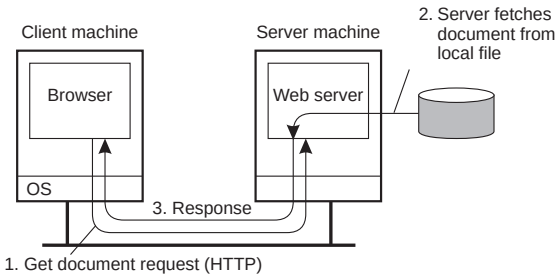
7 Distributed Web-Based Systems

7.1 Architecture

Distributed Web-based systems

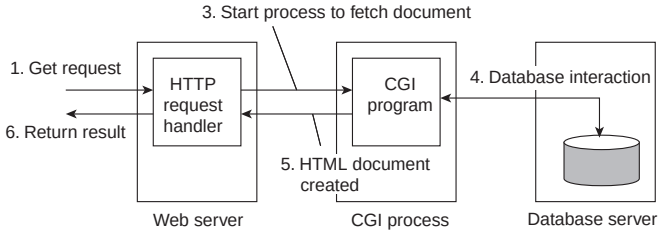
Essence
The WWW is a huge client-server system with millions of servers; each server hosting thousands of hyperlinked documents.

- Documents are often represented in text (plain text, HTML, XML)
- Alternative types: images, audio, video, applications (PDF, PS)
- Documents may contain scripts, executed by client-side software



Multi-tiered architectures

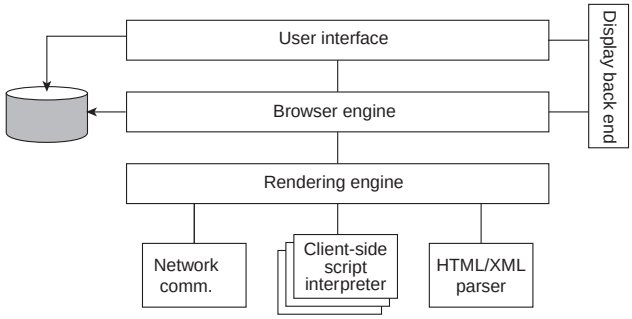
Observation
Web sites were soon organised into three tiers.



7.2 Processes

Clients: Web browsers

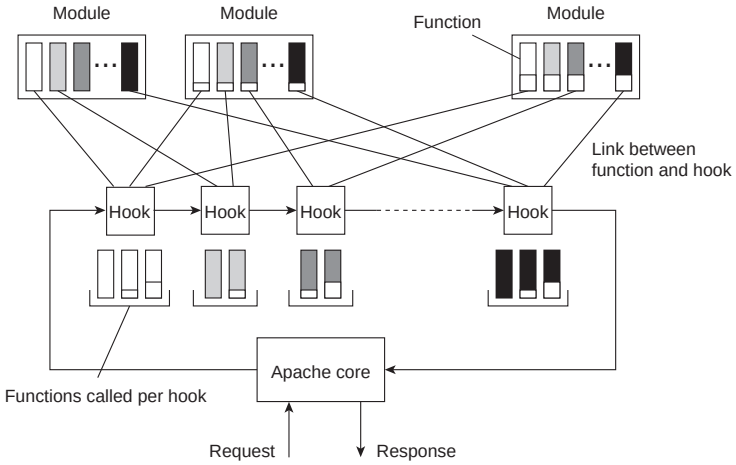
Observation
Browsers form the Web's most important client-side sofware. They used to be simple, but that is long ago.



Apache Web server

Observation: More than 52% of all 185 million Web sites are Apache.

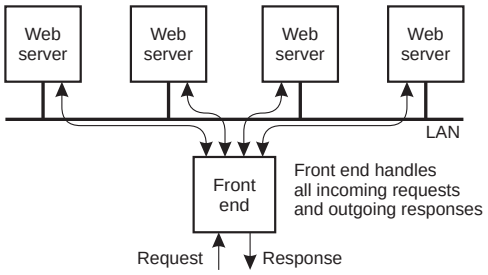
The server is internally organised more or less according to the steps needed to process an HTTP request.



Server clusters

Essence

To improve performance and availability, WWW servers are often clustered in a way that is transparent to clients.



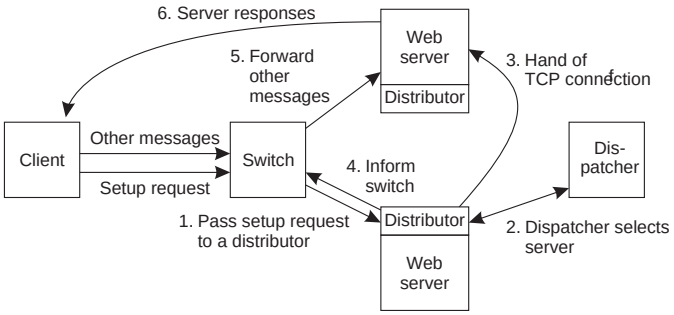
Server clusters

Problem
The front end may easily get overloaded, so that special measures need to be taken.

- Transport-layer switching: Front end simply passes the TCP request to one of the servers, taking some performance metric into account.
- Content-aware distribution: Front end reads the content of the HTTP request and then selects the best server.

Server Clusters

Question
Why can content-aware distribution be so much better?



7.3 Communication

Communication

Essence
Communication in the Web is generally based on HTTP; a relatively simple client-server transfer protocol having the following request messages.

Operation	Description
Head	Request to return the header of a document
Get	Request to return a document to the client
Put	Request to store a document
Post	Provide data that are to be added to a document (collection)
Delete	Request to delete a document

Communication

Header	C/S	Contents
Accept	C	The type of documents the client can handle
Accept-Charset	C	The character sets are acceptable for the client
Accept-Encoding	C	The document encodings the client can handle
Accept-Language	C	The natural language the client can handle
Authorization	C	A list of the client's credentials
WWW-Authenticate	S	Security challenge the client should respond to
Date	C+S	Date and time the message was sent
ETag	S	The tags associated with the returned document
Expires	S	The time for how long the response remains valid
From	C	The client's e-mail address
Host	C	The TCP address of the document's server
If-Match	C	The tags the document should have
If-None-Match	C	The tags the document should not have
If-Modified-Since	C	Return a document only if it has been modified since the specified time
If-Unmodified-Since	C	Return a document only if it has not been modified since the specified time
Last-Modified	S	The time the returned document was last modified
Location	S	A document reference to which the client should redirect its request
Referer	C	Refers to client's most recently requested document
Upgrade	C+S	The application protocol sender wants to switch to
Warning	C+S	Information about status of the data in the message

7.4 Naming

Naming: Uniform Resource Locator (URL)

Scheme		Host name	Pathname
http	://	www.cs.vu.nl	/home/steen/mbox

(a)

Scheme		Host name		Port	Pathname
http	://	www.cs.vu.nl	:	80	/home/steen/mbox

(b)

Scheme		Host name		Port	Pathname
http	://	130.37.24.11	:	80	/home/steen/mbox

(c)

http	HTTP	http://www.cs.vu.nl:80/globe
mailto	Mail	mailto:steen@cs.vu.nl
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	file:/edu/book/work/chp/11/11
data	Inline data	data:text/plain;charset=iso-8859-7, %e1%e2%e3
telnet	Remote login	telnet://flits.cs.vu.nl
tel	Telephone	tel:+31201234567
modem	Modem	modem:+31201234567;type=v32

URI vs URL vs URN

- URI: Universal Resource Identifier
  - Used to identify resources
  - Includes URL and URN
- URL: Universal Resource Locator
  - Is a URI used to locate resources
- URN: Universal Resource Name

  - Is a URI used only to identify a resource

- General form:
  - protocol:protocol specific
  - protocol can be http, ftp, urn, mailto, . . .

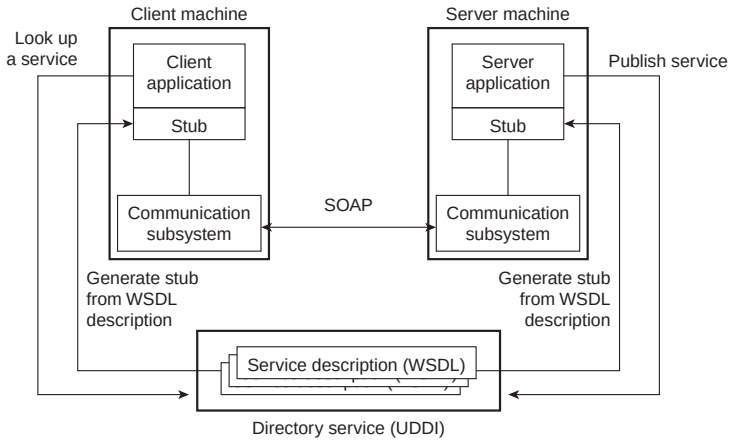
- Example:
  - `http://schemas.xmlsoap.org/soap/envelope/` is a URN used to name the SOAP 1.0 namespace
  - In addition it is a URL to obtain the XML Schema definition of SOAP 1.0

7.5 Web Services

Web services

Observation

At a certain point, people started recognising that it is was more than just user ↔ site interaction: sites could offer services to other sites ⇒ standardisation is then badly needed.



Service-Oriented Architectures (SOA)

- A set of principles for organising the software
- Not restricted to the use of Web services
- SOA principles:
  - Loose Coupling
  - Discoverability
  - Abstract service description (independent of implementation)
  - Encapsulation (autonomy and abstraction)
  - Compositionality
  - Additional for web services: based on open standards and vendor neutral

Web Services

- Providing services to other computer programs (not to Web browsers)
- Interoperability between software applications running on different computers
- Loosely coupled
- Machine-processable
- Use of standards: XML, HTTP, SOAP, WSDL, . . .
- Open infrastructure
- Language transparency
- Modular design

Web Services

Types of Web Service:

- SOAP-based (Simple Object Access Protocol)
- REST-style (REpresentational State Transfer)

XML Technologies Comprising Web Services:

- Message structure: Simple Object Access Protocol (SOAP)
- Description of Web services: Web Services Description Language (WSDL)
- Discovery of Web services: Universal Description Discovery and Integration (UDDI)

SOAP

Simple Object Access Protocol
Based on XML, this is the standard protocol for communication between Web services.

- SOAP is bound to an underlying protocol such as HTTP or SMTP (i.e., it is **not** independent from its carrier)
- Document-style exchange: Send a document one way, get a filled-in response back.
- RPC-style exchange: Used to invoke a Web service.

A Note on XML

Observation

XML has the advantage of allowing self-describing documents. Full stop (i.e., it introduces performance problems and is **not** meant to be read by human beings)

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Header>
        <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
            <n:priority>1</n:priority>
            <n:expires>2001-06-22T14:00:00-05:00</n:expires>
        </n:alertcontrol>
    </env:Header>
    <env:Body>
        <m:alert xmlns:m="http://example.org/alert">
            <m:msg>Pick up Mary at school at 2pm</m:msg>
        </m:alert>
    </env:Body>
</env:Envelope>
```

Java Web Services

- Java provides support for web services through JAX-WS
- JAX-WS = Java API for XML-Web Services.
- Java Web Services can be deployed in the following ways:
  - Core Java only
  - Core Java with the current Metro release
  - Stand-alone web container (e.g. Tomcat)
  - Java application server (e.g. Glassfish)

Java Web Services

- A SOAP-based web service can be implemented as a single Java class
- But usually consists of the following:
  - SEI (Service Endpoint Interface)
    - Declares the methods (web service operations)
  - SIB (Service Implementation Bean)
    - Defines the methods declared in the interface
    - Can be either POJO (Plain Old Java Object) or EJB (Enterprise Java Bean)

Writing a Web Service Client

- A Web service client is any program using a Web service, e.g. a Java application
- How to access the Web services:

  - send a HTTP POST request with the request as a SOAP message to the server
  - better: use the program wsimport to generate Java stubs to do this for you

- However, wsimport needs a description of the Web services offered by the Web server:
  - use the WSDL (Web Service Description Language) document generated by the Web server
  - The URL of this document can be obtained by looking at the Web services section at `http://localhost:4848`

A First Web Service: TimeServer

Task
Return the current time as either a string or as the elapsed milliseconds from the Unix epoch, midnight January 1, 1970 GMT.

TimeServer: SEI

```
package ch01.ts;  // time server

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

/**
 *  The annotation @WebService signals that this is the
 *  SEI (Service Endpoint Interface). @WebMethod signals
 *  that each method is a service operation.
 *
 *  The @SOAPBinding annotation impacts the under-the-hood
 *  construction of the service contract, the WSDL
 *  (Web Services Definition Language) document. Style.RPC
 *  simplifies the contract and makes deployment easier.
*/
@WebService
@SOAPBinding(style = Style.RPC) // more on this later
public interface TimeServer {
    @WebMethod String getTimeAsString();
    @WebMethod long getTimeAsElapsed();
}
```

TimeServer: SIB

```
package ch01.ts;

import java.util.Date;
import javax.jws.WebService;

/**
 *  The @WebService property endpointInterface links the
 *  SIB (this class) to the SEI (ch01.ts.TimeServer).
 *  Note that the method implementations are not annotated
 *  as @WebMethods.
*/
@WebService(endpointInterface = "ch01.ts.TimeServer")
```

```
public class TimeServerImpl implements TimeServer {
    public String getTimeAsString() { return new Date().toString(); }
    public long getTimeAsElapsed() { return new Date().getTime(); }
}
```

TimeServer: Endpoint Publisher

```
package ch01.ts;

import javax.xml.ws.Endpoint;

/**
 * This application publishes the Web service whose
 * SIB is ch01.ts.TimeServerImpl. For now, the service
 * service is published at network address 127.0.0.1.,
 * which is localhost, and at port number 9876, as this
 * port is likely available on any desktop machine. The
 * publication path is /ts, an arbitrary name.
 *
 * The Endpoint class has an overloaded publish method.
 * In this two-argument version, the first argument is the
 * publication URL as a string and the second argument is
 * an instance of the service SIB, in this case
 * ch01.ts.TimeServerImpl.
 *
 * The application runs indefinitely, awaiting service requests.
 * It needs to be terminated at the command prompt with control-C
 * or the equivalent.
 *
 * Once the applicatation is started, open a browser to the URL
 *
 *     http://127.0.0.1:9876/ts?wsdl
 *
 * to view the service contract, the WSDL document. This is an
 * easy test to determine whether the service has deployed
 * successfully. If the test succeeds, a client then can be
 * executed against the service.
*/
public class TimeServerPublisher {
    public static void main(String[ ] args) {
      // 1st argument is the publication URL
      // 2nd argument is an SIB instance
      Endpoint.publish("http://127.0.0.1:9876/ts", new TimeServerImpl());
    }
}
```

TimeServer: Compiling and Running

- Compiling the SEI, SIB and publisher: javac ch01/ts/*.java
- Running the publisher: java ch01.ts.TimeServerPublisher
- Testing the web service with the browser: Access the URL: http://127.0.0.1:9876/ts?wsdl
- Accessing the WSDL using *curl*: curl http://127.0.0.1:9876/ts?wsdl

TimeServer: Perl Client

```
#!/usr/bin/perl -w

use SOAP::Lite;
my $url = 'http://127.0.0.1:9876/ts?wsdl';
my $service = SOAP::Lite->service($url);

print "\nCurrent time is: ",
    $service->getTimeAsString();
print "\nElapsed milliseconds from the epoch: ",
    $service->getTimeAsElapsed(), "\n";
```

TimeServer: Ruby Client

```
#!/usr/bin/ruby

# one Ruby package for SOAP-based services
require 'soap/wsdlDriver'

wsdl_url = 'http://127.0.0.1:9876/ts?wsdl'

service = SOAP::WSDLDriverFactory.new(wsdl_url).create_rpc_driver

# Save request/response messages in files named '...soapmsgs...'
service.wiredump_file_base = 'soapmsgs'

# Invoke service operations.
result1 = service.getTimeAsString
result2 = service.getTimeAsElapsed

# Output results.
puts "Current time is: #{result1}"
puts "Elapsed milliseconds from the epoch: #{result2}"
```

TimeServer: HTTP Request

```
POST  http://127.0.0.1:9876/ts HTTP/ 1.1
Accept:  text/html
Accept:  multipart/*
Accept:  application/soap
User-Agent:  SOAP::Lite/Perl/0.69
Content-Length:  434
Content-Type:  text/xml; charset=utf-8
SOAPAction:  ""


<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tns="http://ts.ch01/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <soap:Body>
    <tns:getTimeAsString xsi:nil="true" />
  </soap:Body>
</soap:Envelope>
```

TimeServer: HTTP Response

```
HTTP/1.1 200 OK
Content-Length: 323
Content-Type: text/html; charset=utf-8
Client-Date: Mon, 28 Apr 2008 02:12:54 GMT
Client-Peer: 127.0.0.1:9876
Client-Response-Num: 1


<?xml version="1.0"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <soapenv:Body>
    <ans:getTimeAsStringResponse xmlns:ans="http://ts.ch01/">
      <return>Thu Mar 21 14:45:17 GMT 2013</return>
    </ans:getTimeAsStringResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

TimeServer: Java Client

```
package ch01.ts;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import java.net.URL;
class TimeClient {
    public static void main(String args[]) throws Exception {
        URL url = new URL("http://localhost:9876/ts?wsdl");

        // Qualified name of the service:
        //   1st arg is the service URI
        //   2nd is the service name published in the WSDL
        QName qname = new QName("http://ts.ch01/", "TimeServerImplService");

        // Create, in effect, a factory for the service.
        Service service = Service.create(url, qname);

        // Extract the endpoint interface, the service "port".
        TimeServer eif = service.getPort(TimeServer.class);

        System.out.println(eif.getTimeAsString());
        System.out.println(eif.getTimeAsElapsed());
   }
}
```

WSDL Document Structure

A WSDL document has two parts:

- Interface (abstract)
  - Available services
    - operations grouped in port types
  - Which messages are needed by the operations
    - A message can have parts
  - Used data types and XML-elements
- Implementation (concrete)
  - Binding to the message layer (e.g. SOAP)
    - How message parts are mapped to body/header elements of SOAP messages
  - Bindings to the transport layer (e.g. HTTP)
  - Where do I find the service?
    - A service may offer several ports, i.e. ways to call it

WSDL Document Structure

```
<definitions name="nmtoken"?  targetNamespace="uri"?>
```

- Interface
  - `<import namespace="uri" location="uri"/>*`

  - `<documentation .... />?`
  - `<types>?`
  - `<message name="nmtoken">*`
  - `<portType name="nmtoken">*`
- Implementation
  - `<binding name="nmtoken" type="qname">*`
  - `<service name="nmtoken">*`
  - `<-- extensibility element -->*`

WSDL Document Structure

- Definitions element

```
<definitions
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   name="TimeServerImplService"
   targetNamespace="http://ts.ch01/"
   xmlns:tns="http://ts.ch01/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" >
```

- The namespace `http://ts.ch01` is defined by using the targetNamespace attribute
- But `http://ts.ch01` is also used in the WSDL file
  - its namespace needs to be declared: `xmlns:tns="http://ts.ch01/"`
  - definition of a name space and its use are separated

WSDL Document Structure

- Types element: contain XML schemas for:
  - the messages being exchanged
  - the user defined data types like person data, registration data, etc.

```
<types>?
   <documentation .... />?
   <xsd:schema .... />*
   <!-- extensibility element -->*
```

- No special types are needed for the TimeServer service.

WSDL Document Structure

- Message part: messages exchanged between client and server
- This is an abstract description and only the binding actually determines the concrete XML/SOAP message
- `<message name="nmtoken">*`
  - `<documentation..../>?`
  - `<part name="nmtoken" element="qname"? type="qname"?/>*`
- the parts have different meanings depending on the binding
  - SOAP RPC binding:
    - each part is a parameter
  - SOAP document binding:
    - only one part for the body (can include several parameters)
    - additional parts are mapped to a header block

WSDL Document Structure

- For the TimeServer service:
  - Four messages: `getTimeAsString`, `getTimeAsStringResponse`, `getTimeAsElapsed`, `getTimeAsElapsedResponse`
  - `getTimeAsString` and `getTimeAsElapsed` have no parts
  - `getTimeAsStringResponse`, `getTimeAsElapsedResponse` have one part: `return`

```
<message name="getTimeAsString"></message>
<message name="getTimeAsStringResponse">
   <part name="return" type="xsd:string"></part>
</message>
<message name="getTimeAsElapsed"></message>
<message name="getTimeAsElapsedResponse">
   <part name="return" type="xsd:long"></part>
</message>
```

WSDL Document Structure

- The `portType` for TimeService has two operations, each with one input message and one output message

```
<portType name="TimeServer">
   <operation name="getTimeAsString" parameterOrder="">
      <input message="tns:getTimeAsString"></input>
      <output message="tns:getTimeAsStringResponse"></output>
   </operation>
   <operation name="getTimeAsElapsed" parameterOrder="">
      <input message="tns:getTimeAsElapsed"></input>
      <output message="tns:getTimeAsElapsedResponse"></output>
   </operation>
</portType>
```

WSDL: Message Exchange Patterns

- One-way
  - `<operation name="nmtoken">`
    - `<input name="nmtoken"? message="qname">`
- Request/Response
  - `<operation name="nmtoken" parameterOder="nmtokens"?>*`
    - `<input name="nmtoken"? message="qname">`
    - `<output name="nmtoken"? message="qname">`
    - `<fault name="nmtoken" message="qname">*`
- Solicit-response
  - `<operation name="nmtoken">*`
    - `<output name="nmtoken"? message="qname">`
    - `<input name="nmtoken"? message="qname">`
    - `<fault name="nmtoken" message="qname">*`
    - `<documentation..../>?`
- Notification
  - `<operation name="nmtoken">*`
    - `<output name="nmtoken"? message="qname">`

WSDL Document Structure

Implementation Part

- `<binding name="nmtoken" type="qname">*`
- `<service name="nmtoken">*`
- `<-- extensibility element -->*`
- binding section for each port type: How the operations in the port type section are realized using SOAP and HTTP
  - One port type can have several bindings
- service section: The service section defines how to reach a service by defining a location and a binding (how to communicate with that service)
  - It is possible that the operations of one port type are offered with several endpoints and different protocols

WSDL Document Structure

- The binding element describes how the abstract port type is mapped to an actual message exchange
  - Both message layer (e.g. SOAP) and transport layer (e.g. HTTP)
  - All elements with the `soap` prefix are not part of WSDL directly but of the binding description how to transport messages described with WSDL via SOAP
    * `xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"`

WSDL Document Structure

```
<binding name="TimeServerImplPortBinding" type="tns:TimeServer">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http">
   </soap:binding>
   <operation name="getTimeAsString">
      <soap:operation soapAction=""></soap:operation>
      <input>
         <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </input>
      <output>
         <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
       </output>
   </operation>
   <operation name="getTimeAsElapsed">
      <soap:operation soapAction=""></soap:operation>
      <input>
         <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </input>
      <output>
         <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
       </output>
   </operation>
</binding>
```

WSDL Document Structure

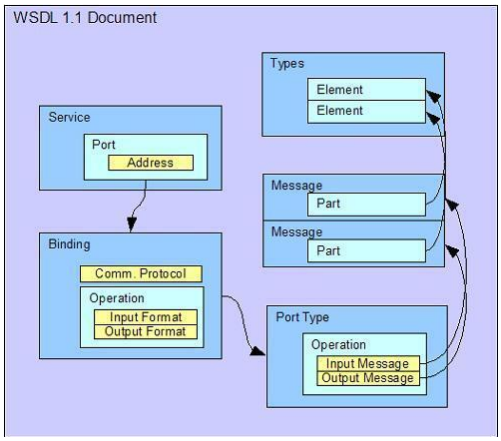
- The service consists of a set of ports.
- A port refers to a portType via a binding and defines an endpoint to access the service
- The service section can contain several ports each describing a way to access the operations defined in the port types by providing different endpoints and/or different bindings

```
<service name="TimeServerImplService">
   <port name="TimeServerImplPort" binding="tns:TimeServerImplPortBinding">
      <soap:address location="http://localhost:9876/ts"></soap:body>
   </port>
</service>
```

WSDL: Tying It Together 1

WSDL

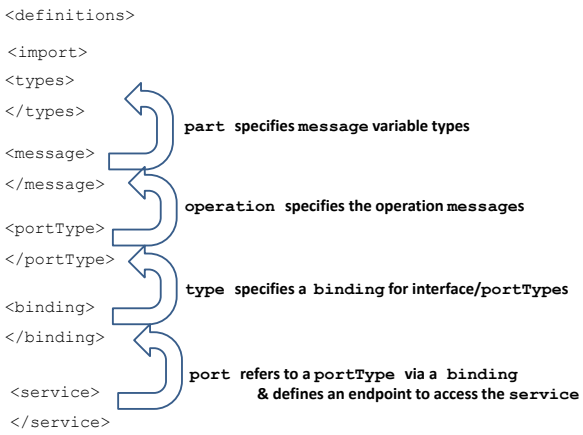
How all the parts fit together:



WSDL: Tying It Together 2

More WSDL things:

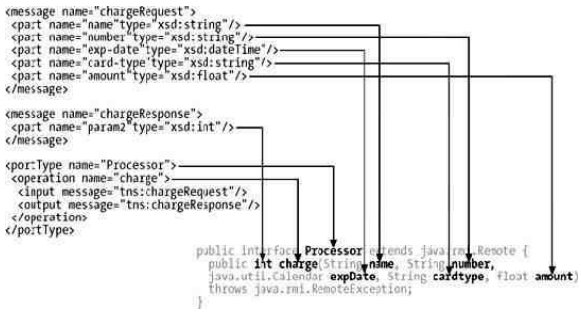
How all the parts fit together:



WSDL: Tying It Together 3

How does WSDL relate to an RMI Interface?

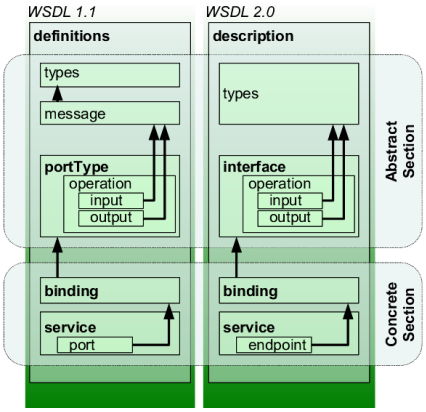
How all the parts fit together:



WSDL: The Old and The New

WSDL

WSDL's current specification is 2.0; WSDL 1.1 has not been endorsed by the W3C but WSDL 2.0 has.



Generating Client Support Code From WSDL

- After the WSDL has been generated by TimeServerPublisher, execute: wsimport -keep -p client http://localhost:9876/ts?wsdl
- The -keep option specifies that the source files should be kept
- The -p client option specifies the Java package in which the generated files are to be placed
- The above command generates two source and two compiled files in the subdirectory client

Approaches to Web Services 1: The *Contract-First* Approach

The above approach, where the WSDL *contract* is used to generate all the required artifacts for web service development, deployment, and invocation is known as the *Contract-First* Approach.

wsimport-generated TimeServer

```
package client;

import javax.jws.WebMethod;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name = "TimeServer", targetNamespace = "http://ts.ch01/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface TimeServer {

    @WebMethod
    @WebResult(partName = "return")
    public String getTimeAsString();

    @WebMethod
    @WebResult(partName = "return")
    public long getTimeAsElapsed();
}
```

wsimport-generated TimeServerImplService

```
package client;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceFeature;

@WebServiceClient(name = "TimeServerImplService",
                  targetNamespace = "http://ts.ch01/",
                  wsdlLocation = "http://localhost:9876/ts?wsdl")
public class TimeServerImplService extends Service {

    private final static URL TIMESERVERIMPLSERVICE_WSDL_LOCATION;

    static {
```

```
        URL url = null;
        try {
            url = new URL("http://localhost:9876/ts?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        TIMESERVERIMPLSERVICE_WSDL_LOCATION = url;
    }
```

wsimport-generated TimeServerImplService

```
    public TimeServerImplService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public TimeServerImplService() {
        super(TIMESERVERIMPLSERVICE_WSDL_LOCATION,
              new QName("http://ts.ch01/","TimeServerImplService"));
    }

    @WebEndpoint(name = "TimeServerImplPort")
    public TimeServer getTimeServerImplPort() {
        return (TimeServer)super.getPort(
                new QName("http://ts.ch01/","TimeServerImplPort"),
                TimeServer.class);
    }

}
```

Client Using wsimport-generated Support Code

```
package client;

class TimeClientWSDL {
    public static void main(String[] args) {
       // The TimeServerImplService class is the Java type bound to
       // the service section of the WSDL document.
       TimeServerImplService service = new TimeServerImplService();

       // The TimeServer interface is the Java type bound to
       // the portType section of the WSDL document.
       TimeServer eif = service.getTimeServerImplPort();

       // Invoke the methods.
       System.out.println(eif.getTimeAsString());
       System.out.println(eif.getTimeAsElapsed());
    }
}
```

Observation

This is much easier to write than the previous client as there is no need for the `QName` stuff.

SOAP Message Structure

- Envelope (mandatory)
  - Top element of the XML document representing the message
- Header (optional)

  - Determines how a recipient of a SOAP message should process the message
  - Adds features to the SOAP message such as authentication, transaction management, message routes, etc...

- Body (mandatory)
  - Exchanges information intended for the recipient of the message.
  - Typical use is for RPC calls and error reporting.

SOAP Body: RPC Style Request

- Calling operation $op(p_1, \ldots, p_n)$
  - Use operation name op as root tag in the SOAP body
  - Example: getTeam for getTeam(name: String)
  - Arguments are sublements (tag name is irrelevant)

```
<s:Envelope>
   <s:Header>
      <n:p3 xmlns="http://...">...</n:p3>
   </s:Header>/
   <s:Body>
      <n:op xmlns="http://...">
         <n:p1>...</n:p1>
         <n:p2>...</n:p2>
         ...
      </n:op>
   </s:Body>
</s:Envelope>
```

SOAP Body: RPC Style Response

- Response to operation call $op(p_1, \ldots, p_n)$
  - Use tag opResponse
  - Can have more than one return parameter

```
<s:Envelope>
   <s:Header>
      <n:p3 xmlns="http://...">...</n:p3>
   </s:Header>/
   <s:Body>
      <n:opResponse xmlns="http://...">
         <n:p1>...</n:p1>
         <n:p2>...</n:p2> ...
      </n:op>
   </s:Body>
</s:Envelope>
```

SOAP Header

- SOAP header can be used for:
  - processing instructions for the service intermediaries
    - signing/encrypting/decrypting a message
    - logging a message
  - routing of messages
    - who should be dealing with that message?
  - context/meta data and transaction management
    - transaction identifier/start of transaction/end of transaction/common data
    - information necessary to establish reliable messaging

SOAP Header: Attributes

- `role="next|none|ultimateReceiver|.."` ?
  - called `actor` in SOAP 1.1
  - standard roles: `next`, `none`, `ultimateReceiver`
  - user defined roles: `http://example.com/Log`
  - identifies the SOAP intermediary that needs to act on this header information
  - role takes the form of a URI, e.g. `http://www.w3.org/2003/05/soap-envelope/role/next` for next role
- `mustUnderstand="true"/"false"` ?
  - Is it mandatory that the header is processed or optional?
- `relay="true"/"false"` (only in SOAP 1.2)
  - If the header block cannot be processed, forward/relay it to the next intermediary or not

SOAP Fault

- Used to carry error/status information within a SOAP message
- Appears within the SOAP body
- Defines the following:
  - faultcode (mandatory)
    - Possible values: `VersionMismatch`, `MustUnderstand`, `Client`, `Server`
  - faultstring (mandatory)

        * Human readable explanation of the fault
    - faultactor (optional)
        * URI identifying the faulty actor
    - detail
        * Needs to be present in case of an error in the body

```
<s:Fault>
   <faultcode>s:Server</faultcode>
   <faultstring>Internal Application Error</faultstring>
   <detail xmlns:f=http://www.a.com/CalculatorFault>
      <f:errorCode>794634</f:errorCode>
      <f:errorMsg>Divide by zero</f:errorMsg>
   </detail>
</s:Fault>
```

SOAP Request

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
      time_request
    </ms:TimeRequest>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body/>
</SOAP-ENV:Envelope>
```

SOAP Response

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
      time_request
    </ms:TimeRequest>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
      Mon Oct 27 14:45:53 CDT 2008
    </ms:TimeRequest>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example Web Service with Richer Data Types: Teams

```
package ch01.team;

import java.util.List;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Teams {
    private TeamsUtility utils;

    public Teams() {
       utils = new TeamsUtility();
       utils.make_test_teams();
    }
```

```
    @WebMethod
    public Team getTeam(String name) { return utils.getTeam(name); }

    @WebMethod
    public List<Team> getTeams() { return utils.getTeams(); }
}
```

Teams: Additional Classes

```
package ch01.team;

import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

public class TeamsUtility {
   private Map<String, Team> team_map;

   public TeamsUtility() {
      team_map = new HashMap<String, Team>();
   }

   public Team getTeam(String name) {
      return team_map.get(name);
   }

   public List<Team> getTeams() {
      List<Team> list = new ArrayList<Team>();
      Set<String> keys = team_map.keySet();
      for (String key : keys)
         list.add(team_map.get(key));
      return list;
   }
```

Teams: Additional Classes

```
   public void make_test_teams() {
      List<Team> teams = new ArrayList<Team>();

      Player burns = new Player("George Burns", "George");
      Player allen = new Player("Gracie Allen", "Gracie");
      List<Player> ba = new ArrayList<Player>();
      ba.add(burns); ba.add(allen);
      Team burns_and_allen = new Team("Burns and Allen", ba);
      teams.add(burns_and_allen);

      Player abbott = new Player("William Abbott", "Bud");
      Player costello = new Player("Louis Cristillo", "Lou");
      List<Player> ac = new ArrayList<Player>();
      ac.add(abbott); ac.add(costello);
      Team abbott_and_costello = new Team("Abbott and Costello", ac);
      teams.add(abbott_and_costello);

      Player chico = new Player("Leonard Marx", "Chico");
      Player groucho = new Player("Julius Marx", "Groucho");
      Player harpo = new Player("Adolph Marx", "Harpo");
      List<Player> mb = new ArrayList<Player>();
      mb.add(chico); mb.add(groucho); mb.add(harpo);
      Team marx_brothers = new Team("Marx Brothers", mb);
      teams.add(marx_brothers);

      store_teams(teams);
   }
   private void store_teams(List<Team> teams) {
      for (Team team : teams)
         team_map.put(team.getName(), team);
   }
}
```

Teams: Additional Classes

```
package ch01.team;
public class Player {
    private String name;
    private String nickname;
```

```
    public Player() { }
    public Player(String name, String nickname) {
       setName(name);
       setNickname(nickname);
    }

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
    public void setNickname(String nickname) { this.nickname = nickname; }
    public String getNickname() { return nickname; }
}
```

Teams: Additional Classes

```
package ch01.team;

import java.util.List;
public class Team {
    private List<Player> players;
    private String name;

    public Team() { }
    public Team(String name, List<Player> players) {
       setName(name);
       setPlayers(players);
    }

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
    public void setPlayers(List<Player> players) { this.players = players; }
    public List<Player> getPlayers() { return players; }
    public void setRosterCount(int n) { } // no-op but needed for property
    public int getRosterCount() {
       return (players == null) ? 0 : players.size();
    }
}
```

Teams: Publishing the Service

```
package ch01.team;
import javax.xml.ws.Endpoint;
class TeamsPublisher {
    public static void main(String[ ] args) {
       int port = 8888;
       String url = "http://localhost:" + port + "/teams";
       System.out.println("Publishing Teams on port " + port);
       Endpoint.publish(url, new Teams());
    }
}
```

Teams: Writing a Client TeamClient.java

```
import teamsC.TeamsService;
import teamsC.Teams;
import teamsC.Team;
import teamsC.Player;
import java.util.List;
class TeamClient {
    public static void main(String[] args) {
        TeamsService service = new TeamsService();
        Teams port = service.getTeamsPort();

        List<Team> teams = port.getTeams();
        for (Team team : teams) {
```

```
            System.out.println("Team name: " + team.getName() +
                               " (roster count: " + team.getRosterCount() + ")");
            for (Player player : team.getPlayers())
                System.out.println("  Player: " + player.getNickname());
        }
    }
}
```

Teams: Compiling and Running

- Compile the source files: javac ch01/team/*.java
- Generate various Java classes needed by the method `Endpoint.publish` to generate the service's WSDL: wsgen -cp . ch01.team.Teams
- Run the `TeamsPublisher` application: java ch01.team.TeamsPublisher
- Generate various Java classes in the *teamsC* subdirectory to make it easier to write a client using the service: wsimport -p teamsC -keep http://localhost:8888/teams?wsdl

Teams: Compiling and Running

- Run the client: java TeamClient
- The output should be as follows:

```
Team name: Abbott and Costello (roster count: 2)
  Player: Bud
  Player: Lou
Team name: Marx Brothers (roster count: 3)
  Player: Chico
  Player: Groucho
  Player: Harpo
Team name: Burns and Allen (roster count: 2)
  Player: George
  Player: Gracie
```

A Second Approach to Web Services

Approaches to Web Services 2: The *Code-First* Approach

The above approach, where the Java classes are used to generate all the required artifacts for web service development, deployment, and invocation is known as the *Code-First* Approach.

- The command above: wsgen -cp . ch01.team.Teams illustrates another approach.
- This contrasts with the *Contract-First* seen earlier which was a top-down approach to generate JAX-WS Artifacts
- In general, for a number of reasons the *Contract-First* approach is preferred to *Code-First*

How to pick a tool?

The following lists the process to create a web service starting from Java sources, classes, or a WSDL file (server side):

- Starting from Java classes use *Code-First*:
  - Use wsgen to generate portable artifacts[1].
  - Deploy the Web Service
- Starting from a WSDL file use *Contract-First*:
  - Use wsimport to generate portable artifacts.
  - Implement the service endpoint.
  - Deploy the Web Service

The following lists the process to invoke a web service (client side):

- Starting from deployed web service's WSDL
- Use wsimport to generate the client-side artifacts.
- Implement the client to invoke the web service.

A Compromise Approach

Code First, Contract Aware

Updating a *Code-First* service, you might find that the WSDL changes as well. To get around this, there is a style called *Code First, Contract Aware*, where you write the code first but use available annotations to tightly constain the generated WSDL.

Some annotations:

- `@WebMethod`, indicates a method that is exposed as a Web Service operation,
- `@SOAPBinding` Specifies the mapping of the Web Service onto the SOAP message protocol
- `@WebParam` maps of a parameter to a Web Service message part and XML element,
- `@WebResult` specifies that the operation result in the generated WSDL is something other than the default `return` e.g. `IntegerOutput`.

[1] such as Service Endpoint Interface (SEI) class, Service Endpoint Implementation class etc