Proposal

JAX-RPC SOAP Message Handlers

DRAFT **Document Version 0.3**

1 Introduction

The Handler section of the JAX-RPC 1.1 specification needs to be more developed. Unlike the rest of the JAX-RPC specification, this one section deals with message FLOW and processing rules related to flow, many of which were ignored (roles on the SOAP node?). As such, we reposition handlers under the larger context of what it means to act as a SOAP node – and then define handlers in that context. Handlers & SOAP node processing deserve to be more properly defined before we bring further extensions into the handler model..

Wordsmithing, formatting, and reorganization has been done – along with wholesale rewrite, in an attempt to:

- Improve flow through material
- Minimize verbiage
- Tighten loose ideas.
- Relax unnecessarily restrictive constraints.
- Present spec statements first, qualifying statements and alternatives to follow.
- Eliminate redundant statements. Where appropriate, replace with references.

1.1 Significant Changes

1.1.1 DISCLAIMER

IT IS UNDERSTOOD THAT MUCH OF WHAT IS PRESENTED BELOW IS NOT BACKWARD COMPATIBLE WITH JAX-RPC 1.x.

ISSUES RELATED TO BACKWARDS COMPATIBILITY NEED TO BE DISCUSSED. I BELIEVE THAT IN ALL CASES THESE CAN BE RESOLVED, **YET FOR CLARITY OF THE UNDERLYING ISSUES THESE MUST BE ADDRESSED LATER. [I.E. LET'S GET IT RIGHT BEFORE TRYING TO MAKE IS COMPATIBLE]**

1.1.2 Eliminate HandlerChain

HandlerChain was an (internal) implementation artifact not exposed through the APIs. Combine mandated interface (List of Handler instances), Handler instance life-cycle management, implied chain life-cycle management, what was config map FOR?, HandlerInfo exposed to chain where/how, etc. We can be thankful it was NOT exposed through API's in its current state.

So, to begin discussion for JAX-RPC 2.0, we have eliminated HandlerChain.

If it is to be added back in, it must be redesigned & exposed (get/set HandlerChain instead of List), or all references to (alternate chain implementations) must be eliminated.

Arguments for exposing:

1. HandlerChain logic is somewhat simplified. SOAP actor/header/mustunderstand semantics no longer belong to the HandlerChain. They belong to the SOAPProtocolBinding (which properly encompasses the entire message path through handlers to target endpoint).

2. Customer or 3rd party could conceivably write their own Chain [and chain logic].

So, how would it be exposed?

- Customer instantiates an alternate HandlerChain implementation, populates List with HandlerInfos (or extension) [1.x spec mandates populate with Handler instances], and sets on ProtocolBinding. Note that this does NOT change the protocol bindings, whereas with the original design changing the chain implied changing SOAP semantics. Chain assumes responsibility for Handler instance life-cycle as mandated by spec.
- New interface, to be exposed to HandlerChain, that isolates Handler instances from HandlerChain. Let JAX-RPC SOAP binding runtime manage handler instances, their lifecycles, any pooling, etc. Get/set HandlerChain class name & "Map config" param. New interface to facilitate: a) obtaining and returning a handler instance from runtime, b) other?

```
package javax.xml.rpc.handler;

import javax.xml.rpc.MessageContext;

public interface HandlerChain extends java.util.List {
    boolean handleRequest(MessageContext context);

    boolean handleResponse(MessageContext context);
    boolean handleFault(MessageContext context);
    void    handleClosure(MessageContext context);

    void setRoles(String[] soapActorNames);[1] [2]
    String[] getRoles();

    // Lifecycle methods
    void init(java.util.Map config);
    void destroy();
}
```

1.1.3 Expose ProtocolBinding, push Handler Chain into ProtocolBindings

ProtocolBindings provides access to a ProtocolBinding (SOAPProtocolBinding), which:

- Presents a mechanism that allows programmatic configuration of the underlying binding for a specific port.
- Allows proper abstraction of SOAP bindings.
    - Roles apply to the port. Headers can be mapped (in)to target parameters, and can be marked as mustunderstand, for both client/server. Therefore SOAP semantic checking, as previously described for handlerChains, must apply to more than handlers. Why allow only SOAP handlers (which are not described by SOAP spec) to be configured, when there are SOAP attributes (described by spec) that are not configurable?[3]
    - Handlers and/or handler chains now associated with bindings, not exposed to Service (via HandlerRegistry). List of HandlerInfo.

[1] What is relationship with what is on SOAPMessageContext?

[2] Note that the algorithm described for checking roles/headers IGNORES the node-level roles/headers [those associated with the ultimate destination [target end-point]. Is the implication here that a HandlerChain be an independent SOAP node? This has STRONG implications either way. If independent, then it makes sense to describe the processing from the context of the c hain. If dependent, then we need to describe the overall behavior. In general, the relationship between a JAX-RPC endpoint and a SOAP node needs to be defined [even if loosely].

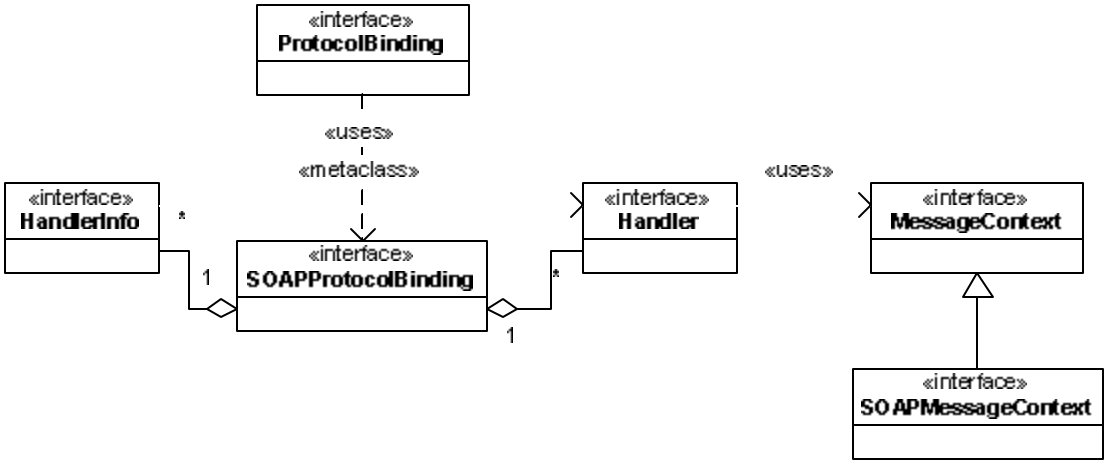
[3] Port is the lowest level at which it makes sense to scope actor roles. Service or application could makes more sense, or at least make sense at a higher level… If defined at multiple levels: qualify or override?

1.1.4 SOAP Handler versus Handler

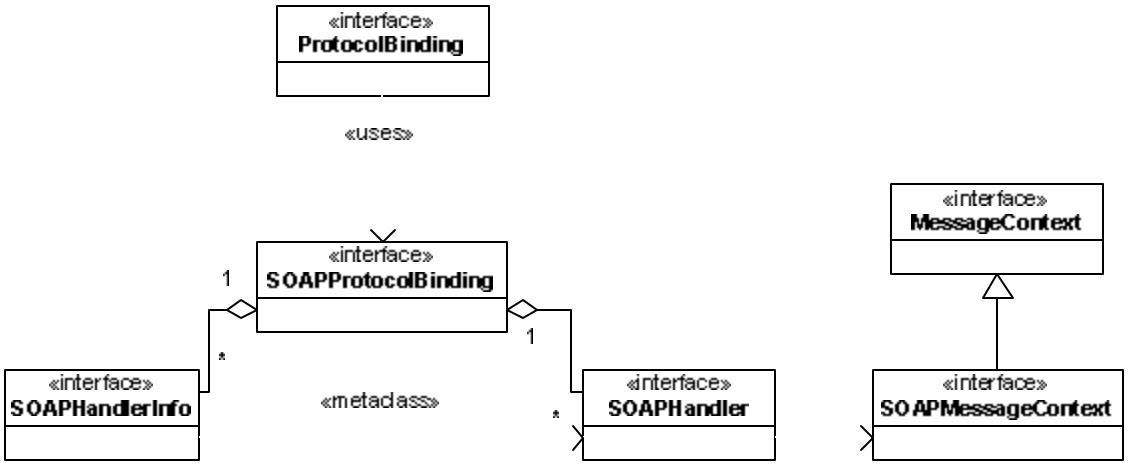
JAX-RPC 1.x waffles on the issue of defining agnostic handlers versus SOAP handlers. It wavers toward the "general notion of a handler" (MessageContext vs. SOAPMessageContext), implies that handlers for other protocols may be introduced someday, then hardens everything related to handlers into the SOAP world.

Two approaches (described API's are based on 1, but 2 preserves some original ideas started in JAX-RPC 1.x):

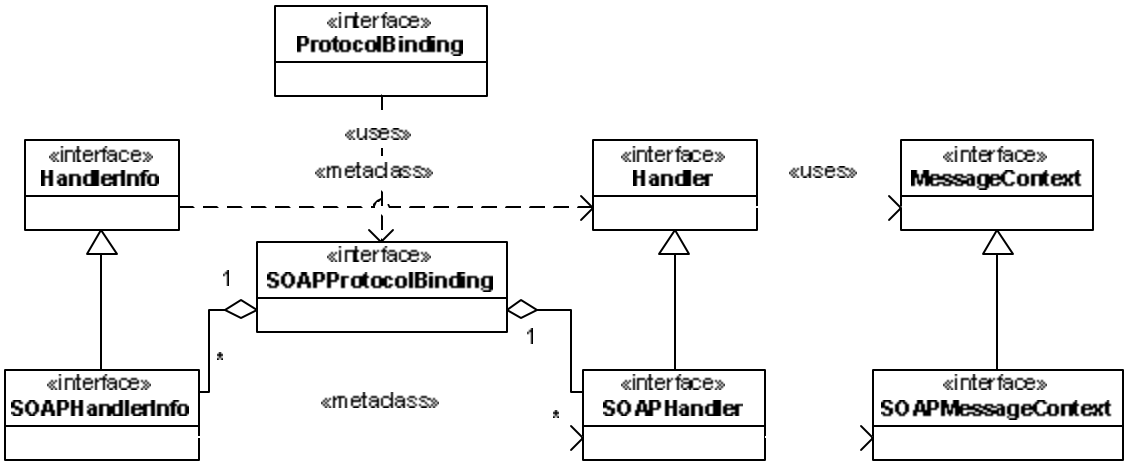
1) acknowledge that everything in handlers IS SOAP (rename and/or repackage handlers in javax.xml.rpc.soap.handlers.*), or



or



2) introduce a higher level abstraction:




```
                              QName[] headers) { ... }

    public void  setHandlerClass(Class handlerClass) { ... }
    public Class getHandlerClass() { ... }

    public void setHandlerConfig(java.util.Map config) { ... }
    public java.util.Map getHandlerConfig() { ... }

    public QName[] getHeaders() { ... }
    public void    setHeaders(QName[] headers) { ... }
}
```

```
package javax.xml.rpc.handler;

import javax.xml.rpc.MessageContext;

public interface Handler {
    boolean handleRequest(MessageContext context);
    boolean handleResponse(MessageContext context);
    boolean handleFault(MessageContext context);
    void    handleClosure(MessageContext context);

    QName[] getHeaders();

    // Life-cycle methods
    void    init(HandlerInfo config);
    void    destroy();
}
```

1.1.5 ?Handlers to be able to locate headers that match the nodes actors?

??? Did we catch this in the APIs?

From one perspective, if would be appropriate to claim that handlers should only be able to process headers that match the nodes actors. This is impractical, and inappropriate for some situations (logging).

However, we do see it as necessary that the handlers be able to ID the headers that correspond to the nodes actors/roles, and leave alone other headers.

1.1.6 Change Packages

MessageContext and SOAPMessageContext are not unique to handlers. They are exposed to target endpoints.

Move MessageContext from javax.xml.rpc.handler to javax.xml.rpc.

Move SOAPMessageContext from javax.xml.rpc.handler.soap to javax.xml.rpc.soap.

From perspective of "handlers are SOAP artifacts", move javax.xml.rpc.handlers.* into javax.xml.rpc.soap.handlers. If we end up introducing a protocol agnostic handler (to which a multitude of directions can be pursued), then it might make sense to introduce a parent handler in javax.xml.rpc.handlers.*.

2 Protocol Bindings

The JAX-RPC runtime is intended to be protocol neutral, supporting multiple messaging protocols. It is therefore necessary that a protocol can be configured and bound into the JAX-RPC runtime.

This chapter specifies the requirements and APIs for binding a protocol to a port. The SOAP bindings are introduced, enabling the JAX-RPC runtime to act according to SOAP semantics as defined by the SOAP specification(s).

2.1 Client-side Configuration

A developer must configure the client-side protocol binding, by port, prior to the runtime creation of a `Call` object or an endpoint proxy. Configuration may be completed using either a deployment model or via the API's.

2.1.1 Deployment Model

The JAX-RPC specification does not specify a standard deployment or packaging model for the client-side protocol bindings. A model may be defined by other specifications, as appropriate[4]. A client-side deployment model must clearly map[5] to the logical model presented for each protocol binding defined by this specification.

2.1.2 Configuration API's

A client-side developer performs the programmatic configuration of client-side protocol bindings using the `javax.xml.rpc.ProtocolBindings` interface. Access to a `ProtocolBindings` instance is provided by the `Service` interface:

```
package javax.xml.rpc;
public interface Service {
    ...
    /**
     * Returns the <code>ProtocolBindings</code> instance for
     * this <code>Service</code> instance.
     *
     * @return ProtocolBindings
     * @throws java.lang.UnsupportedOperationException - if the
     *         <code>Service</code> class does not support the
     *         configuration of <code>ProtocolBindings</code>.
     */
    ProtocolBindings getProtocolBindings();

    HandlerRegistry getHandlerRegistry();
}
```

[4] Such a model is defined as part of the J2EE 1.4 specifications [3].

[5] map directly?

2.2 Server-side Configuration

A developer must configure the server-side protocol binding, by port, prior to the runtime creation of the target end-point. Configuration may be completed using either a deployment model or via the API's.

Server-side configuration is not defined by JAX-RPC[6].

2.2.1 Deployment Model

The JAX-RPC specification does not specify a standard deployment or packaging model for the server-side protocol bindings. A model may be defined by other specifications, as appropriate[7]. A server-side deployment model must clearly map (?map directly) to the logical model presented for each protocol binding defined by this specification.

2.2.2 Configuration API's

The JAX-RPC specification does not specify a standard API for the server-side protocol bindings. APIs may be defined by other specifications, as appropriate. A server-side API must expose the `ProtocolBindings` for a port, and otherwise adhere to the programming model as presented.

2.3 ProtocolBindings

A JAX-RPC compliant runtime system is required to provide an implementation class for the `ProtocolBindings` interface[8].

```
package javax.xml.rpc;
public interface ProtocolBindings extends Serializable {
    /**
     * Returns the <code>ProtocolBinding</code> instance for
     * a port bound to this <code>Service</code> instance.
     *
     * @return ProtocolBinding
     */
    ProtocolBinding getProtocolBinding(QName portName);

    /**
     * Sets the <code>ProtocolBinding</code> instance for
     * a port bound to this <code>Service</code> instance.
     * The default is the SOAPProtocolBinding.
     */
    void setProtocolBinding(QName portName, ProtocolBinding binding)[9];
}
```

A `ProtocolBinding` is bound to each service endpoint, as indicated by the qualified name of a port.

---

[6] This is a gapping hole… yet it appears that I am not alone in NOT wanting to introduce a deployment model here.

[7] Such a model is defined as part of the J2EE 1.4 specifications [3].

[8] J2EE 1.4 is an extension to JAX-RPC, and is therefore not strictly "JAX-RPC compliant" in and of itself. A J2EE 1.4 compliant runtime provides a declarative deployment model, and prevents programmatic access by throwing the java.lang.UnsupportedOperationException exception if an attempt is made to access the ProtocolBindings.

[9] Is setProtocolBinding appropriate? WSDL specifies binding! Eliminate this intermediate ProtocolBindings

2.4 ProtocolBinding

```
package javax.xml.rpc;
public interface ProtocolBinding {
}
```

`ProtocolBinding` is an abstraction for configurable bindings supported by the JAX-RPC runtime implementation.

2.4.1 Configuration / Developer Responsibilities

A developer must perform programmatic configuration of the protocol binding, for each port, prior to the runtime creation of a `Call` object or an endpoint proxy. Configurable attributes are defined by specific implementations of `ProtocolBinding`.

2.4.2 JAX-RPC Runtime System Responsibilities

A JAX-RPC runtime system implementation is required to:

- Provide an implementation class for the `SOAPProtocolBinding` interface.
- When an instance of an implementation of `ProtocolBinding`[10] is set on a given port, that port must act according to the protocol's semantics.
- Throw a `JAXRPCException` when a `ProtocolBinding` not recognized or supported by a JAX-RPC runtime implementation is set onto a port. An implementation is NOT required to support arbitrary protocols from 3rd parties.

[10] Such as `SOAPProtocolBinding`.

3 SOAP Protocol Binding

This chapter specifies the requirements and APIs for using and executing the SOAP protocol in the JAX-RPC runtime. Interface extensions and semantics are defined to fully support processing of SOAP messages as specified by the SOAP specification [4].

3.1 SOAPProtocolBinding

An implementation class for the `SOAPProtocolBinding` interface provides a point for configuring SOAP behavior, and is an abstraction for the policy and mechanisms of the SOAP semantics specified in the remainder of this section.

```
package javax.xml.rpc.soap;

import java.util.List;
import javax.xml.rpc.ProtocolBinding;


public interface SOAPProtocolBinding extends ProtocolBinding {
    public void setRoles(String[] soapActorURIs);
    public String[] getRoles();

    List getHandlerInfos();
    void setHandlerInfos(List handlerInfos);
}
```

3.1.1 Configuration / Developer Responsibilities

The API's only provide a method for configuring client-side behavior. A developer may perform programmatic configuration of the:

- SOAP actor roles; specified as `URIs`.
- `List` of `SOAPHandlerInfo`'s.

3.1.2 Implementation Requirements

An implementation of `SOAPProtocolBinding` must:

- provide a default constructor.[11]
- In addition to the configured roles, ensure that (see section 3.2.3):
  - the service endpoint always acts in the role of the "ultimate destination"[12].
  - handlers never act in the role of the "ultimate destination".
  - the service endpoint and handlers always act in the role `next`[13].

[11] Do we need a factory?

[12] The default SOAP actor (actor not specified) indicates that the header is intended for the "ultimate destination".

[13] The `next` SOAP actor name is defined by the SOAP specification [4].

- A `JAXRPCException` must be thrown if an attempt is made to add an object other than `SOAP`**`HandlerInfo`** to the `List` returned by **`getHandlerInfos`**.
- A `JAXRPCException` must be thrown if an attempt is made to set a `List`, using **`setHandlerInfos`**, containing an object other than `SOAP`**`HandlerInfo`**.

3.2 JAX-RPC Runtime System Requirements

3.2.1 Client-Side Message Flow

When a JAX-RPC SOAP invocation is performed on a client:

- A `SOAPMessageContext` is obtained and associated to the JAX-RPC message.
- A SOAP request message is constructed and bound to the `SOAPMessageContext`.
- SOAP Handler `handleRequest` methods are invoked with the `SOAPMessageContext`.
- The SOAP request is marshaled and transmitted to a remote service.
- If a response is expected:
  - A SOAP response (normal or fault) is received from the remote service, de-marshaled, and bound to the `MessageContext`.
  - SOAP roles and "mustUnderstand headers must be checked, as-per section 3.2.3.
  - SOAP Handler `handleResponse` or `handleFault` methods are invoked with the `SOAPMessageContext`.
  - The response, or exception represented by the response, is delivered to the caller.
- If a response is not expected (one-way message):
  - SOAP Handler `handleClosure` methods are invoked with the `SOAPMessageContext`.
- The `SOAPMessageContext` is released.
- If processing is unsuccessful, exactly one SOAP fault is generated by this port, either by handlers or by the JAX-RPC runtime system. This SOAP fault is propagated to the client instead of the response SOAP message.

3.2.2 Server-Side Message Flow

When a JAX-RPC SOAP message is received:

- A `SOAPMessageContext` is obtained and associated with the JAX-RPC message.
- The SOAP request is received, de-marshaled, and bound to the `SOAPMessageContext`.
- SOAP roles and "mustUnderstand headers must be checked, as-per section 3.2.3.
- SOAP Handler `handleRequest` methods are invoked with the `SOAPMessageContext`.
- Invoke service-endpoint.
- If a response is expected:
  - A response (normal or fault) is obtained from the service endpoint and bound to the `SOAPMessageContext`.
  - SOAP Handler `handleResponse` or `handleFault` methods are invoked with the `SOAPMessageContext`.
  - The response or exception is marshaled and returned.
- If a response is not expected (one-way message):
  - SOAP Handler `handleClosure` methods are invoked with the `SOAPMessageContext`.
- The `SOAPMessageContext` is released.
- If processing is unsuccessful, exactly one SOAP fault is generated by this port, either by handlers or by the JAX-RPC runtime system. This SOAP fault is propagated to the client instead of the response SOAP message.

3.2.3 Roles and mustUnderstand Headers

The SOAP protocol binding performs the following logical steps during the processing of a *received* SOAP message. A "received message" is a request message on a server, or a response message on a client[14] [15]. Refer to the SOAP specification [4] for the normative details on the SOAP message processing model; this section is not intended to supercede any requirement stated within the SOAP specification, but rather to outline how various pieces of information tie together to satisfy the SOAP requirements:

1. Identify the set of SOAP actors, `"PortActors"`, for this port as the set consisting of the value of the port's `SOAPProtocolBinding.getRoles()`, the SOAP actor `next`, and the default actor representing the "ultimate destination".
2. Bind to each actor `"a"` in `"PortActors"` the set of header element qualified names (`QNames`) `"Headers(a)"` that the port understands, in that role[16]:
    a. The set of header `QNames` corresponding to parameters mapped to the operation by the original WSDL.
    b. If an actor `"a"` is not the default actor ("ultimate destination"), then include in `"Headers(a)"` the set of header `QNames` formed by the union of all header `QNames` obtained from the `SOAPHandlerInfos` found in the `List` returned by `getHandlerInfos()`.[17]
3. Identify the set of header element `QNames` `"MustUnderstandHeaders"` in the received message that are targeted at this node. `"MustUnderstandHeaders"` is the set of header element `QNames` for which element the `actor` attribute value is in the set `"PortActors"` AND the `mustUnderstand` attribute value is `"1"`.
4. For each header `QName` `"h"` in `"MustUnderstandHeaders"`, let `"role"` be the value of the header's `actor` attribute. Then the header `"h"` **is understood** by the node if `"h"` is in the set `"Headers(role)"`.
5. If every header `QName` `"h"` in `"MustUnderstandHeaders"` is understood, then the node understands how to process the message. If one or more of the header `QNames` are not understood by this node, then the node does not understand how to process the message.
6. If the node does not understand how to process the message, then generate a single SOAP `MustUnderstand` fault. If such a fault is generated, any further processing is not done. The `MustUnderstand` fault is propagated to the client in both cases where `MustUnderstand` fault is generated on either the server side or client side as part of SOAP message processing.

---

[14] At some point, the idea of a intermediate node (SOAP provides such) should be discussed.

[15] As a TERMINAL node (SOAP node), should we reject ANY message containing a header marked mustUnderstand, if we cannot process the header – regardless of role? Currently we reject ONLY if the role is in getRoles().

[16] It is very important that we don't imply that a handler knows how to process a header bound to an actor just because the target-endpoint acts in that role… and visa-versa.

[17] Who OWNS the headers? HandlerInfo or the Handler instance? If the handler instance, then does it makes sense to let the handler implement getHeaders as a static, so we can obtain without instantiating a handler?

3.2.4 SOAP Handler Processing Model

JAX-RPC runtime support for the SOAP protocol delegates processing of SOAP messages to a chain of handlers. The chain of handlers is represented by the `List` of `SOAPHandlerInfos`. Each `SOAPHandlerInfo` object in the `List` describes a `SOAPHandler`.

The following does not preclude other types[18] of handlers from being developed. In particular, container-specific processing of messages is beyond the scope of this specification.

3.2.4.1 Order of Execution

Handlers are executed in order, as-per the order of the `SOAPHandlerInfo` instances that define the handlers.

Handlers are executed in-order for a request.

Handlers are executed in reverse-order for a response, fault, or closure.

Example

In this example, three `SOAPHandler` instances `H1`, `H2` and `H3` are registered (in this order) via appropriately configured `SOAPHandlerInfo` objects on the `SOAPProtocolBinding`. The invocation order for these handlers is as follows:

- `H1.handleRequest`
- `H2.handleRequest`
- `H3.handleRequest`

- `H3.handleResponse`
- `H2.handleResponse`
- `H1.handleResponse`

The implementation of the handle methods in a SOAP message handler may alter this invocation order. Refer to section 3.2.4.3, “SOAP Handler Invocation” for more details.

[18] Other types of handlers can be developed for SOAP message processing specific to a JAX-RPC runtime system and corresponding containers. For example: stream based handlers, post-binding typed handlers.

3.2.4.2 SOAP Message Context

If the RPC call bound to a `MessageContext` is represented by a SOAP message, then the `MessageContext` shall be an instance of `SOAPMessageContext`[19].

The `handleRequest`, `handleResponse` and `handleFault` methods for a SOAP message handler get access to the `SOAPMessage` from the `SOAPMessageContext`. The implementation of these methods can modify the `SOAPMessage` including the headers and body elements.

Example: SOAP Message Handler

The following shows an example of a SOAP message handler.

```
package com.example;
public class MySOAPMessageHandler
        extends javax.xml.rpc.soap.handler.GenericSOAPHandler {

    public MySOAPMessageHandler() { ... }

    public boolean handleRequest(SOAPMessageContext ~~context~~smc) {
        try {
            ~~SOAPMessageContext smc = (SOAPMessageContext)context;~~
            SOAPMessage msg = smc.getMessage();
            SOAPPart sp = msg.getSOAPPart();
            SOAPEnvelope se = sp.getEnvelope();
            SOAPHeader sh = se.getHeader();
            // Process one or more header blocks
            // ...
            // Next step based on the processing model for this
            // handler
        }
        catch(Exception ex) {
            // throw exception
        }
    }

    // Other methods: handleResponse, handleFault init, destroy
}
```

[19] !!!If we leave the "new" interface with SOAPMessageContext, it would be unnecessary to state this as a requirement.

3.2.4.3 SOAP Handler Invocation

!!! The wording of the following sections has been changed significantly. Changes are significant enough that the following was difficult to read with change-bars. Please compare with originals if you are interested in this detail. The goal was clearer wording, with incremental (?) improvements… not wholesale redesign, with ONE exception.

The handler flow has been changed to fit the following rules:

- One of `handleResponse`, `handleFault`, or `handleClosure` will be called if and only if `handleRequest` returns true.
- At most one of `handleResponse`, `handleFault`, or `handleClosure` will be invoked on a handler for a given message.

This more easily allows a handler chain to function in-place of a handler.

??? Would propose that we follow JSR-109's lead, and restrict what a handler can do to a message.

3.2.4.3.1 handleRequest

The `handleRequest` method performs one of the following steps after ~~performing handler specific~~ processing ~~of~~ the request SOAP message:

- Return `true` to indicate continued processing of the request handler chain[20]. The runtime invokes the next entity. The next entity may be the next handler in the handler chain, or if this handler is the last handler in the chain, the next entity is the target service endpoint. The mechanism for dispatch or invocation of the target service endpoint depends on whether the request is on the client side or service endpoint side.
- Return `false` to indicate blocking of the request handler chain. The `SOAPHandler` implementation class is responsible for setting the response SOAP message in the `handleRequest` method before returning `false`[21]. Further processing of the request handler chain is blocked, and the target service endpoint is not dispatched. The runtime invokes ~~the~~ `handleResponse` methods going backwards through the handler chain ~~with the same `SOAPMessageContext`,~~ starting with the ~~same~~ `SOAPHandler` instance preceding the handler ~~(~~that returned `false`~~) and going backwards through the handler chain~~.
- Throw ~~the~~ `javax.xml.rpc.soap.SOAPFaultException` to indicate a SOAP fault. This is valid only for a server-side request handler; a client-side request handler's `handleRequest` method should never throw `SOAPFaultException`. Further processing of the request handler chain is blocked, and the target service endpoint is not dispatched. The runtime checks the message content, ~~and~~ if the message does NOT (already) represent a SOAP fault[22]~~,~~ then the runtime creates a response from the `SOAPFaultException`[23]. The runtime then invokes the `handleFault` methods going backwards through the handler chain ~~with the same `SOAPMessageContext`,~~ starting with the ~~same~~ `SOAPHandler` instance preceding the handler ~~(~~that threw the exception~~) and going backwards through the handler chain~~. Refer to the SOAP specification for details on the various SOAP `faultcode` values and corresponding specification. ??? How is this delivered to the client? Particularly if it does not map to an application exception?

[20] What if they "change" the message and return true? The easy answer that the message set is the "new" request message. In 109 we restricted what the handlers could DO to a request message.

[21] ???What happens if the handler does NOT do this? If we are to take a position, we must define the consequences.

[22] Set by handleResponse.

[23] This gives the handleRequest method an override? Is this good? what scenario is there for creating one SOAP fault and throwing a different SOAPFaultException? We could defer this logic until AFTER the handleFault (on the throwing handler) executes.. but no later.

- Throw `RuntimeException` or a subtype of `RuntimeException` (including `JAXRPCException`). Further processing of the request handler chain is blocked, and the target service endpoint is not dispatched. On the server side, the ~~`HandlerChain`~~ runtime generates a SOAP fault that indicates that "the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to a runtime error during the processing of the message"[24]. On the client side???. The runtime invokes the `handleClosure` methods going backwards through the handler chain starting with the `SOAPHandler` instance preceding the handler that threw the exception. Refer to the SOAP specification for details on the various SOAP `faultcode` values. ~~On the client side, t~~The `RuntimeException` ~~`JAXRPCException` or runtime exception~~ is propagated to the client code as a `RemoteException` or a subtype of `RemoteException`~~its subtype~~.

[24] This means that the SOAP fault must not contain detail.

Example

The following shows an example of the SOAP fault processing. In this case, the request handler `H2` on the server side throws a `SOAPFaultException` in the `handleRequest` method:

- `H1.handleRequest`
- `H2.handleRequest` -> throws `SOAPFaultException`
- `H2.handleFault`
- `H1.handleFault`

3.2.4.3.2 handleResponse

The `handleResponse` method performs ~~the processing of the SOAP response message. It does~~ one of the following steps after ~~performing its handler specific~~ processing ~~of~~ the response SOAP message:

- Return `true` to indicate continued processing of the response handler chain. The `HandlerChain` invokes the `handleResponse` method on the next `SOAPHandler`, going backwards through ~~in~~ the handler chain.
- Return `false` to indicate blocking of the response handler chain. Further processing of the response handler chain is blocked~~In this case, no other response handlers in the handler chain are invoked~~. ~~On the service endpoint side, this may be useful if response handler chooses to issue a response directly without requiring other response handlers to be invoked~~The runtime invokes the `handleClosure` methods going backwards through the handler chain starting with the `SOAPHandler` instance preceding the handler that returned `false`.
- Throw `RuntimeException` or a subtype of `RuntimeException` (including `JAXRPCException`).~~Throw the `JAXRPCException` or any other `RuntimeException` for any handler specific runtime error~~. Further processing of the response handler chain is blocked.~~If `JAXRPCException` is thrown by the `handleResponse` method, the `HandlerChain` terminates the further processing of this handler chain.~~ On the server side, the runtime ~~`HandlerChain`~~ generates a SOAP fault that indicates that "the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to a runtime error during the processing of the message"[25]. On the client side???. The runtime invokes the `handleClosure` methods going backwards through the handler chain starting with the `SOAPHandler` instance preceding the handler that threw the exception. Refer to the SOAP specification for details on the various SOAP `faultcode` values. The `RuntimeException` is propagated to the client code as a `RemoteException` or a subtype of `RemoteException`~~On the client side, the `JAXRPCException` or runtime exception is propagated to the client code as a `RemoteException` or its subtype~~.

[25] This means that the SOAP fault must not contain detail.

3.2.4.3.3 handleFault

~~The `handleFault` method performs the SOAP fault related processing.~~ The JAX-RPC runtime system ~~should~~ invoke<u>s</u> the `handleFault` method if a SOAP fault needs to be processed by either client-side or server-side handlers[26]. The `handleFault` method <u>performs</u> ~~does~~ one of the following steps after ~~performing handler specific~~ processing ~~of~~ the SOAP fault <u>message</u>:

- Return `true` to indicate continued processing of the fault ~~handlers in the~~ handler chain. The `HandlerChain` invokes the `handleFault` method on the next `SOAPHandler`<u>, going backwards through</u> ~~in~~ the handler chain.
- Return `false` to indicate blocking of the fault ~~processing in the~~ handler chain. <u>Further processing of the fault handler chain is blocked</u>~~In this case, no other handlers in the handler chain are invoked. The JAX-RPC runtime system takes the further responsibility of processing the SOAP message~~. <u>The runtime invokes the `handleClosure` methods going backwards through the handler chain starting with the `SOAPHandler` instance preceding the handler that returned `false`.</u>
- <u>Throw `RuntimeException` or a subtype of `RuntimeException` (including `JAXRPCException`). Further processing of the response handler chain is blocked</u>~~Throw `JAXRPCException` or any other `RuntimeException` for any handler specific runtime error. If `JAXRPCException` is thrown by the `handleFault` method, the `HandlerChain` terminates the further processing of this handler chain~~. On the server side, the <u>runtime</u> ~~`HandlerChain`~~ generates a SOAP fault that indicates that <u>"</u>the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to a runtime error during the processing of the message<u>"</u> [27]. <u>On the client side???. The runtime invokes the `handleClosure` methods going backwards through the handler chain starting with the `SOAPHandler` instance preceding the handler that threw the exception. Refer to the SOAP specification for details on the various SOAP `faultcode` values. The `RuntimeException` is propagated to the client code as a `RemoteException` or a subtype of `RemoteException`</u>~~On the client side, the `JAXRPCException` or runtime exception is propagated to the client code as a `RemoteException` or its subtype~~.

Please note that when a `JAXRPCException` or `RuntimeException` raised on the server is converted to a SOAP fault for the purpose of being transmitted to the client, there are no guarantees that any of the information it contains will be preserved.

[26] <u>This means that if a handleResponse CHANGES a (normal) response to a fault, then the next handler will be invoked via handleFault.</u>

[27] This means that the SOAP fault must not contain detail.

3.2.4.3.4 handleClosure

The `handleClosure` method is invoked in all situations where the `handleResponse` and `handleFault` methods would NOT otherwise be invoked. Examples include one-way messages, or when a `JAXRPCException` is thrown by the `handleFault` method.

3.2.4.4 Exceptions and SOAP Handlers

When a `JAXRPCException` or `RuntimeException` raised on the server is converted to a SOAP fault for the purpose of being transmitted to the client, there are no guarantees that any of the information it contains will be preserved.

Other clarifications related to handling SOAPFaultException?

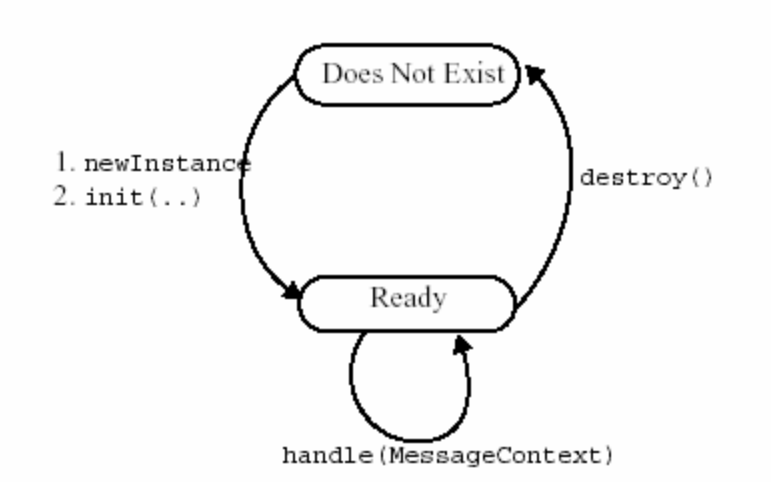
- If it matches an application fault, then . . .
- If it does not match an application fault?

3.2.5 SOAP Handler Lifecycle

The JAX-RPC runtime system is required to manage the lifecycle of `SOAPHandler` instances by invoking the `init` and `destroy` methods.

```
package javax.xml.rpc.soap.handler;
public interface SOAPHandler {
    void init(SOAPHandlerInfo config);
    void destroy();
    // ...
}
```

The following state transition diagram shows the lifecycle of a `SOAPHandler` instance:



The JAX-RPC runtime system is responsible for loading the `SOAPHandler` class and instantiating the corresponding `SOAPHandler` object. The lifecycle of a `SOAPHandler` instance begins when the JAX-RPC runtime system creates a new instance of the `SOAPHandler` class.

After a `SOAPHandler` is instantiated, the JAX-RPC runtime system is required to initialize the `SOAPHandler` before this `SOAPHandler` instance can start processing SOAP messages. The JAX-RPC runtime system invokes the `init` method to enable the `SOAPHandler` instance to initialize itself. The `init` method is ~~passes~~ passed the handler configuration as a `SOAPHandlerInfo` instance. ~~The `HandlerInfo` is used to configure the `Handler` (for example: setup access to an external resource or service) during the initialization. The `init` method also associates a `Handler` instance (using the `HandlerInfo`) with zero or more header blocks using the corresponding `QNames`.~~

~~In the `init` method, the `Handler` class may get access to any resources (for example; access to an authentication service, logging service) and maintain these as part of its instance variables. Note that these instance variables must not have any state specific to the SOAP message processing performed in the various `handle` method.~~

Once the `SOAPHandler` instance is created and initialized it is placed into ~~(and is in~~ the `Ready` state~~),~~. ~~the~~ In the `Ready` state the JAX-RPC runtime system may invoke the different `handle` method~~s~~ multiple times, and in parallel~~.~~

The JAX-RPC runtime system is required to invoke the `destroy` method when the runtime system determines that the `SOAPHandler` object is no longer needed. For example, the JAX-RPC runtime may remove a `SOAPHandler` object when the runtime system (in a~~n~~ managed operational environment) is shutting down or managing memory resources.

The `destroy` method indicates the end of lifecycle for a `SOAPHandler` instance. The `SOAPHandler` instance must release~~s~~ its resources and perform~~s~~ cleanup in the implementation of the `destroy` method. After ~~successful~~ invocation of the `destroy` method, the `SOAPHandler` object is available for the garbage collection.

A `RuntimeException` (other than `SOAPFaultException`) thrown from any `handle` method of the `SOAPHandler` results in the `destroy` method being invoked and transition to the "`Does Not Exist`" state.

3.2.6 Threading

The runtime system must be capable of processing multiple messages simultaneously on a given handler chain.

3.3 Developer Responsibilities in Writing a SOAP Handler

3.3.1 Interface

A SOAP handler must implement the `javax.xml.rpc.soap.handler.SOAPHandler` interface (section 3.4.2).

3.3.2 Constructor

A `SOAPHandler` must provide a default constructor. Initialization of the handler should be deferred until the `init()` method is invoked by the runtime.

3.3.3 SOAP Header Elements

??? Clean up redundancies with life-cycle, below, and cross-reference as appropriate between the two.

A `SOAPHandler` must declare the set of SOAP header elements (`QNames`) it understands, and must limit processing of header elements to that set.

??? There appear to be many options here, some relevant to spec evolution: a) handler can hard-code header elements and return via getHeaders (remove config from `SOAP`HandlerInfo?), b) handler can accept configurable list of header elements (`SOAP`HandlerInfo) – and MUST be written accordingly (has ANYONE ever has written such a handler?), c) we can refactor the current getHeaders method to be a 'static' method. Any relevant or foreseeable scenarios need to be recorded, even if just a footnote.

A SOAP message `SOAPHandler` instance is associated with SOAP header blocks using the qualified name of the outermost element of each header block. A `SOAPHandler` indicates that it would process specific header blocks through this association. The following method initializes a `SOAPHandler` instance for this association:

```
package javax.xml.rpc.soap.handler;
public interface SOAPHandler {
    void init(SOAPHandlerInfo config);
    // ...
}
```

3.3.4 SOAP Actor Roles

A `SOAPHandler` must limit processing of header elements to those header elements whose actor attribute values appear in the list returned by `SOAPMessageContext.getRoles()`.

A SOAP message `SOAPHandler` instance gets access to the SOAP Actor roles bound to a port through the `SOAPMessageContext.getRoles()` method. ~~A `Handler` instance uses this information about the SOAP Actor roles to process the SOAP header blocks.~~ Note that the SOAP actor roles cannot be changed on a port during the processing of a SOAP message.

3.3.5 State

A SOAP message handler is required to be implemented as a stateless instance. A SOAP message handler must not maintain any SOAP message related state in its instance variables across multiple invocations of the `handleRequest()`, `handleResponse()`, `handleFault()`, and `handleClosure()` methods.

In terms of the SOAP message processing functionality, the JAX-RPC runtime system considers all instances of a specific handler class as equivalent: the runtime may choose any ready instance of a `SOAPHandler` class on which to invoke the handle methods[28].

3.3.6 Life-Cycle

A SOAP message handler is required to implement the following lifecycle methods of the `javax.xml.rpc.soap.handler.SOAPHandler` interface:

```
package javax.xml.rpc.soap.handler;
public interface SOAPHandler {
    void init(SOAPHandlerInfo config);
    void destroy();
    // ...
}
```

Review the life-cycle responsibilities of the runtime (section 3.2.5). The `SOAPHandlerInfo` is passed to the `init` method to configure the `SOAPHandler` during initialization. ~~For example: setup access to an external resource or service.~~ The `init` method also associates a `SOAPHandler` instance (using the `SOAPHandlerInfo`) with zero or more header blocks using the corresponding `QNames`.

In the `init` method, the `SOAPHandler` class may get access to external resources or services (for example; access to an authentication service, logging service) and maintain these as part of its instance variables.

3.3.7 Threading

Handlers should make NO assumptions about the thread of execution

- Between two handlers on same handler chain, for a given MessageContext
- Between invocation of `handleRequest` and `handleResponse/Failure/Closure` for a given MessageContext.

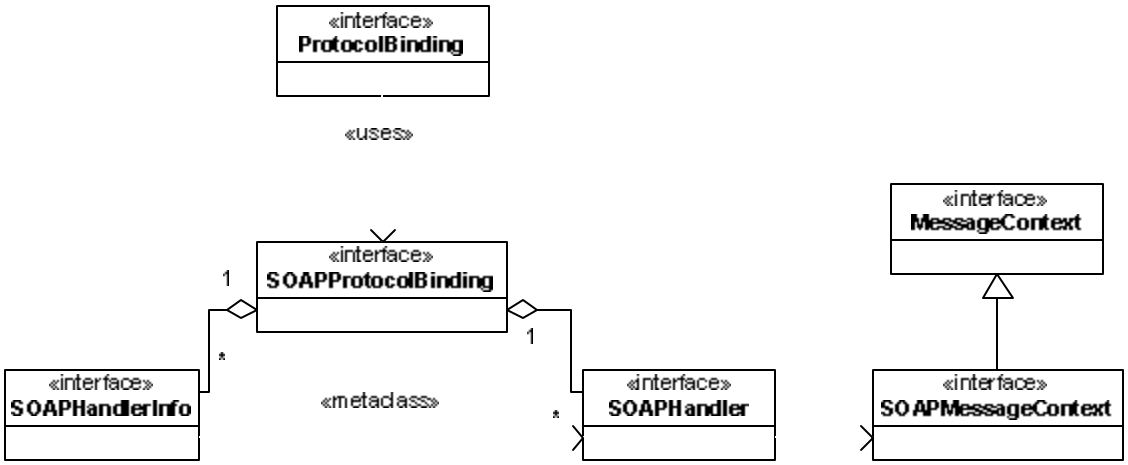
3.3.8 Processing

The handler SHOULD remove header blocks, processed by this handler, that are marked mustUnderstand. There are exceptions to this due to existing/alternate specs. In the absence of such justifications, this SHOULD be done. ?? Runtime to enforce or automate?

[28] This makes the `Handler` instances capable of being pooled by the JAX-RPC runtime system per deployed endpoint component. However, JAX-RPC runtime system is not required to support pooling of `Handler` instances.

3.4 Additional JAX-RPC SOAP Protocol APIs

The following diagram shows the class diagram for the handler APIs.



SOAPHandlerInfo

The `SOAPHandlerInfo` class represents the configuration data for a `SOAPHandler`. A `SOAPHandlerInfo` instance is passed in the `SOAPHandler.init` method to initialize a `SOAPHandler` instance.

```
package javax.xml.rpc.soap.handler;
public class SOAPHandlerInfo implements java.io.Serializable {
    public SOAPHandlerInfo() { }

    public SOAPHandlerInfo(Class handlerClass,
                           java.util.Map config,
                           QName[] headers) { ... }

    public void  setHandlerClass(Class handlerClass) { ... }
    public Class getHandlerClass() { ... }

    public void setHandlerConfig(java.util.Map config) { ... }
    public java.util.Map getHandlerConfig() { ... }

    public QName[] getHeaders() { ... }
    public void    setHeaders(QName[] headers) { ... }
}
```

3.4.2 SOAPHandler

```
package javax.xml.rpc.soap.handler;

import javax.xml.rpc.soap.SOAPMessageContext;

public interface SOAPHandler {
    boolean handleRequest(SOAPMessageContext context);

    boolean handleResponse(SOAPMessageContext context);
    boolean handleFault(SOAPMessageContext context);
    void    handleClosure(SOAPMessageContext context);

    QName[] getHeaders();

    // Life-cycle methods
    void    init(SOAPHandlerInfo config);
    void    destroy();
}
```

3.4.3 GenericSOAPHandler

The `javax.xml.rpc.soap.handler.GenericSOAPHandler` class is a convenience abstract class that implements the `SOAPHandler` interface. Handler developers may subclass the `GenericSOAPHandler` class as an aid in writing a `SOAPHandler` implementation class. A handler developer need only override methods that need to be specialized for the derived `SOAPHandler` implementation class.

```
package javax.xml.rpc.soap.handler;

import javax.xml.rpc.SOAPMessageContext;

public abstract class GenericSOAPHandler implements SOAPHandler {
    protected SOAPHandlerInfo handlerInfo;

    protected GenericSOAPHandler() {}

    public boolean handleRequest(SOAPMessageContext context) {
        return true;
    }

    public boolean handleResponse(SOAPMessageContext context) {
        return true;
    }

    public boolean handleFault(SOAPMessageContext context) {
        return true;
    }

    public void    handleClosure(SOAPMessageContext context) {
    }

    public void    init(SOAPHandlerInfo info) {
        handlerInfo = info;
    }

    public void    destroy() {}

    public QName[] getHeaders() {
      return handlerInfo.getHeaders();
    }
}
```

3.4.4 MessageContext

The interface `MessageContext` abstracts the message context that is processed by a handler in the `handleRequest`, `handleResponse`, `handleFault`, or `handleClosure` methods.

```
package javax.xml.rpc;
public interface MessageContext {
    void    setProperty(String name, Object value);
    Object  getProperty(String name);

    void    removeProperty(String name);
    boolean containsProperty(String name);

    java.util.Iterator getPropertyNames();
}
```

The `MessageContext` interface provides methods to manage a property set. The `MessageContext` properties enable handlers in a handler chain to share processing related state. For example, a handler may use the `setProperty` method to set value for a specific property in the message context. One or more other handlers in the handler chain may use the `getProperty` method to get the value of the same property from the message context.

Please note that there is no specified relationship between the message context properties and either the Stub properties described in section 8.2.2, "Stub Configuration" or the Call properties described in section 8.2.4, "DII Call Interface".

The runtime is required to share the same `MessageContext` across `SOAPHandler` instances that are invoked during a single request and response or fault processing on a specific service endpoint.

`SOAPHandler` instances in a handler chain should not rely on the thread local state to share state between handler instances. Handler instances should use the `MessageContext` to share any SOAP message processing related state.

3.4.5 SOAPMessageContext

The interface `SOAPMessageContext` provides access to the SOAP message for either RPC request or response. The `javax.xml.soap.SOAPMessage` specifies the standard Java API for the representation of a SOAP 1.1 message with attachments. Refer to the JAXM specification [13] for more details on the `SOAPMessage` interface.

```
package javax.xml.rpc.soap;
public interface SOAPMessageContext extends MessageContext {
    SOAPMessage getMessage();
    void        setMessage(SOAPMessage message);

    /**
     * Return the actor/roles bound to the
     * port's SOAPProtocolBinding.
     */
    String[]    getRoles();
}
```

4 ~~SOAP~~ Message Handlers

Reasonable intro to handlers, but in context above, and in context of spec, is this appropriate?

4.1 Introduction

A JAX-RPC SOAP message handler extends the message processing facilities of the JAX-RPC runtime environment, into which it is deployed.

A typical use of a JAX-PRC SOAP message handler is to process SOAP header blocks as part of the processing of an RPC request or response.

Example

A few examples of handlers are:

- Encryption and decryption handler
- Logging and auditing handler
- Caching handler

Example

An example of a JAX-RPC SOAP message handler is:

A secure stock quote service requires that the SOAP body be encrypted and the SOAP message digitally signed to prevent unauthorized access or tampering with RPC requests or responses. SOAP message handlers are used to implement this:

- On the client side, a SOAP message handler is deployed on a port to encrypt content and digitally sign SOAP messages before they are sent to the remote service endpoint.
- On the server side, a SOAP message handler is deployed on a port/service to validate signed messages and decrypt content before the RPC request is dispatched to the target service endpoint implementation.
- The same steps are repeated in reverse for the RPC response carried in a SOAP message.