscribers
- delivery modes

asynchronous messaging

using message-oriented middleware

message ordering

- messages are received in the order* they were sent with respect to a given producer
- no ordering is guaranteed across producers created from different contexts

*fifo order = first in first out

message priorities

- priorities allow programmers to have finer control over ordering of messages
- priorities range from 0 (lowest) to 9 (highest), e.g., `producer.setPriority(5);`

message time-to-live

- the time-to-live specifies how long the broker should keep the message at most
- the time-to-live is expressed in milliseconds, e.g., `producer.setTimeToLive(20000);`

asynchronous messaging

using message-oriented middleware

acknowledgment modes and transactions

- a acknowledgment informs the broker that the client did received a message

AUTO_ACKNOWLEDGE	messages are automatically acknowledged by the context `context = connectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE);`
DUPS_OK_ACKNOWLEDGE	messages are automatically acknowledged by the context but duplicate messages are possible in case of failures (but more efficient than AUTO_ACKNOWLEDGE)
CLIENT_ACKNOWLEDGE	client acknowledges messages itself, invoking `acknowledge()` on each message
SESSION_TRANSACTED	messages are grouped in the context of local transactions that are committed by explicitly calling `context.commit()` or rolled back by calling `context.rollback()` as soon as messages are sent or received via a transacted session, the first transaction starts and those messages will be grouped, until the client calls either `context.commit()` or `context.rollback()` after this call, the current transaction terminates and a new one is started

asynchronous messaging

using message-oriented middleware

acknowledgment modes and transactions

- the termination of a transaction affects a producer and a consumer as follows:

on a producer, what happens to messages sent during that transaction?
after `context.commit()`, all messages are effectively sent
after `context.rollback()`, all messages are disposed

on a consumer, what happens to messages received during that transaction?
after `context.commit()`, all messages are disposed
after `context.rollback()`, all messages are recovered and might be delivered again as part the newly started transaction

why "might be delivered again" and not "will be delivered again" ?

asynchronous messaging

using message-oriented middleware

durable subscribers

- in the publish/subscribe model, messages are only received by subscribers that are connected at the time of the publication and lost for late comers

- a durable subscriber is one that needs to receive all messages published on a topic, even those published when it was disconnected from the broker

- to tell the broker what messages should be kept for a disconnected durable subscriber, we must provide a unique name identifying that subscriber:

```
consumer = context.createDurableConsumer(topic, "fomo");
```

- to delete the state maintained by the broker for a durable subscriber:

```
context.unsubscribe("fomo");
```

asynchronous messaging

using message-oriented middleware

delivery modes

- delivery modes allow to balance transport reliability and throughput, depending whether the occasional loss of messages is tolerable or not
- in jms, there exists two delivery modes:

NON_PERSISTENT

most efficient but less reliable, since messages are guaranteed to be delivered at most once, i.e., some might be lost due to network failures

```
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

PERSISTENT

most reliable, since messages are guaranteed to be delivered once and only once, which is usually achieved by persisting sent messages on stable storage

```
producer.setDeliveryMode(DeliveryMode.PERSISTENT);
```