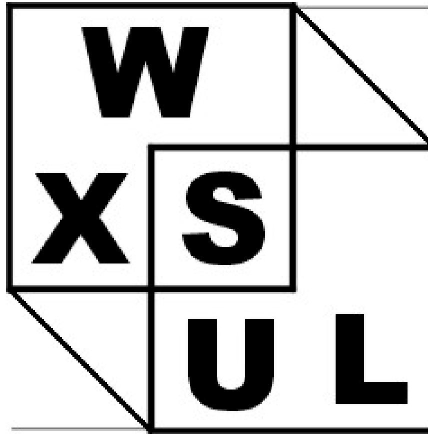


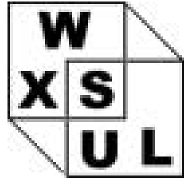
Brief Guide to Programming with Web/XML Services Utility Library

Aleksander Slominski

Dennis Gannon

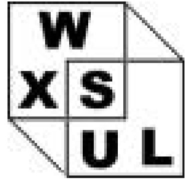
Liang Fang

2005-02-15



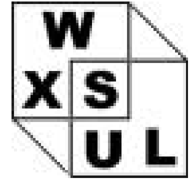
Standards and APIs Supported

- HTTP 1.0/1.1 and TLS/SSL
- One-way messaging and Request-response (RPC) message exchange patterns (over HTTP)
- SOAP 1.1/1.2 (only minimal support for SOAP-ENC)
- WSDL 1.1 doc/literal (minimally support rpc/encoded).
- WS-Addressing *and asynchronous messaging*
- Extended Apache WSIF API (complex types etc)
- XML Schemas (through XmlBeans)
- *Security (next slide)*



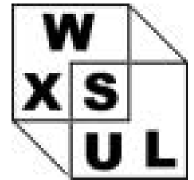
Security (Liang Fang)

- GSI Security (grid-proxy, myproxy)
- WS-Security and WS-SecureConversation
- Security based on capabilities model
  - XPola and CapMan



Programming Model

- WSDL with XML Schemas is a **contract**:
  - Generate binding Java classes from XML Schema
  - Generate Java interface from WSDL using `ant`:
    - <xwsdlc *[options]*> <fileset>WSDL files
- WSDL and Java Interface: cornerstones
  - Implement service: class implements Java Interface
  - Host service: class implementation and WSDL
  - Access service: use Java interface and WSDL



Service Development Steps

Obtain WSDL (registry, Google, etc)

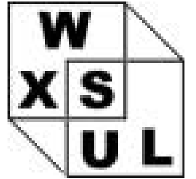
WSDL → XSUL Dynamic Invoker (Client side)

WSDL → WSDL compiler <xwsdlc> Decide type of Java interface to generate

MSG binding Java interface → Use Java interface and WSDL

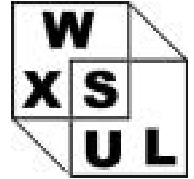
Client side → Access Service Described in WSDL using Java interface

Server side → Implement Java interface and host impl. Service, Publishes its WSDL



XML Schema Binding

- Use XmlBeans to generate Java classes that are encapsulating XML Schemas
- Use XmlBeans generated Java classes for parameters and to return response
  - XSUL automatically converts between XmlBeans XmlObject and internal representation used for sending (XmlElement)



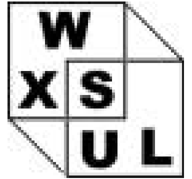
Generate Java Interface

- Generate different types of Java Interface from WSDL using ANT task called <xwsdlc>:
- `<xwsdlc`

  `wsdlgendir=`*"directory for java interfaces"*

  `destfile=`*"JAR file with XmlBeans classes"*`>`

  `<fileset>`*.wsdl and .xsdconfig files*`</></>`
- Generates source code for Java interface based on WSDL PortType(s)
  – All method parameters and return types are mapped to Java types generated by XmlBeans

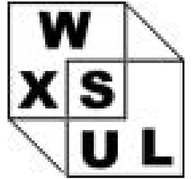


Using WS (Client Side APIs)

Java client code example: DecoderPortType is Java Interface generated from WSDL and dynamic stub is client based on service WSDL and Java Interface that is used to invoke “runDecoder” operations

```
DecoderPortType client = (DecoderPortType)
  XmlBeansWSIFRuntime
  .newClient(“http://somehost/decoder?wsdl”)
  .generateDynamicStub(DecoderPortType.class);

responseMsg = client.runDecoder(inputMsg);
```



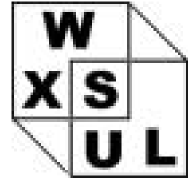
Using WS (Dynamic WSIF API)

Dynamic Invocation (No Code Generation, No Compilation):

`java xsul.dii.XsulDynamicInvoker`
WSDL portType parameters

Example:

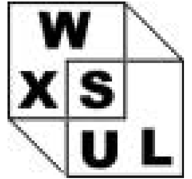
```
java xsul.dii.XsulDynamicInvoker
   http://somhost/decoder?wsdl runDecoder
   Topic File Dir 1
```



Providing WS (Server Side API)

Implement Java Interface: use *XmlBeans generated classes* that represent XML messages:

```
class DecoderImpl implements DecoderPortType {
  public RunDecoderResultDocument runDecoder( RunDecoderDocument input) {
    //extract parameters from input message
    DecoderParameters params = inputMsg.getRunDecoder();
    String topic = params.getTopic();
    String inputFile = params.getInputFile();
    String outputDirectory = params.getOutputDirectory();
    int nproc = params.getNproc();
    // do something with input
    logger.finest("got topic="+topic+" inputFile="+inputFile
        +" outputDirectory="+outputDirectory+" nproc="+nproc);
    // prepare response message
    RunDecoderResultDocument resultDoc =
    RunDecoderResultDocument.Factory.newInstance();
    DecoderResults result = resultDoc.addNewRunDecoderResult();
    result.setStatus("OK");
    return resultDoc;
  }
}
```



Providing WS (Server Side API)

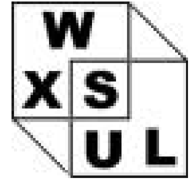
Make service available: service will implement contract described in WSDL (loaded from wsdlLoc) and implementation comes from DecoderImpl class:

```
HttpBasedServices httpServices = new
  HttpBasedServices(tcpPort);
XService xsvc = httpServices.addService(
    new XmlBeansBasedService(
        “decoder”, wsdlLoc, new
  DecoderImpl()));
xsvc.startService();
```

WSDL for service will be available at *http://host:port/**decoder**?wsdl*

Advanced Web Services

Handlers



XML Web Services Aspects

Transport (ex. HTTP)

Security (XML dsig.)

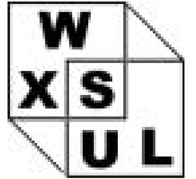
RPC style

WS/XML Service

Reliable Messaging

Transactions

Future? WS-Stuff

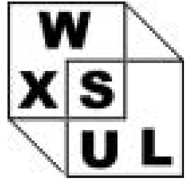


Handler

- **Handler does something with message context**)and handler has name)

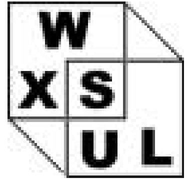
```
interface XHandler {
  boolean process(
     MessageContext context);
  String getName()
}
```

- If process() returns **true it indicates that processing of handler chain should be stopped** (see next slides)



Message Context

- Message Context is XML Document
  - XML is easy to share and send between processing nodes (inside one JVM, multiple JVMs or even hosts)
- MessageContext is XmlElement with:
  - `String getDirection()`
    - Returns MessageContext.DIR_INCOMING or MessageContext.DIR_OUTGOING
  - `XmlElement getIncomingMessage()`
  - `XmlElement getOutgoingMessage()`



Server Side Handlers

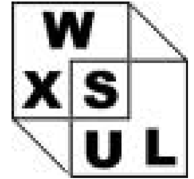
Adding handlers to service:

XService.addHandler(…)

Example:
```
xsvc.addHandler(
        new ServerSecConvHandler(
          "secconv-server"));
```

Server side handler can easily access and modify WSDL to declare supported policies and features (<feature … /> etc.)

In future handler deployment may be automated through auto-discovery



Client Side Handlers

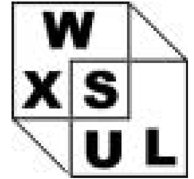
Adding handler to implement feature

WSIFClient.addHandler(…)

Example:
```
wcl.addHandler(
        new ClientSecConvHandler(
         "secconv-c", scsvcloc));
```

Client side handler will read WSDL and determine if it should be used and how depending on WSDL policies and features (presence of <feature … /> etc.)

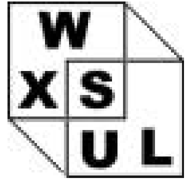
In future it may be automated through inspecting WSDL (policy and required features) and auto discovery of available handlers on client side.



Customized Server Side Handlers

Detailed control over how service chain is created and composed and executed (next slides):

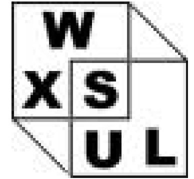
```
XServiceServo server = new HttpBasedServices(tcpPort);
XService xsvc = server.addService(
  new XmlBeansBasedService(“decoder”, wsdlLoc, new DecoderImpl());
 xsvc.addHandler(new ServerSignatureHandler("sig-server"))
   .addHandler(new ServerCapabilityHandler( "cap-server",
      "http://localhost:"+ httpServices.getServerPort()+ "/decoder”));
xsvc.addHandler(new ReliableMessagingHandler());
xsvc.startService();
```



Handler Chain

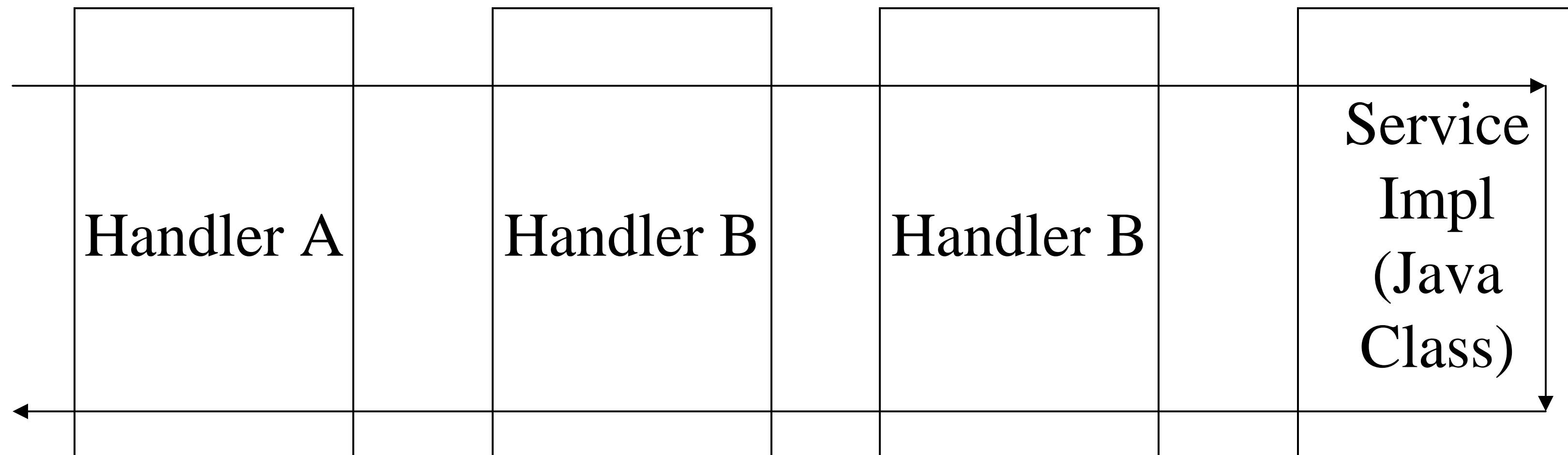
- Handlers are chained and executed in order they were added
- Server Side and client side are similar except for pivot.
- Example scenario:

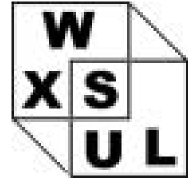
```
addHandler(handlerA)
addHandler(handlerB)
addHandler(handlerC)
useService(serviceImpl);
```



Typical Handler Chain Execution

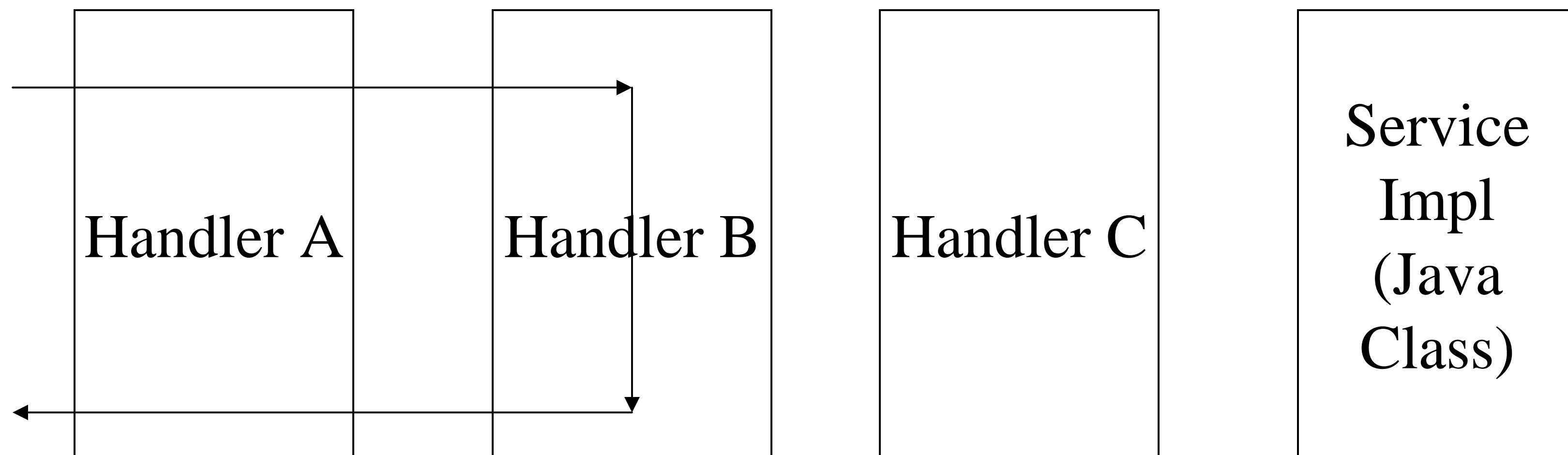
- Client side first processing outgoing message then incoming message and no need to create faults – exceptions are local
- Server side execution flow:

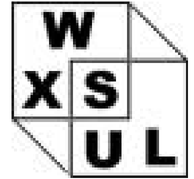




Fault Creation in Handler

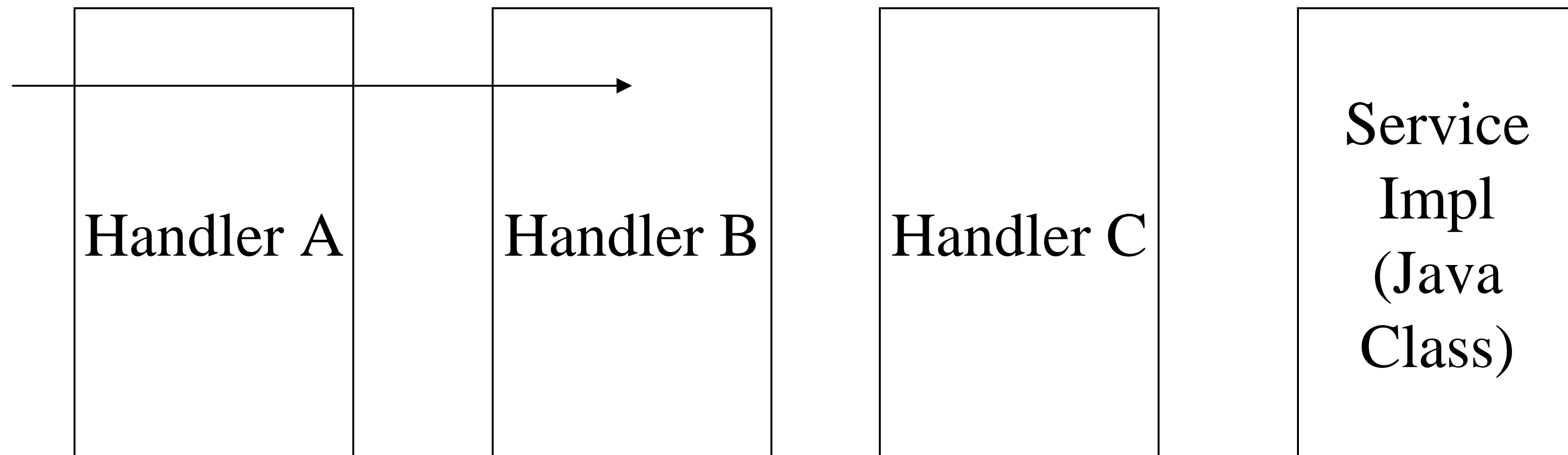
- Handler B determines based on incoming message that processing should not continue
  - Handler B creates (or modifies) outgoing message to put fault information
  - Handler B returns true to indicate that processing should finish
  - Already called handlers are called (Handler A)
- Handler C and service implementation is never called

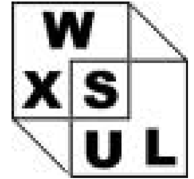




Exception in Handler

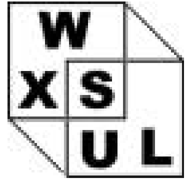
- Handler B throws runtime exception
- Exception is intercepted by engine
- Exception is converted to SOAP Fault
- Processing stops at that moment





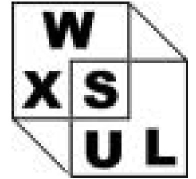
Conclusions

- WSDL is contract
  - Minimal amount of Java code generated
- Easy to invoke any WSDL described service
  - Web Services Invocation Framework (WSIF) supports multiple bindings
    - Currently only SOAP/HTTP
    - Possible JMS, local Java, …
- Easy to implement services
  - XmlBeans classes
- Easy to add aspects/handlers
  - *Driven by metadata in WSDL*



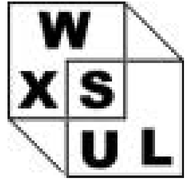
Future Is Asynchronous!

- API Improvements: asynchronous WS and support on client side
- Adding support for wsdl:fault for <xwsdlc>
- Handlers configuration and auto-discovery
  - Automatically deployed from Jar MANIFEST



Asynchronous means …

- *Asynchronous support on server side*
  - Built into web service hosting container (nothing needs to be done by service developer)
  - Correctly dealing with WS-Addressing headers
- *Asynchronous support on client side:*
  - Future result = service.operation(…)
  - Independent if target service supports WS-Addressing or not



Asynchronous Web Service

- XSUL mini HTTP server supports WS-Addressing and asynchronous responses
  - Nothing needs to be done!
- XSUL client need to declare that asynchronous invocation are to be used
  - Use <**`xwsdlc async`**> to compile WSDL
  - Generated method return value is **`Future<Type>`**
  - Client needs to run mini HTTP server or use WS-MsgBox to retrieve responses