Asynchronous Interactions

Unil | HEC | dop lab
Benoît Garbinato
distributed object programming lab

Overview

- Goal: avoid blocking the caller (client) or forcing it to poll the callee (server) if the latter did not finish its computation
- Difference with asynchronous messaging: no space decoupling, only time decoupling
- Possible approaches:
  - asynchronous methods
  - web sockets

Asynchronous methods (1)

- A session bean can implement asynchronous methods, in order to increase throughput and response time, typically in the case of processor-intensive computation
- With an asynchronous method, the container returns the control to the client before the method is actually invoked and executes it in the background (asynchronously)
- An asynchronous method must return void or a Future<V> object; 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 (2)

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

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

Asynchronous methods (3)

```
Future<Double> value = myPortfolio.computeValue();
...   // Some time goes by...
System.out.println("Portfolio is worth $" + value.get());
```

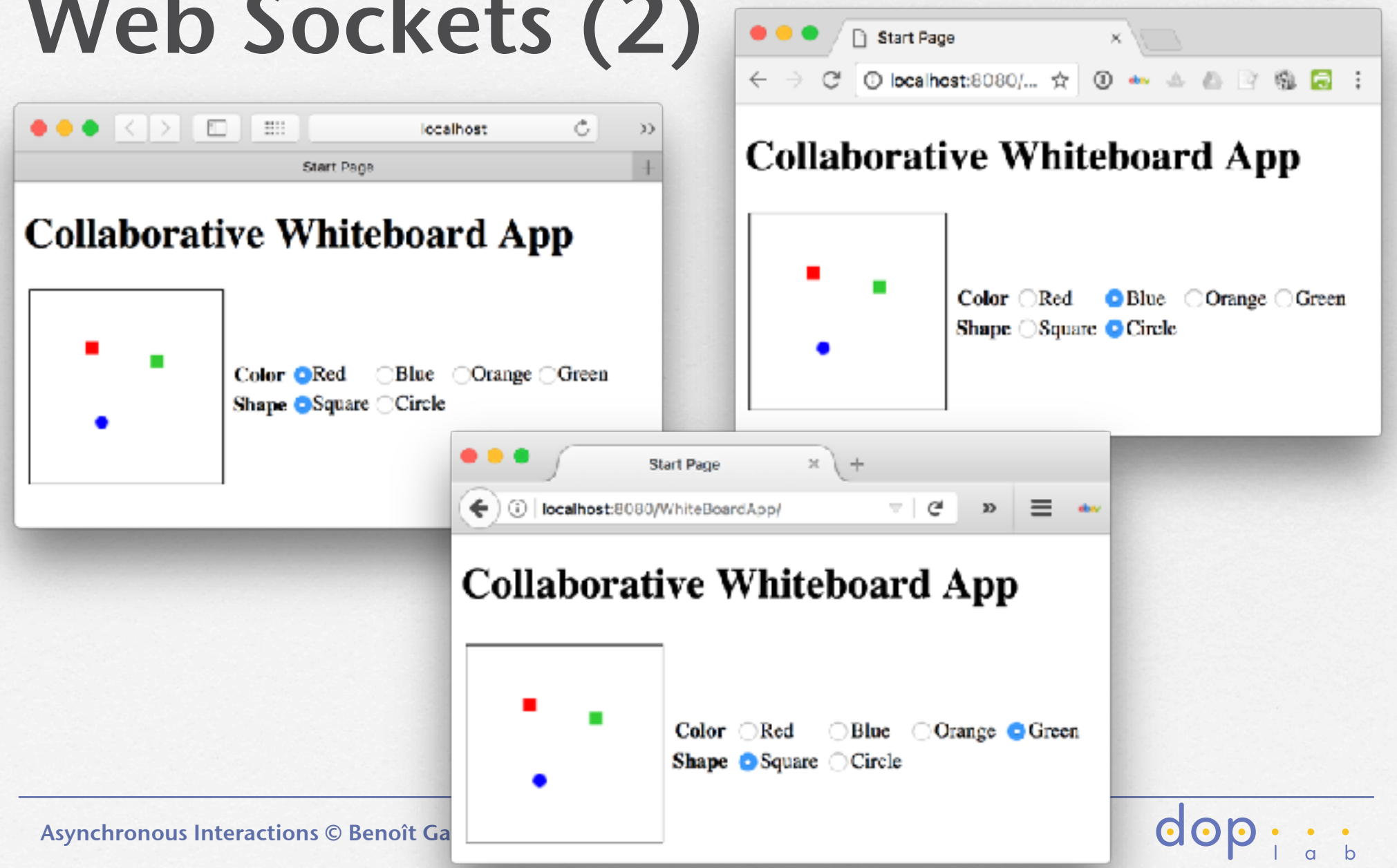
```
Future<Double> value = myPortfolio.computeValue();
try {
  System.out.println("Portfolio is worth $" + value.get(5, TimeUnit.SECONDS));
} catch (TimeoutException ex) {
  value.cancel(true);
  System.err.println("Timeout: operation was cancelled");
}
```

```
@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);
}
```

Web Sockets (1)

- 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 TPC sockets but they offer streams of messages rather than streams of bytes
- Web sockets are supported by most web browsers and web servers and rely on 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

Web Sockets (2)



dop lab

Web Sockets (3) – Server Side

```
@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 {
   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);
  }
}
```

Web Sockets (4) – Client Side

```
<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>
```

index.html

Web Sockets (5) – Client Side

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

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

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

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

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

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

Web Sockets (6) – Client Side

```
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);
}
```

```
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
  };
}
```

whiteboard.js

```
function drawImageText(image) {
 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;
  }
}
```

Web Sockets (7) – Server Side

```
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();
  }
}
```

```
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");
  }
}
```

```
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");
  }
}
```