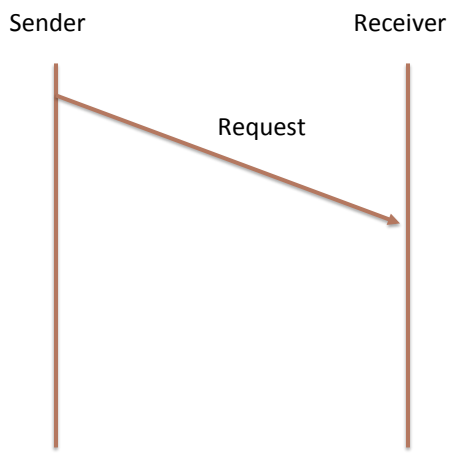
Asynchronous Communication

Dominic Duggan
Stevens Institute of Technology

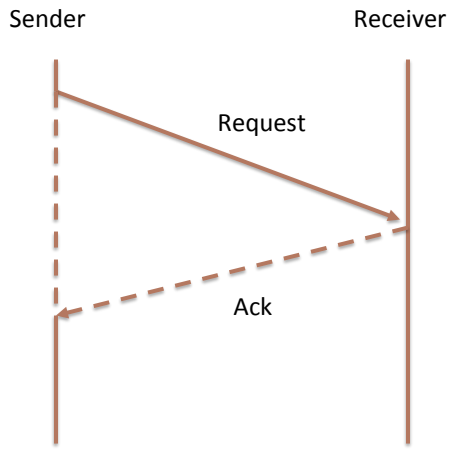
ASYNCHRONOUS COMMUNICATION

2

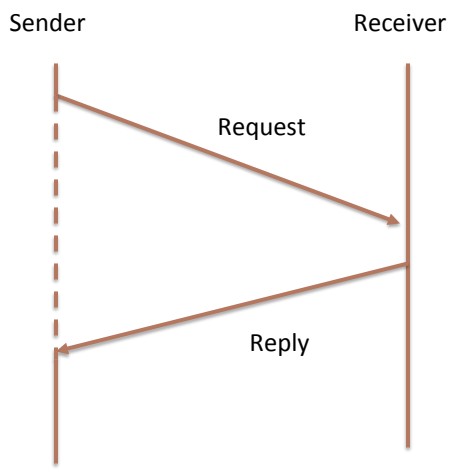
Non-blocking (Async) Send



Send with Acknowledgement

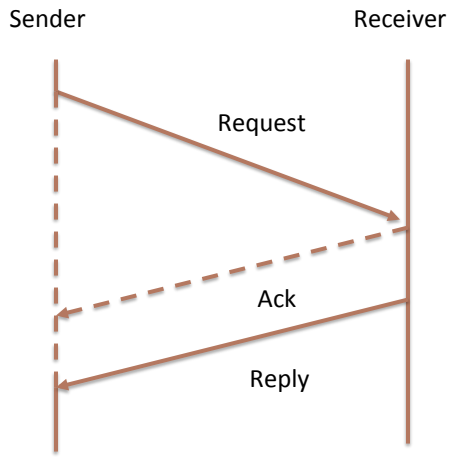


Send with Reply

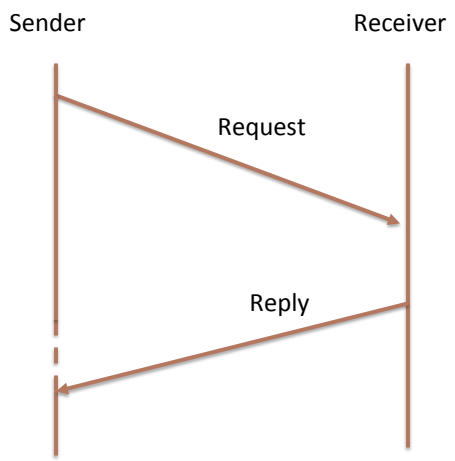


5

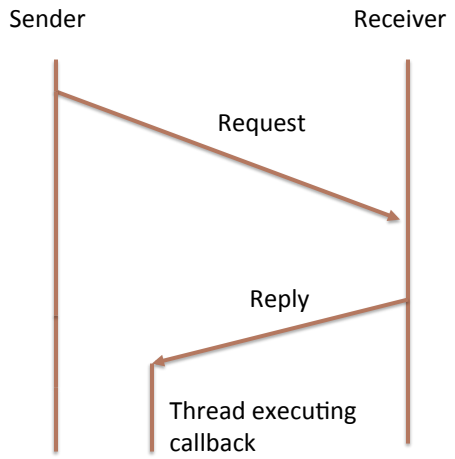
Send with Ack and Reply



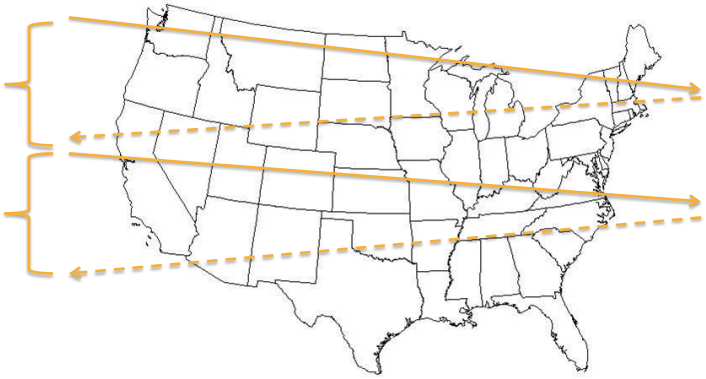
Async Send with Future



Async Send with Callback

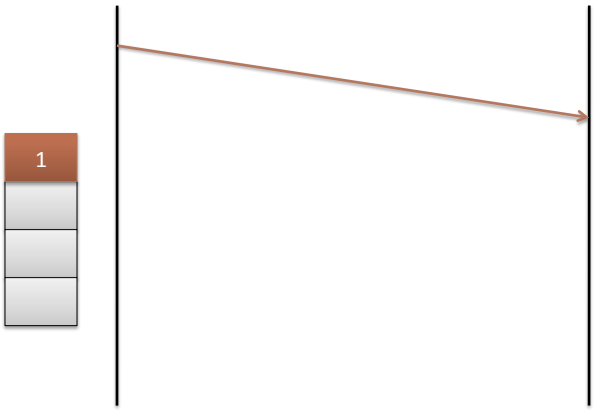


Latency



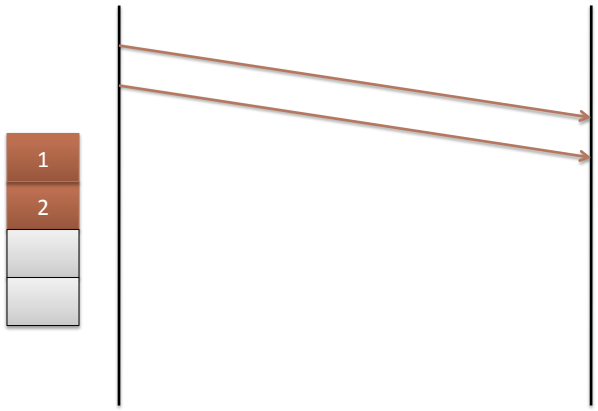
9

Pipelining



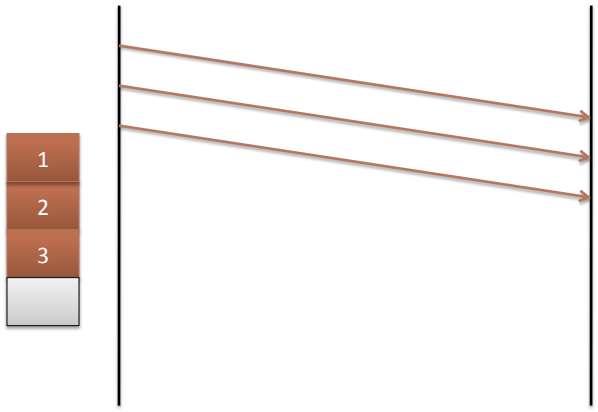
10

Pipelining



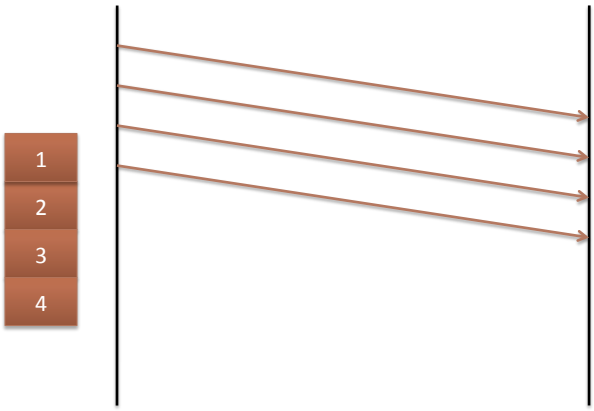
11

Pipelining



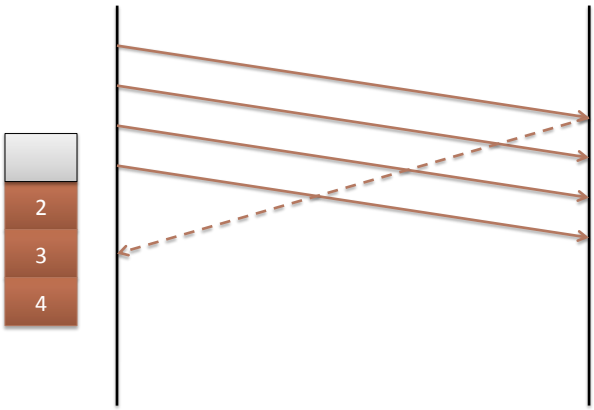
12

Pipelining



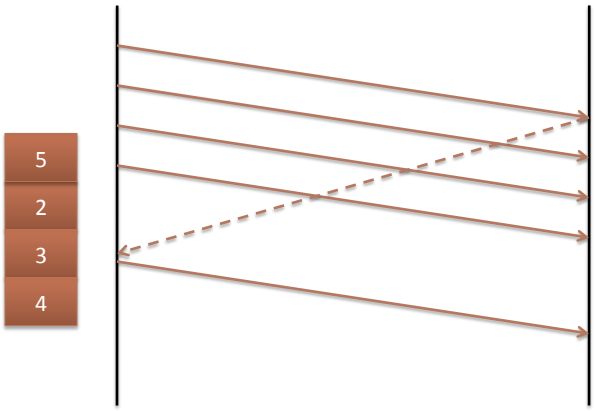
13

Pipelining



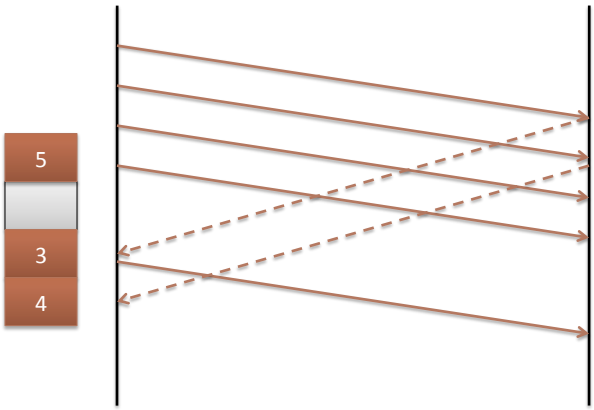
14

Pipelining



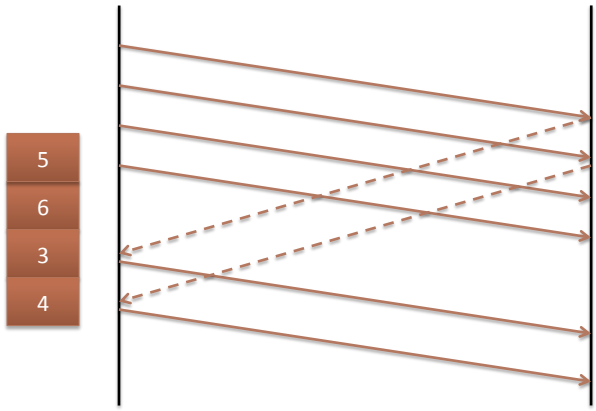
15

Pipelining



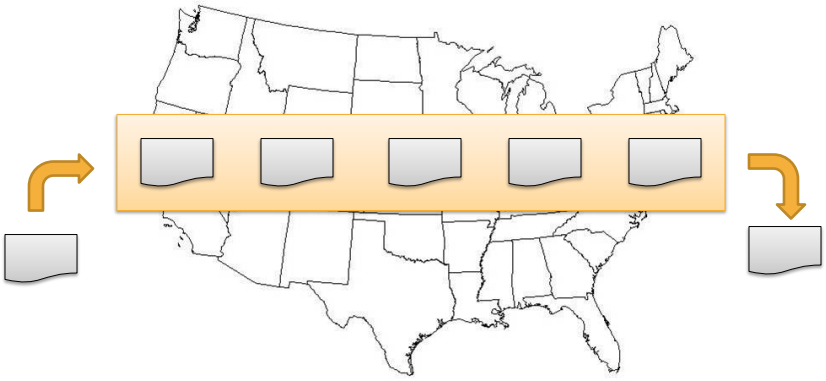
16

Pipelining



17

Pipelining

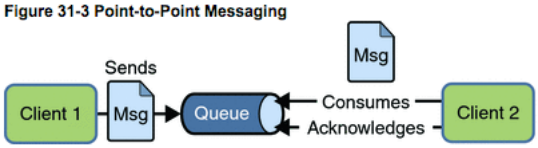


18

RELIABLE MESSAGE QUEUES

Point-to-point messaging

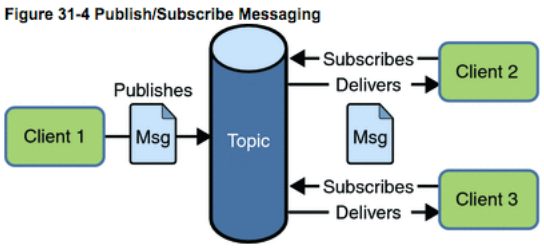
Figure 31-3 Point-to-Point Messaging



- Every message is consumed by only one client
- No time dependency between sender and receiver
- Consumer acknowledges message
- Queues are persistent

20

Publish-Subscribe Messaging



- Every message can be consumed by several clients
- Time dependency: clients only consume messages that arrive after registration
- Durable: consumers disconnect, collect messages

21

Message Consumption

- Synchronous consumption*
  - JMS clients can receive messages explicitly
    - Blocking mode `receive()`
    - Time-out mode `receive(int timeout)`
- Asynchronous consumption
  - JMS clients can register message listener `setMessageListener(...)`

* Send is still non-blocking.

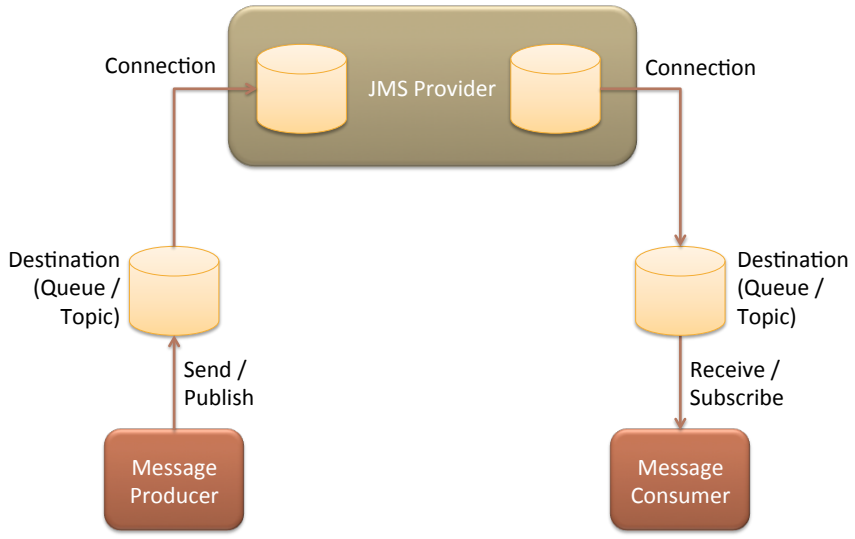
22

Java Messaging System (JMS)

- Layer in Java EE for 3rd party resources
  - Message queues (IBM MQSeries, etc)
  - Other legacy resources
- Transactional semantics
  - Message receives & sends
- Open source versions available
  - E.g. OpenMQ

23

JMS Architecture

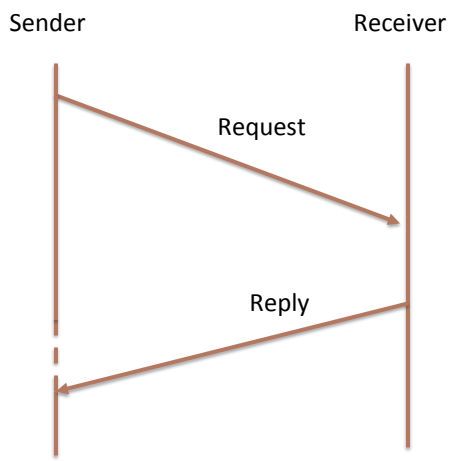


ASYNC SERVICE INVOCATION

Async Service Invocation

- Java EE (EJB & Web Service)
  - One-way method invocation
  - Invocation with futures
- WCF
  - Futures
  - Completion callbacks
  - Duplex

Async Send with Future



EJB Interface

```
@Remote
public interface ShoppingCartRemote {
  public float computeTax (String state);

  @Asynchronous
  public void
      add (String product, int amount);

  @Asynchronous
  public Future<String>
      checkOut (int purchOrder);
}
```

EJB Service with Future

```
public class ShoppingCartBean implements
                              ShoppingCartRemote {
  @Resource SessionContext ctx;
  public Future<String> checkOut (int purchOrder) {
      ... do some processing, generate url ...
      if (ctx.wasCancelled) return null;
      else return new AsyncResult<String>(url);
  }
}
```

EJB Client

```
public class ShoppingCartBean implements
                              ShoppingCartRemote {
  @Resource SessionContext ctx;
  public Future<String> checkOut (int purchOrder) {
      ... do some processing, generate url ...
      if (ctx.wasCancelled) return null;
      else return new AsyncResult<String>(url);
  }
}

Future<String> result = shoppingCart.checkOut(...);
...
String shippingUrl = result.get();
```

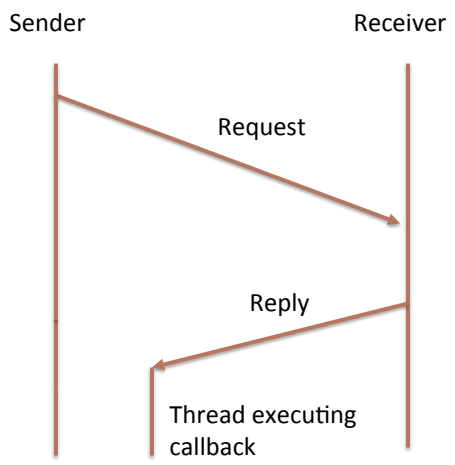
EJB Client

```
@Remote
public interface ShoppingCartRemote {
  public float computeTax (String state);

  @Asynchronous
  public Future<String>
      checkOut (int purchOrder);
}

Future<String> result = shoppingCart.checkOut(...);
...
String shippingUrl = result.get();
```

WCF Client

```
[ServiceContract]
public interface ShoppingCartRemote {
  [OperationContract(ASyncPattern=true)]
  public IAsyncResult BeginCheckOut
    (int purchOrder,
     AsyncCallback cb,
     object asyncstate);
  public string EndCheckOut(IAsyncResult result);
}

IAsyncResult result = cart.BeginCheckOut(...);
...
String shippingUrl = cart.EndCheckOut(result);
```

Async Send with Callback



WCF Completion Callback

```
[ServiceContract]
public interface ShoppingCartRemote {
  [OperationContract(ASyncPattern=true)]
  public IAsyncResult BeginCheckOut
    (int purchOrder,
     AsyncCallback cb,
     object asyncstate);
  public string EndCheckOut(IAsyncResult result);
}

void CompletionCB(IAsyncResult result) {
  Console.WriteLine(cart.EndCheckOut(result);
}
IAsyncResult result =
  cart.BeginCheckOut(..., CompletionCB, null);
```

WCF Duplex Interface

```
public interface IShoppingCB {
  [OperationContract(IsOneWay=true)]
  void Confirm (Uri tracking);
}

[ServiceContract(
  CallbackContract = typeof(IShoppingCB))]
public interface IShoppingCart {
  ...
  [OperationContract(IsOneWay=true)]
  void Checkout (String shipToAddr);
}
```

WCF Duplex Implementation

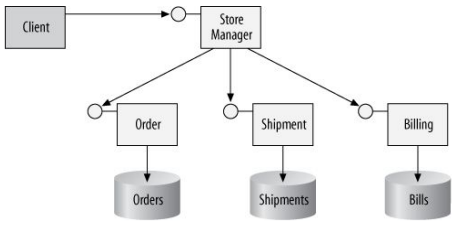
```
[ServiceBehavior(InstanceContextMode =
  InstanceContextMode.PerSession)]
public class ShoppingCartService : IShoppingCart {
  IShoppingCB callback = null;
  public ShoppingCartService() {
      callback =
            OperationContext
            .Current
            .GetCallbackChannel<IShoppingCB>();
  }
  public void Checkout (String shipToAddr) {
      ... callback.Confirm(...);
  }
}
```

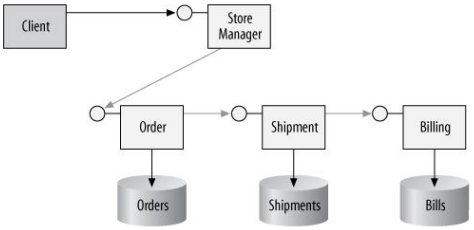
WCF Duplex Configuration

```
<system.serviceModel>
<services>
  <service name="ShoppingCartService">
      <endpoint
            address =
                  "http://localhost:9000/Shopping"
            binding = "wsDualHttpBinding"
            bindingConfiguration = ""
            contract = "IShoppingCart" />
  </service>
</services>
</system.serviceModel>
```

Synchronous vs Asynchronous

- Synchronous service called asynchronously: unlikely in practice





38

MESSAGING VS ASYNC SERVICE

Messaging vs Async Service

- Messaging
  - Persistent, transactional
  - Loosely coupled
- Async Service via Futures
  - Avoid suspension of call stack
  - Still tightly coupled
- Async Service via Callbacks
  - Java EE: Layering over futures
  - WCF Completion CB: Layering over futures
  - WCF Duplex: Two one-way sessional communications

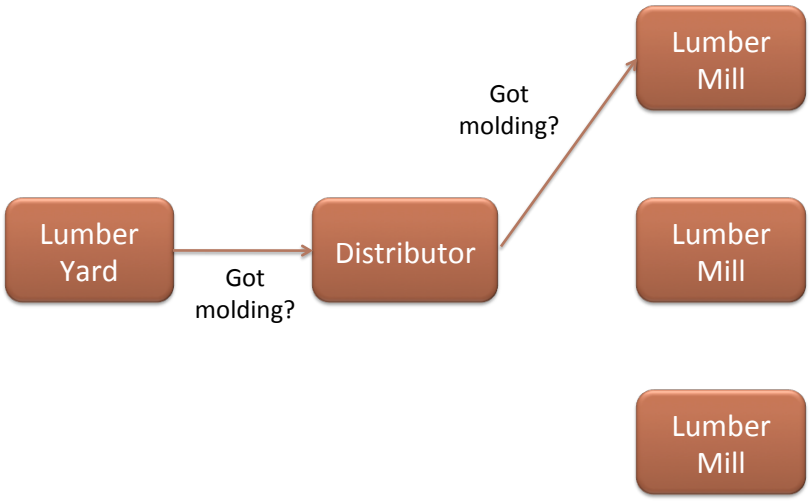
Messaging vs Async Service

- Messaging
  - Persistent, transactional
  - Loosely coupled
- Async Service via Futures
  - Avoid suspension of call stack
  - Still tightly coupled
- Async Service via Callbacks
  - Java EE: Layering over futures
  - WCF Completion CB: Layering over futures
  - WCF Duplex: Two one-way sessional communications

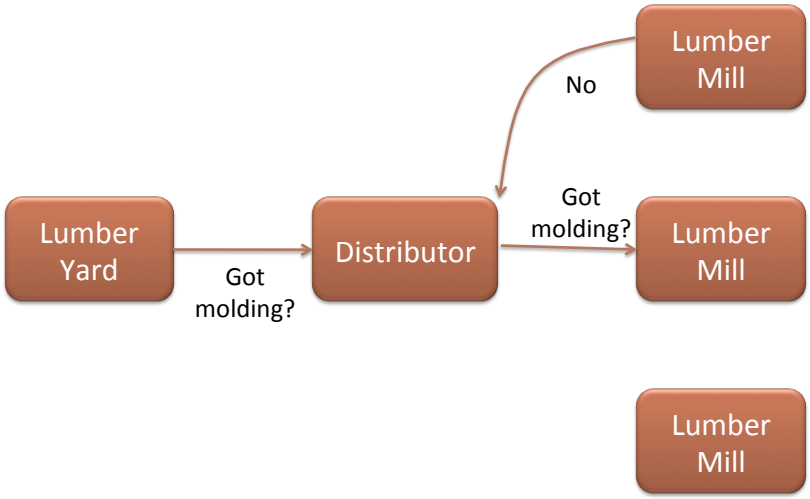
Messaging vs Async Service

- Messaging
  - Persistent, transactional
  - Loosely coupled
- Async Service via Futures
  - Avoid suspension of call stack
  - Still tightly coupled
- Async Service via Callbacks
  - Java EE: Layering over futures
  - WCF Completion CB: Layering over futures
  - WCF Duplex: Two one-way sessional communications

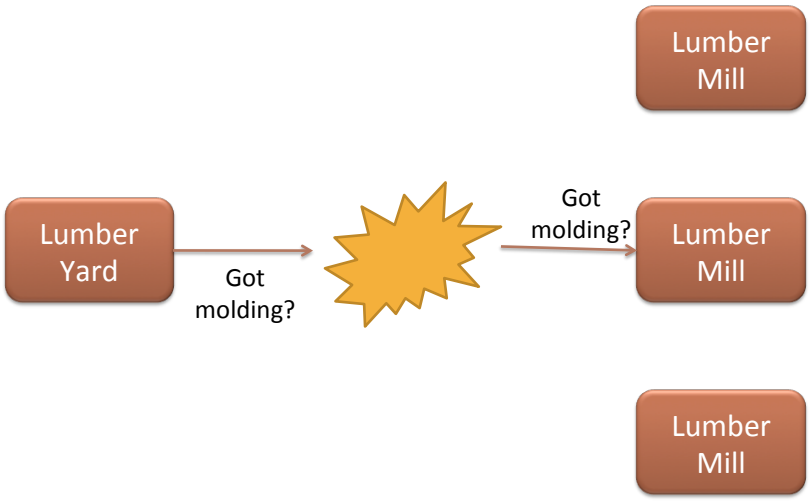
Messaging vs Async Service



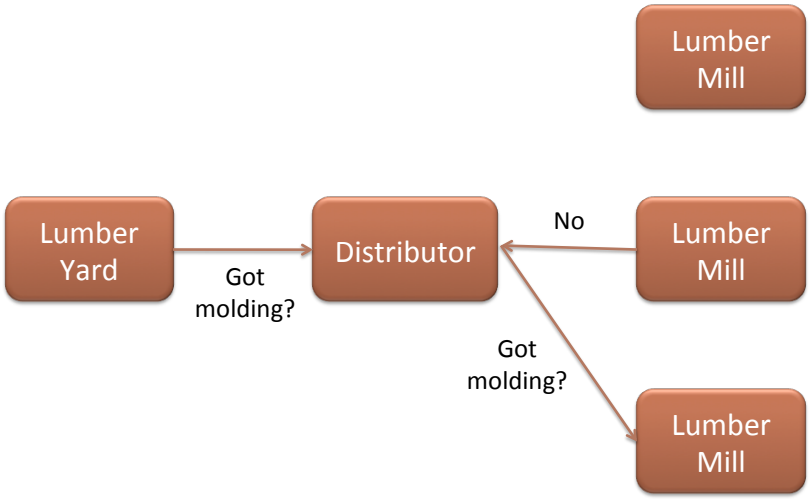
Messaging vs Async Service



Messaging vs Async Service



Messaging vs Async Service



Messaging: Pros and Cons

- Pros:
  - Persistent (survives crashes)
  - Transactional
  - Long-lived communication
    - Callbacks for pseudo-conversational

Messaging: Pros and Cons

- Pros:
  - Persistent (survives crashes)
  - Transactional
  - Long-lived communication
    - Callbacks for pseudo-conversational
- Cons
  - App must manage context
    - including persistent storage
  - Cannot rely on call-stack

Based on materials by Romain Grecourt,
Jitendra Kotamraju and Marek Potociar

SERVER-SIDE EVENTS AND WEBSOCKETS IN JAVA EE

JAX-RS Resources

```
@Path("orders")
@Produces("application/xml")
@Consumes("application/xml")
public class OrderResource {
    @GET
    public List<Order> getOrders() { … }

    @GET
    @Path("{id}")
    public Order getOrder(@PathParam("id") String orderId,
                          @HeaderParam("From") String from)
     { … }

    @Path("{id}/customer")
    public CustomerResource customer(…) { … }

}
```

JAX-RS

- Injectable information
  - Request, HttpHeaders, UriInfo, …
- Advanced HTTP response construction
  - Response, ResponseBuilder
- Message content handlers (a.k.a entity providers)
  - MessageBodyReader & MessageBodyWriter
- Error handlers
  - ExceptionMapper
- Other APIs aiding HTTP request/response processing

JAX-RS 2.0 Client API

```
Client client = ClientFactory.newClient();

WebTarget ordersTarget =
      client.target("http://example.com/eshop/orders");

WebTarget orderTarget = ordersTarget.path("{id}");

Order order = orderTarget.resolveTemplate("id", "1234")
                         .request("application/xml")
                         .get(Order.class);
```

JAX-RS 2.0 Client API

```
Order newOrder = new Order(…);
Response response =
   ordersTarget.request("text/plain")
               .post(Entity.xml(newOrder));


if (response.getStatus() == 200) {
    String orderId = response.readEntity(String.class);
    Link paymentLink = response.getLink("payment");

    client.target(paymentLink).request()…
}
```

SERVER-SIDE EVENTS

Server-Sent Events

```
: an example of a SSE event
id: 1
event: text-message
data: Hello, this is a
data: multi-line message.
<blank line>
```

Server-sent Events in Jersey

- Server side
  - `OutboundEvent`
  - `EventChannel`
  - `SseBroadcaster`
  - `BroadcasterListener`
- Client side
  - `InboundEvent`
  - `EventSource`
  - `EventListener`

SSE Server-side

```
@Path("message/stream")
public class MessageStreamResource {
    private static SseBroadcaster broadcaster =
                                new SseBroadcaster();

    @GET
    @Produces(SseFeature.SERVER_SENT_EVENTS)
    public EventOutput getMessageStream() {
        final EventOutput eventOutput = new EventOutput();
        broadcaster.add(eventOutput);
        return eventOutput;
    }

    ...
```

SSE Server-side

```
    private static AtomicLong nextMessageId
                              = new AtomicLong(0);

    @PUT
    @Consumes(MediaType.APPLICATION_JSON)
    public void putMessage(Message message) {
        OutboundEvent event = new OutboundEvent.Builder()
              .id(String.valueOf(
                   nextMessageId.getAndIncrement()))
              .mediaType(MediaType.APPLICATION_JSON_TYPE)
              .data(Message.class, message)
              .build();

        broadcaster.broadcast(event);
    }
}
```

SSE Client-side

```
EventSource events =
        new EventSource(target.path("message/stream")) {
    @Override
    public void onEvent(InboundEvent event) {
        String name = event.getName();
        Message message = event.getData(Message.class);
        display(name, message);
    }
};

…

events.close();
```

JAX-RS / Jersey

- Specification project: http://jax-rs-spec.java.net
- Implementation project: http://jersey.java.net

WEBSOCKET / TYRUS

“Real-time” interaction

- Update client
  - Chat
  - Sports
  - Stocks
  - Gaming
  - etc

Real-time interaction

- Polling
  - Good if updates at known intervals
  - Much unnecessary connection opening/closing

Real-time interaction

- Long-Polling
  - Server keeps request open for set period
  - If event, send to client
  - Close connection otherwise
  - Could be worse than polling

Real-time interaction

- Streaming (Comet)
  - Client sends request
  - Server "streams" open response
  - Problem: buffering proxies
    - Latency
  - Problem: HTTP header overhead
    - Client requests
  - Problem: complexity
    - Mapping between client requests and streaming events

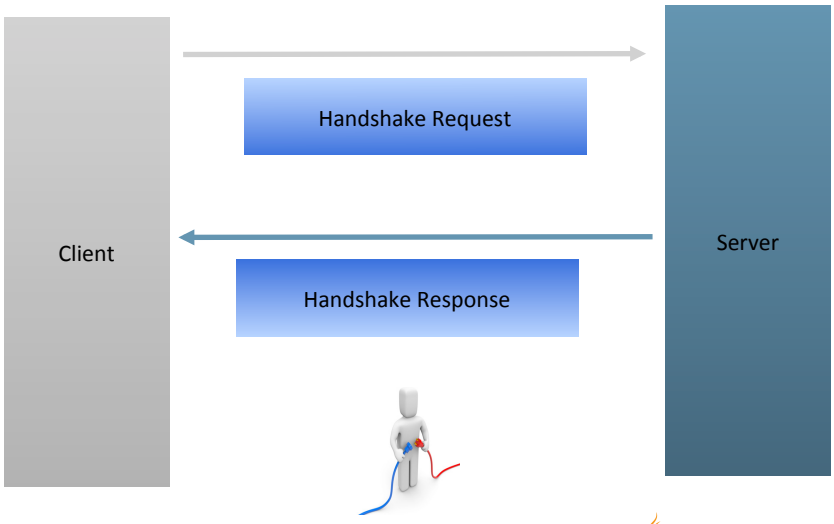
WebSockets

- TCP based, bi-directional, full-duplex messaging
- Originally proposed as part of HTML5
- IETF-defined **Protocol**: RFC 6455
  - Handshake
  - Data Transfer
- W3C defined **JavaScript API**
  - Candidate Recommendation

Basic Idea

- Establish a connection
  - Single TCP connection
- Send messages in both directions
  - Bi-directional
- Send message independent of each other
  - Full Duplex
- End the connection

Establish a connection



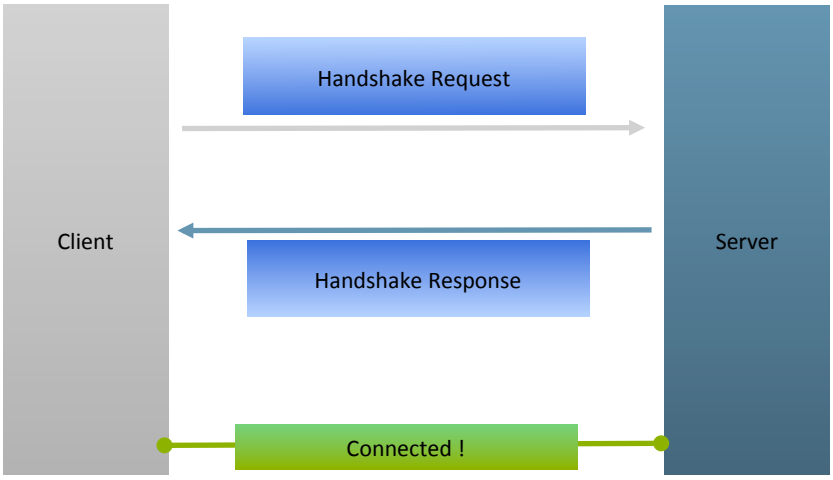
Handshake Request

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key:
      dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

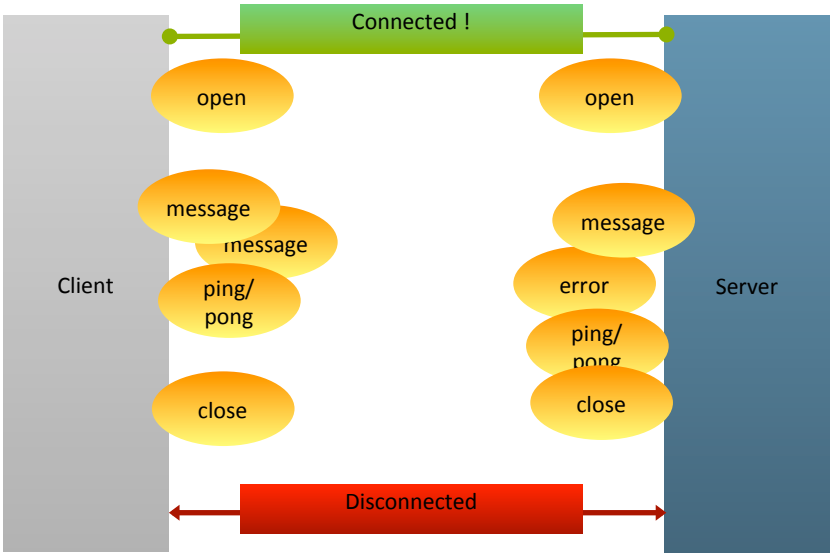
Handshake Response

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
    s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Establishing a Connection



WebSocket Lifecycle



WebSocket wire messages

- Capture traffic on loopback

Filter: http | Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
11	9.489449000	::1	::1	HTTP	648	GET /HelloWebSocket/ HTTP/1.1
13	9.491601000	::1	::1	HTTP	2134	HTTP/1.1 200 OK (text/html)
18	9.669322000	::1	::1	HTTP	501	GET /HelloWebSocket/echo HTTP/1.1
20	9.669489000	::1	::1	HTTP	543	GET /favicon.ico HTTP/1.1
22	9.670298000	::1	::1	HTTP	205	HTTP/1.1 101 Switching Protocols
24	9.671010000	::1	::1	HTTP	1624	HTTP/1.1 404 Not Found (text/html)
26	12.411987000	::1	::1	WebSocket	98	WebSocket Text [FIN] [MASKED]
28	12.413161000	::1	::1	WebSocket	108	WebSocket Text [FIN]
30	13.011122000	::1	::1	WebSocket	98	WebSocket Text [FIN] [MASKED]
32	13.013172000	::1	::1	WebSocket	108	WebSocket Text [FIN]

WebSocket Protocol Summary

- Starts with HTTP handshake
- Data transfer
  - Text/Binary frames
  - Ping/Pong control frames for keep-alive
  - Data frames don't have HTTP overhead
    - No headers/cookies/security/metadata
  - Close frame
- Full duplex and bi-directional

Advantages over TCP

- Upgrades from HTTP
- Origin domain
  - Prevent CSRF attacks
- Proxy connection without losing info
- Message-oriented
- Messages obfuscated by client
  - 32-bit mask, xor-ed with message
- Control frame for closing connection
- Built-in heartbeat support

WEBSOCKET API

The Web Sockets API

- Web Sockets API defines WebSocket javascript interface
- Event handlers for onopen(), onmessage(), onclose(), onerror()
- Send a String, Blob, ArrayBuffer using send()
- Supports sub protocols

JSR 356 Specification

- Standard Java API for creating WebSocket Applications
- Transparent Expert Group
  - jcp.org/en/jsr/detail?id=356
  - java.net/projects/websocket-spec

API Features

- Create WebSocket Client/Endpoints
  - Annotation-driven (`@ServerEndpoint`)
  - Interface-driven (`Endpoint`)
- Integration with Java EE Web container
  - CDI, Security, HttpSession etc.

Hello World Server

```
public class HelloServer extends Endpoint {
  @Override
  public void onOpen(Session session,
      EndpointConfig configuration) {
    session.addMessageHandler(
        new MessageHandler.Whole<String>() {
      public void onMessage(String name) {
        try {
          session.getBasicRemote().sendText("Hello " + name);
        } catch (IOException ioe) {
          // Handle failure.
        }
      }
    });
  }
}
```

Hello World Client

```
public class HelloClient extends Endpoint {
  @Override
  public void onOpen(Session session,
      EndpointConfig configuration) {
    try {
      session.getBasicRemote().sendText("Hello you!");
    } catch (IOException ioe) {
      . . .
    }
  }
}
```

Client Server Configuration

```
ServerContainer serverContainer =
    (ServerContainer) servletContext.getAttribute(
        “javax.websocket.server.ServerContainer”);

ServerEndpointConfig serverConfiguration =
    ServerEndpointConfig.Builder.create(
        HelloServer.class, "/hello").build();

serverContainer.addEndpoint(serverConfiguration);
...
```

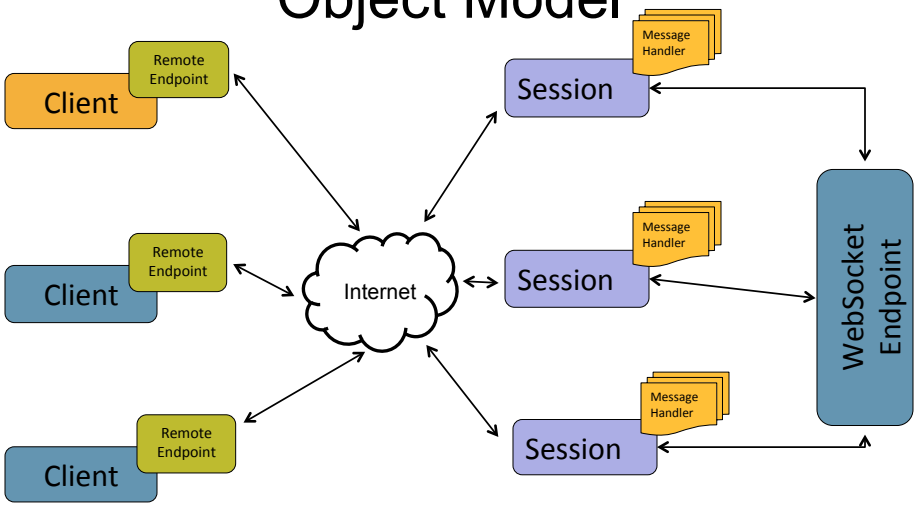
Client Server Configuration

```
URI clientURI =
      new URI("ws://myserver.com/websockets/hello");

WebSocketContainer container =
    ContainerProvider.getWebSocketContainer();

ClientEndpointConfig clientConfiguration =
    ClientEndpointConfig.Builder.create().build();

container.connectToServer(HelloClient.class,
    clientConfiguration, clientURI);
```

Main API Classes: javax.websocket.*

- **Endpoint**: Intercepts WebSocket lifecycle events
- **MessageHandler**: Handles all incoming messages for an Endpoint
- **RemoteEndpoint**: Represents the 'other end' of this conversation
- **Session**: Represents the active conversation

Object Model



Sending the Message

Whole string *	`RemoteEndpoint.Basic`	`sendText(String message)`
Binary data *	`RemoteEndpoint.Basic`	`sendBinary(ByteBuffer message)`
String fragments	`RemoteEndpoint.Basic`	`sendText(String part, boolean last)`
Binary data fragments	`RemoteEndpoint.Basic`	`sendBinary(ByteBuffer part, boolean last)`
Blocking stream of text	`RemoteEndpoint.Basic`	`Writer getSendWriter())`
Blocking stream of binary data	`RemoteEndpoint.Basic`	`OutputStream getSendStream()`
Custom object	`RemoteEndpoint.Basic`	`sendObject(Object customObject)`

* additional flavors: by completion, by future

Receiving the Message

Whole string	`MessageHandler.Whole<String>`	`onMessage(String message)`
Binary data	`MessageHandler.Whole<ByteBuffer>`	`onMessage(ByteBuffer message)`
String fragments	`MessageHandler.Partial<String>`	`onMessage(String part, boolean last)`
Binary data fragments	`MessageHandler.Partial<ByteBuffer>`	`onMessage(ByteBuffer part, boolean last)`
Blocking stream of text	`MessageHandler.Whole<Reader>`	`onMessage(Reader r)`
Blocking stream of binary data	`MessageHandler.Whole<InputStream>`	`onMessage(InputStream r)`
Custom object of type T	`MessageHandler.Whole<T>`	`onMessage(T customObject)`

JSR 356: Reference Implementation

- Tyrus: java.net/projects/tyrus
- Originated from WebSocket SDK
  - java.net/projects/websocket-sdk
- Works in stand alone, Java EE deployments
- Integrated in GlassFish 4 Builds

ANNOTATION-DRIVEN API

Hello World

```
import javax.websocket.annotations.*;

@ServerEndpoint("/hello”)
public class HelloBean {

  @OnMessage
  public void hello(String str) {
    // Receiving a "Hello World”
  }
}
```

WebSocket Annotations

Annotation	Level	Purpose
`@ServerEndpoint`	class	Turns a POJO into a WebSocket Server Endpoint
`@ClientEndpoint`	class	Turns a POJO into a WebSocket Client Endpoint
`@OnOpen`	method	Intercepts WebSocket Open events
`@OnClose`	method	Intercepts WebSocket Close events
`@OnMessage`	method	Intercepts WebSocket Message events
`@PathParam`	method parameter	Flags a matched path segment of a URI-template
`@OnError`	method	Intercepts errors during a conversation

Hello World

```
import javax.websocket.annotations.*;

@WebSocketEndpoint("/hello”,
     encoders = {ServerHello.class},
     decoders = {ClientHello.class})
public class HelloBean {

  @WebSocketMessage
  public ServerHello hello(ClientHello str) {
    return str;
  }
}
```

Lifecycle Events

```
@WServerEndpoint("/chat")
public class ChatBean {

    @OnOpen
    public void xxx(Session peer) { ... }

    @OnClose
    public void yyy(Session peer) { ... }

    @OnError
    public void zzz(Session peer) { ... }
```

Custom Payloads – Text

```
public class MyMessage
    implements Decoder.Text<MyMessage>, Encoder.Text<MyMessage> {
  private JsonObject jsonObject;

  public MyMessage decode(String s) {
    jsonObject = new Json.createReader(
        new StringReader(s)).readObject();
    return this;
  }

  public boolean willDecode(String string) {
    return true;  // Only if can process the payload
  }

  public String encode(MyMessage myMessage) {
    return myMessage.jsonObject.toString();
  }
}
```

Custom Payloads – Binary

```
public class MyMessage
    implements Decoder.Binary<MyMessage>, Encoder.Binary<MyMessage>
{

  public MyMessage decode(ByteBuffer bytes) {
    . . .
    return this;
  }
  public boolean willDecode(ByteBuffer bytes) {
    . . .
    return true;  // Only if can process the payload
  }
  public ByteBuffer encode(MyMessage myMessage) {
    . . .
  }
}
```

Chat Sample

```
@ServerEndpoint("/chat")
public class ChatBean {
    Set<Session> peers = Collections.synchronizedSet(…);

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

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

Chat Sample (Continued)

```
    . . .

    @OnMessage
    public void message(String message, Session client) {
        for (Session peer : peers) {
            peer.getRemote().sendObject(message);
        }
    }
}
```

URI Template Matching

```
@ServerEndpoint("/orders/{order-id}")
public class MyEndpoint {
  @OnMessage
  public void processOrder(
    @PathParam("order-id") String orderId) {
    ...
  }
}
```

@OnMessage Methods

- A parameter type that can be decoded in incoming message
  - String, primitive, Reader, ByteBuffer, byte[], InputStream, or any type for which there is a decoder
- An optional Session parameter
- Boolean partial flag
- 0..n String parameters annotated with `@PathParameter`
- A return type that can be encoded in outgoing message
  - String, primitive, Reader, ByteBuffer, byte[], InputStream, or any type for which there is an encoder

Tyrus / WebSocket

- Specification Project: http://websocket-spec.java.net
- Implementation: http://tyrus.java.net

JSON PROCESSING API

Standard JSON API

- Parsing/Processing JSON
- Data binding : JSON text <-> Java Objects
- Two JSRs (similar to JAXP and JAXB)
  - Processing/Parsing – Java EE 7
  - Binding – Java EE 8

Java API for Processing JSON

- Streaming API to produce/consume JSON
  - Similar to StAX API in XML world
- Object model API to represent JSON
  - Similar to DOM API in XML world

JSR-353: Java API for Processing JSON

- JsonReader – reads JsonObject/JsonArray from i/o

```
try(JsonReader reader = new JsonReader(io)) {
    JsonObject jsonObj = reader.readObject();
}
```

- JsonWriter – writes JsonObject/JsonArray to i/o

```
try(JsonWriter writer = new JsonWriter(io)) {
    writer.writeObject(jsonObj);
}
```

JSR-353: Java API for Processing JSON

- JsonBuilder – builds JSON object/array from scratch

```
JsonArray arr = new JsonBuilder()
    .beginArray()
        .beginObject()
            .add("type", "home”)
            .add("number", "212 555-1234")
        .endObject()
        .beginObject()
            .add("type", "fax”)
            .add("number", "646 555-4567")
        .endObject()
    .endArray()
.build();
```

```
[
    {
        "type": "home”,
        "number": "212 555-1234"
    },
    {
        "type": "fax”,
        "number": "646 555-4567"
    }
]
```

JSON Processing

- Projects
  - Specification Project - http://json-processing-spec.java.net
  - RI Project - http://jsonp.java.net