Pivotal

BUILT FOR THE SPEED OF BUSINESS

A Practical Use of Servlet 3.1: Implementing WebSocket 1.0

Mark Thomas

9 April 2014

Pivotal™

Agenda

- Introductions
- WebSocket
- Implementation aims
- Mapping to Servlet 3.1 features
- Complicating factors
- Summary
- Questions

Introductions

- markt@apache.org
- Apache Tomcat committer
- Developed the majority of Tomcat 7 and Tomcat 8
- ASF security team member
- ASF infrastructure volunteer
- Consultant Software Engineer at Pivotal
- Member of Servlet, WebSocket and EL expert groups
- Pivotal security team lead

WebSocket RFC 6455

- Defined in RFC 6455
- Asynchronous messages
  - Text
  - Binary
  - Control

- One persistent connection
  - No state management
- Uses HTTP upgrade
  - http://... -> ws://…
  - https://... -> wss://…

WebSocket RFC 6455

- Text and Binary messages
  - All text messages are UTF-8 encoded
  - 2^63 limit on data within a single frame
  - Messages may be split across multiple frames
  - No limit on message size

- Control messages
  - Limited to 125 bytes of data
  - May be sent at any time

- No multiplexing
  - draft extension

WebSocket JSR 356

- No requirement to build on Servlet 3.1
  - `HttpSession` passed as `Object` to avoid explicit dependency
- Configuration styles
  - Programmatic
  - Annotation

- Provides client and server APIs
- Client API is sub-set of server API

Implementation Aims

- JSR 356 compliant
- RFC6455 compliant
- Container neutral
  - Only depends on Servlet 3.1

- Is there a performance cost of container neutrality?
  - Will be some
  - Not significant

Mapping to Servlet 3.1 Features

- Single persistent connection
- Asynchronous messages
- Requires non-blocking IO for a scalable solution
  - Blocking IO is possible but it doesn’t scale

- Use Servlet 3.1 non-blocking IO

Mapping to Servlet 3.1 Features

- WebSocket connection starts with HTTP upgrade

- Use Servlet 3.1 HTTP upgrade

- Annotation configuration

- Use Servlet 3.0 annotation scanning

Annotation Scanning

- Feature added in Servlet 3.0

- Implement **`ServletContainerInitializer`**
- Add **`@HandlesTypes`**
- When web application starts the container calls

```
ServletContainerInitializer#
  onStartup(Set<Class<?>>, ServletContext)
```

Annotation Scanning

```
@HandlesTypes({ServerEndpoint.class,
               ServerApplicationConfig.class,
               Endpoint.class})
public class WsSci implements
    ServletContainerInitializer { …
```

Annotation Scanning

- **`ServerEndpoint`** for annotated endpoints
- **`Endpoint`** for programmatic endpoints
- **`ServerApplicationConfig`** for filtering endpoints

Annotation Scanning

- Need to scan every class for **`@HandlesTypes`** matches
- Scanning every class is (relatively) expensive
- Don’t want to scan if it isn’t necessary
- Servlet 3.0 provides options for minimizing scanning
  - Specification language wasn’t clear
  - Discovered Tomcat’s implementation wasn’t quite as intended

Annotation Scanning

- SCIs discovered in container provided JARs are always processed
- SCI discovery must follow the web application's class loader delegation model
- No specification requirements for the order that SCIs are invoked

Annotation Scanning

- SCIs are not loaded from web application JARs excluded using ordering preferences in web.xml
- JARs excluded from ordering preferences in web.xml are not scanned for classes to be handled by any SCI

- `<metadata-complete>` has no impact on SCI discovery or scanning of classes

HTTP Upgrade

- Feature added in Servlet 3.1
- Implement **`HttpUpgradeHandler`**
- Call **`HttpServletRequest#upgrade`**(…)
- Once the HTTP response has been sent to the client the container calls **`HttpUpgradeHandler#init(WebConnection)`**
- Use **`WebConnection`** to access the input and output streams

HTTP Upgrade

```
package javax.servlet.http;

public interface HttpUpgradeHandler {

  void init(WebConnection connection);

  void destroy();
}
```

- Interface applications must implement to handle upgraded connections

HTTP Upgrade

```
package javax.servlet.http;

public interface HttpServletRequest extends
    ServletRequest {

  public <T extends HttpUpgradeHandler> T
    upgrade(Class<T> httpUpgradeHandlerClass)
    throws IOException, ServletException;
}
```

- Method that triggers the upgrade process

HTTP Upgrade

```
package javax.servlet.http;

public interface WebConnection
    extends AutoCloseable {

  ServletInputStream getInputStream()
      throws IOException;

  ServletOutputStream getOutputStream()
      throws IOException;

}
```

- Only provides access to the input and output streams

HTTP Upgrade

- **`HttpUpgradeHandler`** implementations must have a zero argument constructor
- **`WebConnection`** only has access to the input and output streams
- Need to pass far more information to the **`HttpUpgradeHandler`** instance
- No API defined for passing this information
- Applications must provide their own

Pivotal™

HTTP Upgrade

```
public void preInit(
  Endpoint ep,
  EndpointConfig endpointConfig,
  WsServerContainer wsc,
  WsHandshakeRequest handshakeRequest,
  String subProtocol,
  Map<String,String> pathParameters,
  boolean secure) {
…
```

Non-blocking IO

- Feature added in Servlet 3.1
- New methods added to **`ServletInputStream`** and **`ServletOutputStream`**
- May only be used within asynchronous processing or upgraded connections
- Once switched to non-blocking IO it is not permitted to switch back to blocking IO

Non-blocking IO

```
package javax.servlet;
public abstract class ServletInputStream
    extends InputStream {
  …
  public abstract boolean isFinished();
  public abstract boolean isReady();
  public abstract void setReadListener(
    ReadListener listener);
}
```

Non-blocking IO

```
package javax.servlet;

public interface ReadListener extends
    java.util.EventListener{

  public abstract void onDataAvailable()
    throws IOException;

  public abstract void onAllDataRead()
    throws IOException;

  public abstract void onError(
    java.lang.Throwable throwable);

}
```

Non-blocking IO

- Start non-blocking read by setting the **`ReadListener`**
- Container will call **`onDataAvailable()`** when there is data to read
- Application may read once from the **`ServletInputStream`**
- Application must call **`ServletInputStream#isReady()`** before next read
- An **`IllegalStateException`** is thrown if applications don’t do this

Non-blocking IO

- If **`isReady()`** returns true, the application may read again from the **`ServletInputStream`**
- If **`isReady()`** returns false, the application must wait for the next **`onDataAvailable()`** callback
- The container will only call **`onDataAvailable()`** once **`isReady()`** has returned false and there is data to read
- The container will only call **`onAllDataRead()`** when the end of the **`InputStream`** is reached

Non-blocking IO

```
package javax.servlet;
public abstract class ServletOutputStream
    extends OutputStream {
  …
  public abstract boolean isReady();
  public abstract void setWriteListener(
    WriteListener listener);
}
```

Non-blocking IO

```
package javax.servlet;

public interface WriteListener extends
        java.util.EventListener{

    public void onWritePossible()
            throws IOException;

    public void onError(
            java.lang.Throwable throwable);

}
```

Non-blocking IO

- Start non-blocking write by setting the **`WriteListener`**
- Container will call **`onWritePossible()`** when data can be written without blocking
- Application may write once to the **`ServletOutputStream`**
- Application must call **`ServletOuputStream#isReady()`** before next write
- An **`IllegalStateException`** is thrown if applications don’t do this

Non-blocking IO

- If **`isReady()`** returns true, the application may write again to the **`ServletOutputStream`**
- If **`isReady()`** returns false, the application must wait for the next **`onWritePossible()`** callback
- The container will only call **`onWritePossible()`** once **`isReady()`** has returned false and data may be written without blocking

Non-blocking IO

```
private static class WsReadListener
        implements ReadListener {
    …
    public void onDataAvailable() {
        try {
            wsFrame.onDataAvailable();
        } catch … {
            …
        }
    }
}
```

Non-blocking IO

```
public class WsFrameServer extends WsFrameBase {
   public void onDataAvailable() throws IOException {
      synchronized (connectionReadLock) {
         while (isOpen() && sis.isReady()) {
            int read = sis.read(inputBuffer, writePos,
                 inputBuffer.length - writePos);
            if (read == 0) return;
            if (read == -1) throw new EOFException();
            writePos += read;
            processInputBuffer();
         }
      }
   }
}
```

Non-blocking IO

```
private static class WsWriteListener
        implements WriteListener {

    …

    public void onWritePossible() {
        wsRemoteEndpointServer.
                onWritePossible();

        }

    }

}
```

Non-blocking IO

```
public void onWritePossible() {
  boolean complete = true;
  try {
    while (sos.isReady()) {
      complete = true;
      for (ByteBuffer buffer : buffers) {
        if (buffer.hasRemaining()) {
          complete = false;
          sos.write(buffer.array(), buffer.arrayOffset(), buffer.limit());
          buffer.position(buffer.limit());
          break;
        }
      }
```

Non-blocking IO

```
     if (complete) {
        wsWriteTimeout.unregister(this);
        if (close) close();
        break;
      }
    }
  } catch (IOException ioe) {…}
  if (!complete) {
    long timeout = getSendTimeout();
    if (timeout > 0) {
      timeoutExpiry = timeout + System.currentTimeMillis();
      wsWriteTimeout.register(this);
    }
  }
}
```

Non-blocking IO

- Timeouts
  - Only have access to the `ServletInputStream` and `ServletOutputStream`
  - No API for setting timeouts
  - Had to create a timeout mechanism for WebSocket writes
- Thread safety
  - Lots of places to trip up
  - Write with multi-threading in mind
  - Test extensively

Complicating Factors: Non-blocking Styles

- Server uses Servlet 3.1 style
  - Read/write listeners and isReady()
- WebSocket API
  - **`java.util.concurrent.Future`**
  - **`javax.websocket.SendHandler`**
- Client uses AsynchronousSocketChannel
  - **`java.nio.channels.CompletionHandler`**
- Need to convert between these

Pivotal™

Complicating Factors: Non-blocking Styles

- **`Future`** always converted to **`SendHandler`**
- Server side
  - **`SendHandler`** mapped to Servlet 3.1 style
- Client side
  - **`SendHandler`** always converted to **`CompletionHandler`**

Pivotal™

Complicating Factors: Blocking Messages

- The WebSocket API
  - Some messages use blocking IO
  - Some messages use non-blocking IO
- The Servlet 3.1 API does not allow switching from non-blocking to blocking
- Square peg, meet round hole
- Have to simulate blocking

Complicating Factors: Blocking Messages

```
void startMsgBlock(byte opCode, ByteBuffer payload,
    boolean last) throws IOException {
  FutureToSendHandler f2sh = new FutureToSendHandler();
  startMessage(opCode, payload, last, f2sh);
  try {
    long timeout = getBlockingSendTimeout();
    if (timeout == -1) f2sh.get();
    else f2sh.get(timeout, MILLISECONDS);
  } catch (…) {
    throw new IOException(e);
  }
}
```

Complicating Factors: Blocking Messages

- No API to define a timeout for blocking messages
  - Specified via a user property on the session
  - Container specific solution
- What happens under the hood?
  - Data to write is written to the socket
  - Remaining data is buffered
  - Socket registered for write
  - Callback when socket ready for write
  - Repeat until buffer is empty

Complicating Factors: Blocking Messages

- How is the block implemented?
- Simple latch
  - Create a latch when the write starts
  - **`f2sh.get()`** calls **`latch#await()`**
  - Container calls **`latch.countDown()`** when write is complete
- This works for blocking writes on the application thread
- However…

Complicating Factors: Blocking Messages

- Servlet 3.1 (and earlier) is written based on the following assumption:
  - There is only ever one container thread accessing a socket at any one time
- Tomcat enforces this with a lock
  - Prevents all sorts of threading issues with async processing
- This causes big problems for WebSocket

Complicating Factors: Blocking Messages

- Start with an established but idle WebSocket connection
- Poller detects data is available to read
- Poller passes socket to container thread for processing
- Container thread obtains the lock for working with the socket
- Code path eventually reaches application code
- Application processes message

Complicating Factors: Blocking Messages

- Application replies with its own message using a blocking write
- Message is too big for a single write
- Message is partially written
- Remaining message is buffered
- Socket is registered with Poller for write

Complicating Factors: Blocking Messages

- Container thread blocks on latch as message write is not complete
- Poller detects data can be written
- Poller passes socket to container thread for processing
- Container thread blocks waiting for lock to allow it to work with the socket

Complicating Factors: Blocking Messages

- Deadlock
- The thread that initiated the write has the lock for the socket
- That thread is blocked waiting for the write to complete
- The thread that will allow the write to progress is blocked waiting for the lock for the socket

Complicating Factors: Blocking Messages

- Servlet EG discussed several options
- Automatic blocking
  - No call to **`isReady()`** results in a blocking read / write
  - Ends up in same deadlock situation
- **`WebConnection.start(Runnable)`**
  - Clunky
  - Purpose not immediately obvious
  - Should work but was untested

Complicating Factors: Blocking Messages

- For connections using HTTP upgrade, allow concurrent read and write
  - No more than one read thread
  - No more than one write thread
- Breaks the implied one thread per socket rule of the Servlet API
- It was the solution I tried first
  - It worked
  - Some threading issues

Complicating Factors: Generic Types

```
public interface MessageHandler {
  interface Partial<T> extends MessageHandler {
    void onMessage(T messagePart, boolean last);
  }
  interface Whole<T> extends MessageHandler {
    void onMessage(T message);
  }
}
```

Complicating Factors: Generic Types

- The container has to figure out what T is at runtime
- Has to do the same for **`Encoder`** implementations
- **`Foo implements MessageHandler.Whole<String>`**
  - Fairly simple
- **`Bar extends Foo`**
  - Still fairly simple
- It can get more complicated…

Complicating Factors: Generic Types

- `A extends B<Boolean,String>`
- `B<Y,X> extends C<X,Y>`
- `C<X,Y> implements`
  `        MessageHandler.Whole<X>, Other<Y>`
- Generic information is available at runtime

Complicating Factors: Generic Types

- Have to do a little digging to find it
  - **`Class#getGenericInterfaces()`**
  - **`ParameterizedType#getRawType()`**
  - **`ParameterizedType#getActualTypeArguments()`**
- org.apache.tomcat.websocket.Util#getGenericType()

Complicating Factors: UTF-8

- WebSocket text messages are always UTF-8 encoded
- Tomcat uses the Autobahn test suite to check for RFC6455 compliance
- Autobahn includes a lot of tests for UTF-8 handling
  - Autobahn has been incredibly useful
  - Highly recommended for developers of WebSocket clients or servers

Complicating Factors: UTF-8

- The UTF-8 decoder provided by the JRE triggers Autobahn failures
- Wrote some test cases that identified further failures
- WebSocket text messages are always UTF-8 encoded
- Tomcat uses the Autobahn test suite to check for RFC6455 compliance

Complicating Factors: UTF-8

- Autobahn includes a lot of tests for UTF-8 handling
  - Autobahn has been incredibly useful
  - Highly recommended for developers of WebSocket clients or servers
- The UTF-8 decoder provided by the JRE triggers Autobahn failures
- Wrote some test cases that identified further failures

Complicating Factors: UTF-8

- Issues with JRE provided UTF-8 decoder
  - It accepts byte sequences that should be rejected
  - It doesn’t fail fast on invalid sequences
  - Not failing fast means the wrong number of invalid bytes are detected
  - Not failing fast means too many bytes (including valid bytes) are incorrectly replaced with the replacement character

Complicating Factors: UTF-8

- Writing your own UTF-8 decoder is non-trivial
- Apache Harmony to the rescue
- Took the UTF-8 decoder from Apache Harmony
- This also had some failures
- Modified the decoder to fix the issues
- Switched to this new decoder for all Tomcat code including WebSocket

Complicating Factors: SSL

- **`AsynchronousSocketChannel`** is a good match for a WebSocket client implementation
- No SSL support
- Searching for implementations to reuse didn't find any implementations
- Had to write one from scratch
  - Based on Tomcat's HTTP NIO connector SSL implementation

Summary

- WebSocket 1.0 has been implemented on Servlet 3.1
- Tomcat 8
  - Also JSP 2.3 and EL 3.0
- There were some complications
- Had to ‘bend’ the Servlet specification to do it
- https://svn.apache.org/repos/asf/tomcat/trunk

Questions

Thank you

Pivotal

BUILT FOR THE SPEED OF BUSINESS