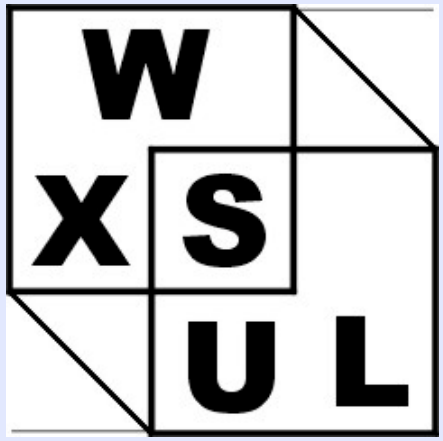
Building Web Services with XML Service Utility Library (XSUL)



Aleksander Slominski

IU Extreme! Lab

August 2005

Linked **E**nvironments for **A**tmospheric **D**iscovery

Outline

- Goals and Features
- Creating Web Services with XSUL
  - XML Schema, WSDL, <xwsdlc>, generated code: Xml Beans, Java Interface
  - Embedded HTTP Container: connections and threading
- Accessing Web Services with XSUL
  - Dynamic invocation, using generated Java Interface and XML Beans, asynchronous messaging
- Other Capabilities and Handlers
- Future work: Scalability and Reliability

Goals

- (No) One library to rule them all?
- Set of pieces that can be combined and recombined …
- XML Schema (XS) 1.0 Support
  - As complete as possible
- WSDL 1.1 doc/literal for SOAP 1.1/1.2
  - Messages described in XS
- Service composed out of messages: in and out
- Handlers: enable security, load balancing, etc.
- Implement and use services with minimal number of XSUL APIs
  - Or none at all: service implementation should deal with logic and be SOAP toolkit independent

XSUL
as easy as
LEGO

Features

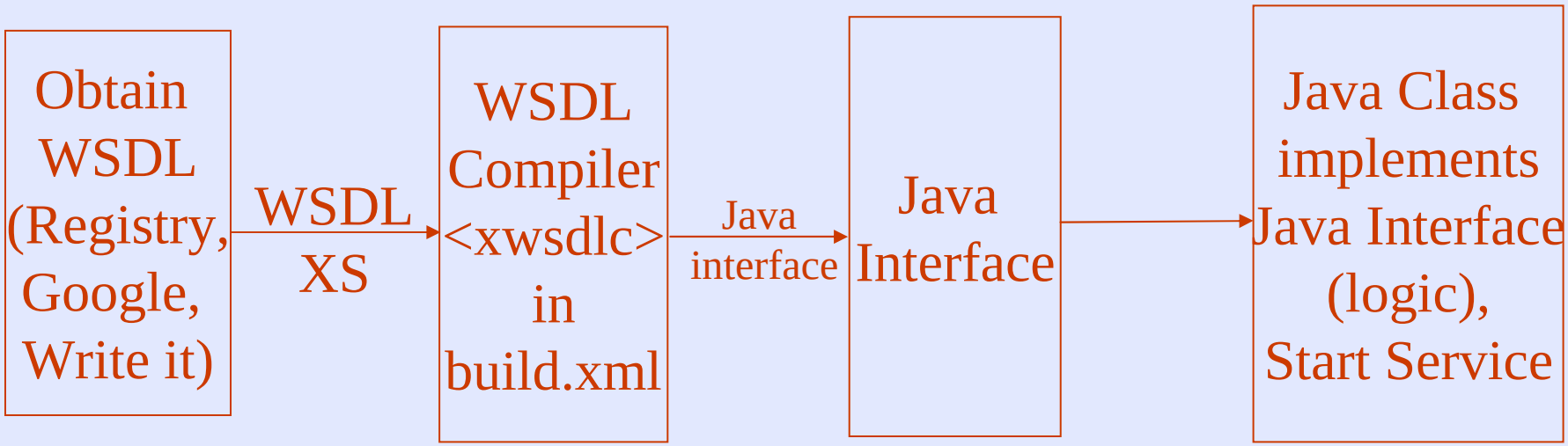
- HTTP 1.0/1.1
- One-way messaging and Request-response (RPC) message exchange patterns (over HTTP)
- SOAP 1.1/1.2 (only minimal support for SOAP-ENC)
- WSDL 1.1 doc/literal (minimally support rpc/encoded).
- WS-Addressing *and asynchronous messaging*
- Extended Apache WSIF API (complex types etc)
-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

Security

- This is one of the most important requirements for Grid Web Services
- TLS/SSL
- GSI Security (grid-proxy, myproxy)
- WS-Security and WS-SecureConversation
- Security based on capabilities model
  - XPola and CapMan
- Work on load balancing to boost time to process signed messages

Service Development

Typical steps involved



XML Schema in WSDL

- Apache XML Beans is used to generate Java classes that are encapsulating XML Schema
  - Apache XmlBeans has the most complete support for XML Schema in Java
- WSDL 1.1 can contain any schema but binding must be doc/literal
  - Very limited support for SOAP Encoding
- Ant task <xwsdlc> is used to process WSDL, call XML Beans code generator, and generate Java Interface
  - Details online in XSUL Guide (including example)

Example Schema (Guide)

```
<complexType name="DecoderParameters">
      <annotation><documentation xml:lang="en">
        Type of input message: sequence of parameters.
      </documentation></annotation>

      <sequence>
            <element minOccurs="1" maxOccurs="1" name="Topic" type="xsd:string"/>
            <element minOccurs="1" maxOccurs="1" name="CorrelationId" type="xsd:string"/>
            <element minOccurs="1" maxOccurs="1" name="InputFile" type="xsd:string"/>
            <element minOccurs="1" maxOccurs="1" name="OutputDirectory" type="xsd:string"/>
            <element minOccurs="0" maxOccurs="1" name="StringArr"
                        type="typens:ArrayOfString"/>

            <element name="nproc" type="xsd:int" minOccurs="0" maxOccurs="1" default="64" >
              <annotation><documentation xml:lang="en">
                                    Example parameter with default value.
              </documentation></annotation>
            </element>
      </sequence>
      <attribute name="SomeStringAttrib" type="string"/>
      <attribute name="SomeBoolAttrib" type="boolean"/>
</complexType>
```

Example WSDL

Scaffolding required to declare what is input message and what is output message for

```
<element name="Decoder_Run_InputParams" type="typens:DecoderParameters "/>

<message name="Decoder_Run_RequestMessage">
        <part name="Run_InputParameters" element="typens:Decoder_Run_InputParams"/>
</message>

<message name="Decoder_Run_ResponseMessage">
        <part name="Run_OutputParameters" element="typens:Decoder_Run_OutputParams"/>
</message>

<portType name="DecoderPortType">
        <operation name="runDecoder">
                <input name="Decoder_Run_RequestMessage"
              message="wsdlns:Decoder_Run_RequestMessage"/>
                <output name="Decoder_Run_ResponseMessage"
              message="wsdlns:Decoder_Run_ResponseMessage"/>
        </operation>
</portType>
```

Generated Java Interface

```
public interface DecoderPortType {
  public RunDecoderResultDocument
  runDecoder(RunDecoderDocument inputMsg);
}
```

Implementing Service

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

Embedded HTTP Container

Start simple Web Server that provides Web Service:

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

WSDL for service will be available at

*http://host:port/**decoder**?wsdl*

HTTP Mini Server
Key Features

- Implements HTTP 1.0 and common subset of HTTP 1.1
- Supports Keep-Alive
  - Client can reuse connection to send multiple requests and receive responses
- Each connection creates A new thread
  - Number of threads limited by number of connections (by default around 100s)
  - Other policies possible (Tomcat container)
- Supports SSL/TLS and GSI through extensible socket factory abstraciton

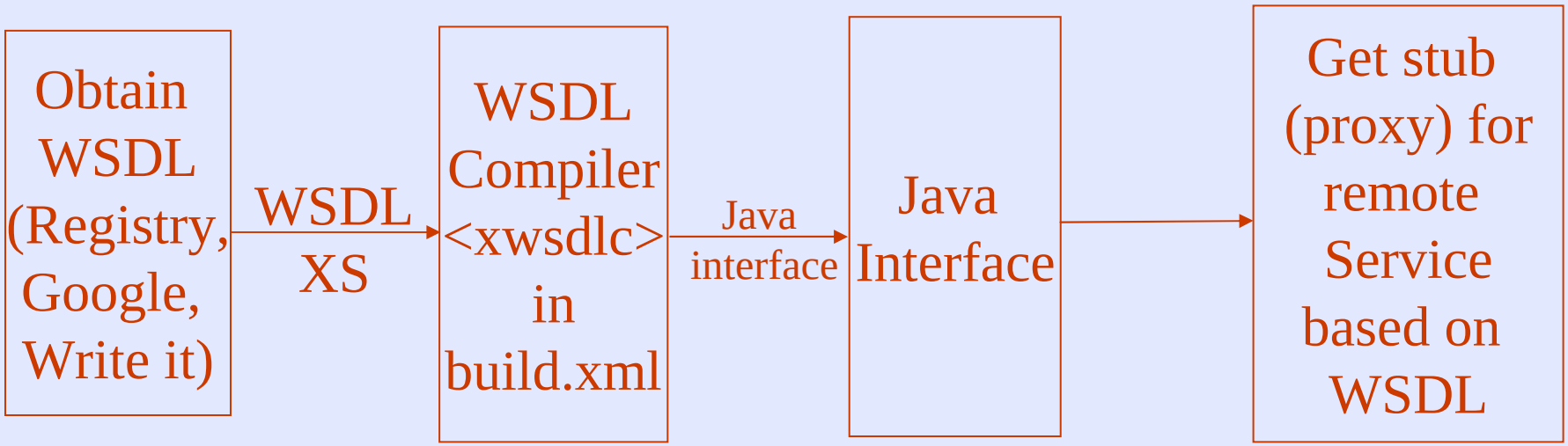
Accessing Web Service

- Web Service Invocation Framework (WSIF)
  - Dynamic service invocations without code generation
  - `java xsul.dii.XsulDynamicInvoker`
    WSDL portType parameters
- Example:

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

Service Invocation

Using XML Beans: steps involved



Accessing Web Service using XML Beans

- Prepare message using Java classes generated by XML beans

```
RunDecoderDocument inputMsg =
    RunDecoderDocument.Factory.newInstance();
DecoderParameters params=inputMsg.addNewRunDecoder();
params.setInputFile(…);
params.setOutputDirectory(…);
```

- Access service by using WSDL and generated Java Interface:

```
DecoderPortType stub = (DecoderPortType)
    XmlBeansWSIFRuntime.newClient(wsdlLoc);
    .generateDynamicStub(DecoderPortType.class);
…
RunDecoderResultDocument outputMsg =
    stub.runDecoder(inputMsg);
```

Asynchronous Messaging

- Server sied is automatically async enabled
  - using WS-Addressing to describe where message goes
- Client Side: AdderPT stub =

  WSIFRuntime.newClient(wsdlLoc)

  .useAsyncMessaging(correlator)

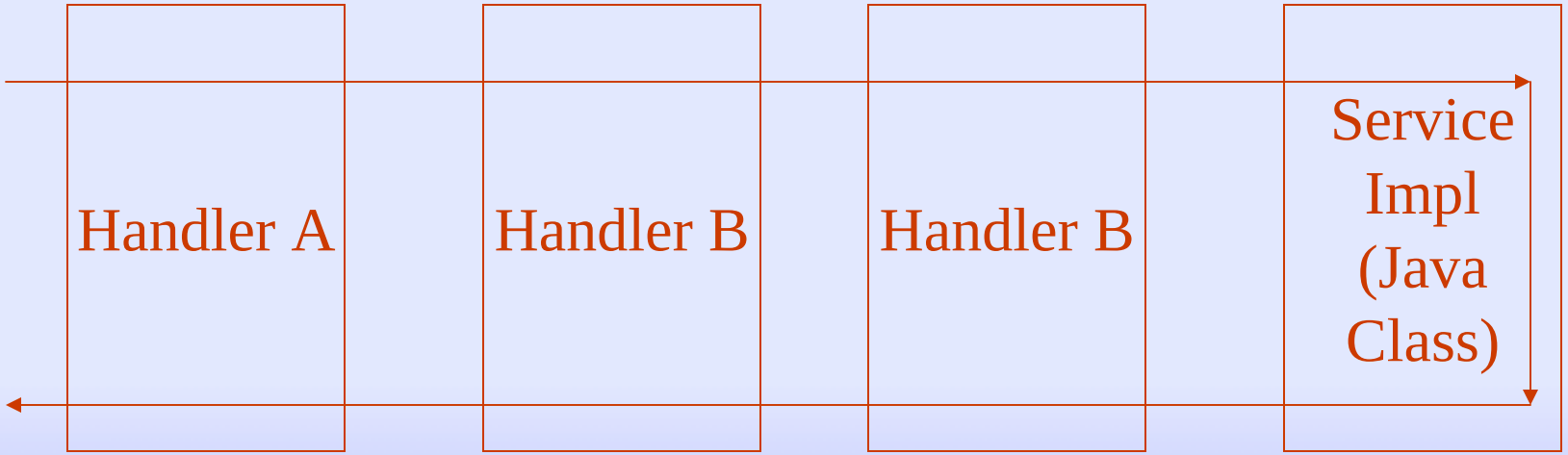
  .generateDynamicStub(AdderPT.class);
- Correlator is an interface that when implemented provides ability to correlate response message (asynchronous) with a request message sent
  - WS-Addressing correlator: client acts as a server that can receive asynchronous messages
    - Mini HTTP Server is started on client side
  - WS-MsgBox correlator: works over firewalls

Other modules (bricks)

- WS-Dispatcher and WS-MsgBox
- Low level HTTP Client/Server libraries
- SOAP 1.1 and SOAP 1.2 and *common* SOAP abstraction
- WSDL 1.1 support
- WS-Addressing
- Some RPC SOAP Encoding support
- Security: digital signatures, XPola, CapMan
- WSIF API implementation
- Sticky SOAP Headers
- Asynchronous invocation support
- Handlers framework
- …

Handler Chain Execution

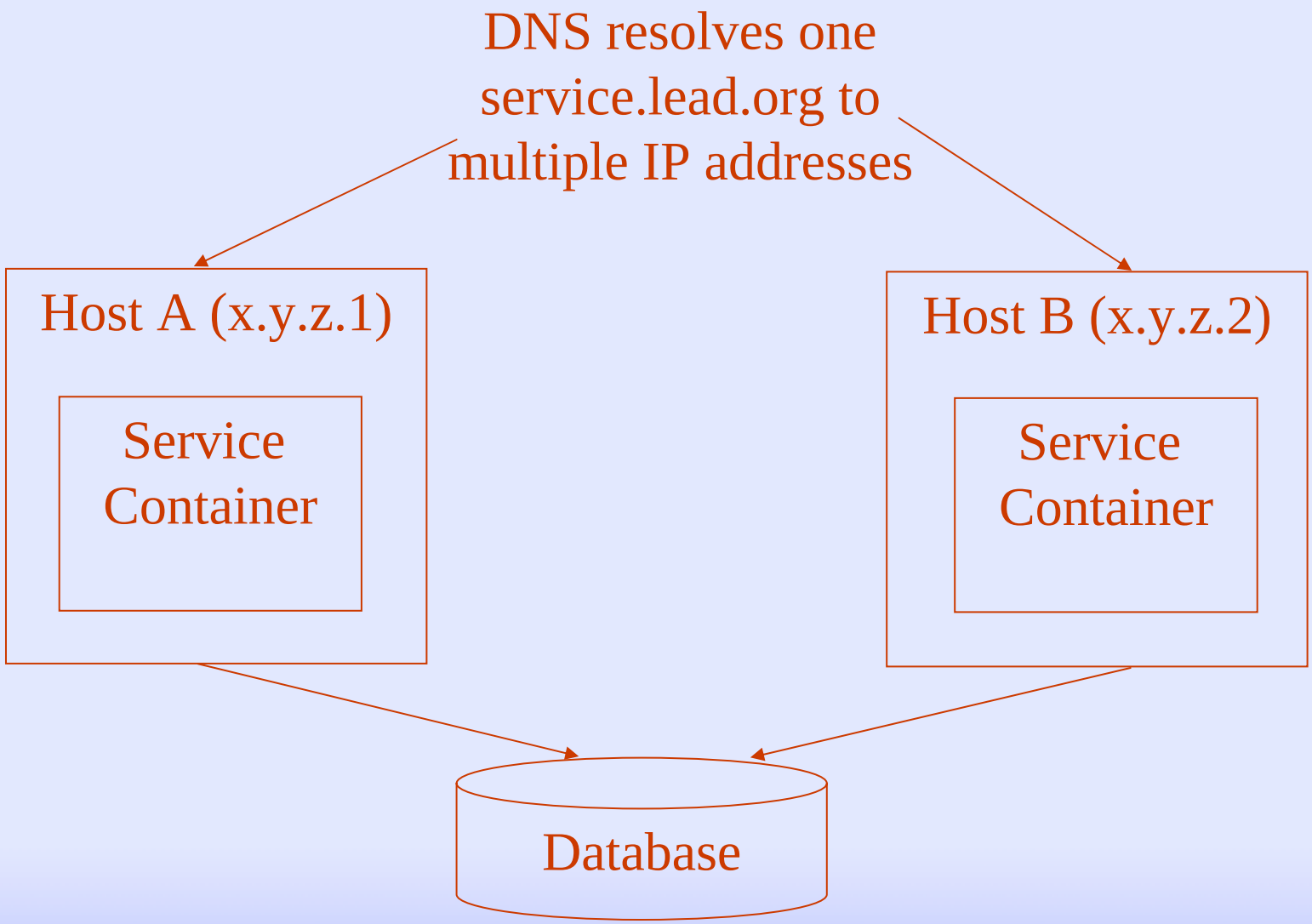
- Handlers contain code independent from service implementation (such as security)
- Client side first processing outgoing message then incoming message and no need to create faults – exceptions are local
- Server side execution flow:



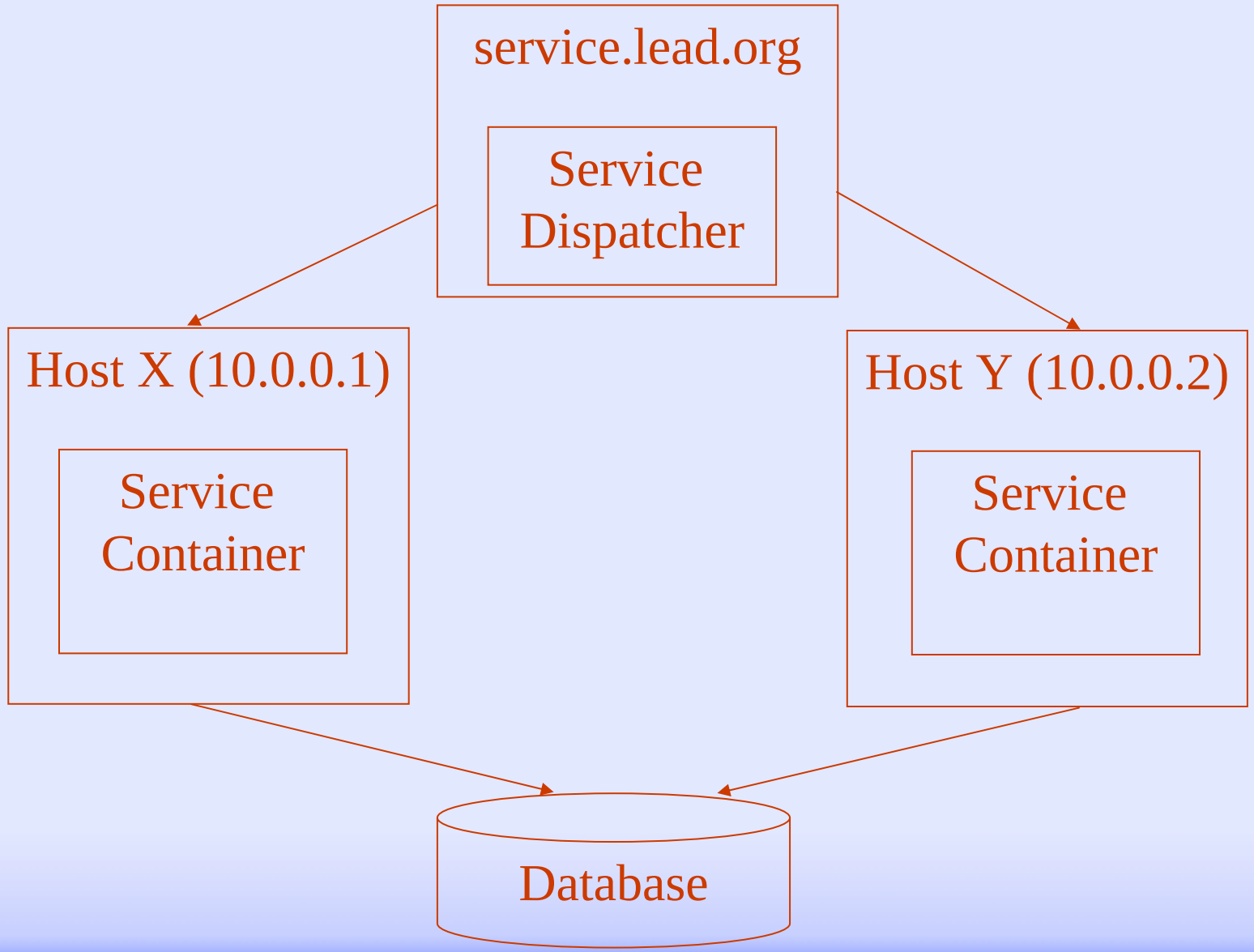
Future Work (XSUL 3.x)

- **Scalability and reliability**
- Monitoring
  - Easy way to check service “health”
- **Load balancing / clustering**
  - Higher performance, scalability, failover
  - Improved WS-MsgBox (clustered)
- Integration with Servlet Container
  - Deploy services to Tomcat
- WSDL 2.0 …

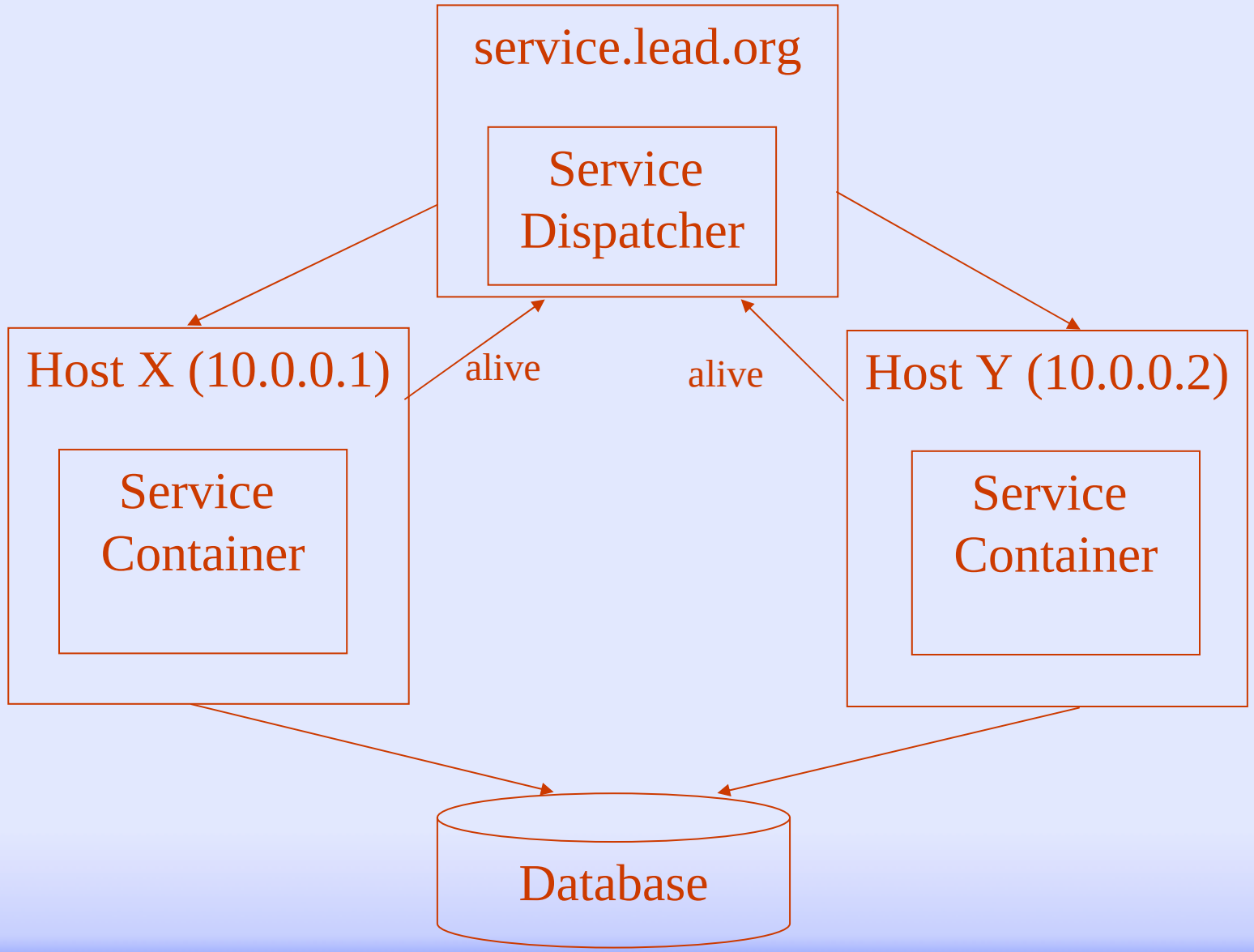
Load Balancing: DNS



Load Balancing: Dispatcher



Load Balancing: Fail-Over



Additional notes

- This kind of load balancing is *invisible* to service clients
  - Exactly as high-traffic Web Server is balanced
- It may be desirable to add load balancing to client:
- WS Reliable Messaging is required for reliability
  - Client retries messages until they are acknowledged
  - Good place to add “invisible” load balancing on client side
- DNS level balancing an dispatcher/failover can be combined to avoid SPoF (Single Point of Failure)
- Database can be replicated
  - For example master (read/write) and slaves (read)
- Portal may require sticky sessions and/or session replication