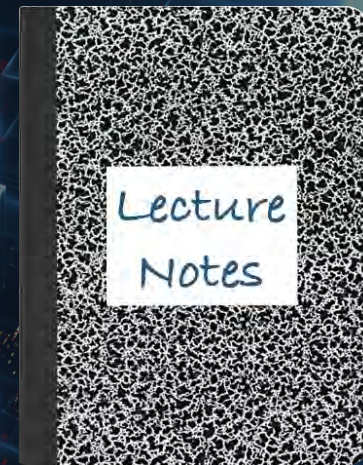


CS 417 – DISTRIBUTED SYSTEMS

# Week 3: Part 1

## Web Services

Paul Krzyzanowski



© 2023 Paul Krzyzanowski. No part of this content may be reproduced or reposted in whole or in part in any manner without the permission of the copyright owner.

# The early web: 1990s

- **Hypertext and HTML**

- Researchers can link documents together
- HTML provides rudimentary formatting

- **Clients: web browsers**

- Netscape navigator, Microsoft Internet Explorer
- Interpret HTML and present formatted content to the user

- **Static formatted content**

- Collections of HTML documents stored on servers
- Every user received the same content

# Evolution toward interactivity

- **Web usage grows beyond academia**
  - Need for interactivity (queries, forms, commerce)
  - Need for richer formatting (tables, iFrames, formatting directives)
- **Servers: Common Gateway Interface (CGI)**
  - Web server forwards certain pathnames (URLs) to executable programs
    - Often Perl scripts initially
  - Server scripts process the request data and generate a return page
  - Enabled rudimentary dynamic content
  - Emergence of data-driven websites
  - Then server-side web application frameworks:
    - Java Servlets, ColdFusion, PHP, Active Server Pages, Ruby on Rails, ...
- **Clients: Java applets**
  - Compiled Java code downloaded from a server as part of a web page
  - Embedded in the page and provided client-side interactivity
  - Made obsolete with the emergence of HTML5, CSS3, and JavaScript

# Limitations

- Web browser:
  - Dominant model for user interaction on the Internet
- Not good for programmatic access to data or manipulating data
  - UI is a major component of the content
  - Data and presentation are not separated
  - *Site scraping* is a pain!
- We wanted
  - **Remotely hosted services – that programs can use**
  - **Machine-to-machine communication**

# Failures of distributed objects in practice

- RPC and distributed objects were built around the idea that you can access remote methods just you would access local ones
- Ideally, you can pass references to objects and the code wouldn't care if they are local or remote
- BUT
  - Remote objects behaved differently
  - There's A LOT more overhead to calling a remote procedure
  - There were lots of ways to fail:
    - Server can die, response can get lost, network could get disconnected, unexpected delays & retransmission
    - *In practice, we cannot ignore the network!*

# RPC and distributed objects had problems

- **Interoperability** – Didn't always play well across different languages, OSes, platforms
- **Transparency** – We couldn't ignore the network & remote execution parts!
  - Memory access, partial failure
- **Firewalls** – RPCs usually use dynamic ports
- **Stateful** – RPCs don't support load balancing, require use of server resources
- **Asynchronous messaging is useful but doesn't fit the RPC model**
  - Large streaming responses
  - Notifications of processing delays, callbacks
  - Publish-subscribe notification models

*Distributed objects mostly ended up in intranets  
of homogenous systems and low-latency networks*

# Think about remote services differently

To program efficiently (make best use of the network) and defensively (handle failures), we may need to think about services differently

If we are aware of the overhead of the network  
(as well as other aspects, like servers that reboot, load balancers, ...)

... then we may design our operations differently

- Example

- Instead of iterating through 100 items and calling an RPC to look up the price
- We might send the entire list of 100 items and request a list of prices

# Web Services

Set of protocols by which services can be published, discovered, and used in a technology neutral form

- Language & architecture independent
- Applications will typically invoke multiple remote services

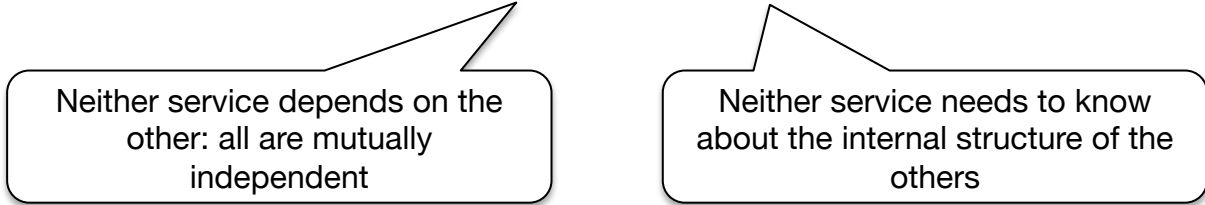
**Microservices in a Service-Oriented Architecture (SOA)**



# Service Oriented Architecture (SOA)

Microservices = SOA = Programming model

- The app is an integration of network-accessible services (components)
- Each service has a well-defined interface
- Components are **unassociated** & **loosely coupled**



Neither service depends on the other: all are mutually independent

Neither service needs to know about the internal structure of the others

# Benefits of microservices (SOA)

- **Autonomous modules**

- Each module does one thing well
- Supports reuse of modules across applications

- **Loose coupling**

- Requires minimal knowledge – don't need to know implementation
- **Migration:** Services can be located and relocated on any servers
- **Scalability:** new services can be added/removed on demand  
... and on different servers – or load balanced
- **Updates:** Individual services can be replaced without interruption

# General Principles of Web Services

- **Coarse-grained**
  - Usually few operations & large messages
- **Platform neutral**
  - Messages don't rely on the underlying language, OS, or hardware
  - Standardized protocols & data formats
  - Payloads are some standard format (text XML or JSON, binary protocol buffers)
- **Message-oriented**
  - Communicate by exchanging messages
- **HTTP** usually used for transport
  - Use existing infrastructure: web servers, authentication, encryption, firewalls, load-balancers

# Web Services vs. Distributed Objects

Web Services	Distributed Objects
Document Oriented <ul style="list-style-type: none"><li>• Exchange documents</li></ul>	Object Oriented <ul style="list-style-type: none"><li>• Instantiate remote objects</li><li>• Request operations on a remote object</li><li>• Receive results</li><li>• ...</li><li>• Delete the object when done</li></ul>
Document design is the key* <ul style="list-style-type: none"><li>• Interfaces are just a way to pass documents</li></ul>	Interface design is the key <ul style="list-style-type: none"><li>• Messages just package data</li></ul>
Servers are stateless <ul style="list-style-type: none"><li>• State is contained within the documents that are exchanged (e.g., customer ID)</li></ul>	Stateful interactions <ul style="list-style-type: none"><li>• Remote object stores state</li></ul>

*\*Disclaimer: sort of – with some interfaces, like REST, the interface is the URL*

# XML RPC

# Origins

- Born: early 1998
- Data marshaled into XML messages
  - All request and responses are human-readable XML
- Explicit typing
- Transport over HTTP protocol
  - Solves firewall issues
- No IDL compiler support for most languages
  - Lots of support libraries
  - Great support in some languages – like those that support introspection (Python, Perl)
- Example: WordPress uses XML-RPC

# XML-RPC example

```
<methodCall>
  <methodName>
    sample.sumAndDifference
  </methodName>
  <params>
    <param><value><int> 5 </int></value></param>
    <param><value><int> 3 </int></value></param>
  </params>
</methodCall>
```

# XML-RPC data types

- int
- string
- boolean
- double
- dateTime.iso8601
- base64
- array
- struct



# Assessment

- Simple (spec about 7 pages)
- Humble goals
- Good language support
  - Little/no function call transparency for some languages
- No garbage collection, remote object references, etc.
  - Focus is on data messaging over HTTP transport
- Little industry support (Apple, Microsoft, Oracle, ...)
  - Mostly grassroots and open source

# SOAP

# SOAP origins

## (Simple) (Object) Access Protocol

- Since 1998 (latest: v1.2 April 2007)
- Started with strong Microsoft & IBM support
- Continues where XML-RPC left off:
  - XML-RPC is a 1998 simplified subset of SOAP
  - User-defined data types
  - Adds error handling, extensions, interaction types, headers for metadata, ...

# SOAP

- Stateless messaging model
- Basic facility is used to build other interaction models
  - Request-response (RPC)
  - Request-multiple response
  - Asynchronous notification
- Objects marshaled and unmarshaled to SOAP-format XML
  - Usually sent over HTTP
- Like XML-RPC, SOAP is a messaging format
  - No garbage collection or object references
  - Does not define transport
  - Does not define stub generation

# SOAP Web Services and WSDL

- **Web Services Description Language**
  - Analogous to an IDL
- A **WSDL** document describes a set of services
  - Name, operations, parameters, where to send requests
  - Goal is that organizations will exchange WSDL documents
    - If you get WSDL document, you can feed it to a program that will generate software to send and receive SOAP messages

# WSDL Structure

<definitions>

<types>

data types used by web service: defined via XML Schema syntax

</types>

<message>

describes data elements of operations: parameters

</message>

<portType>

describes the service: operations and messages involved

</portType>

<binding>

defines message format & protocol details for each port

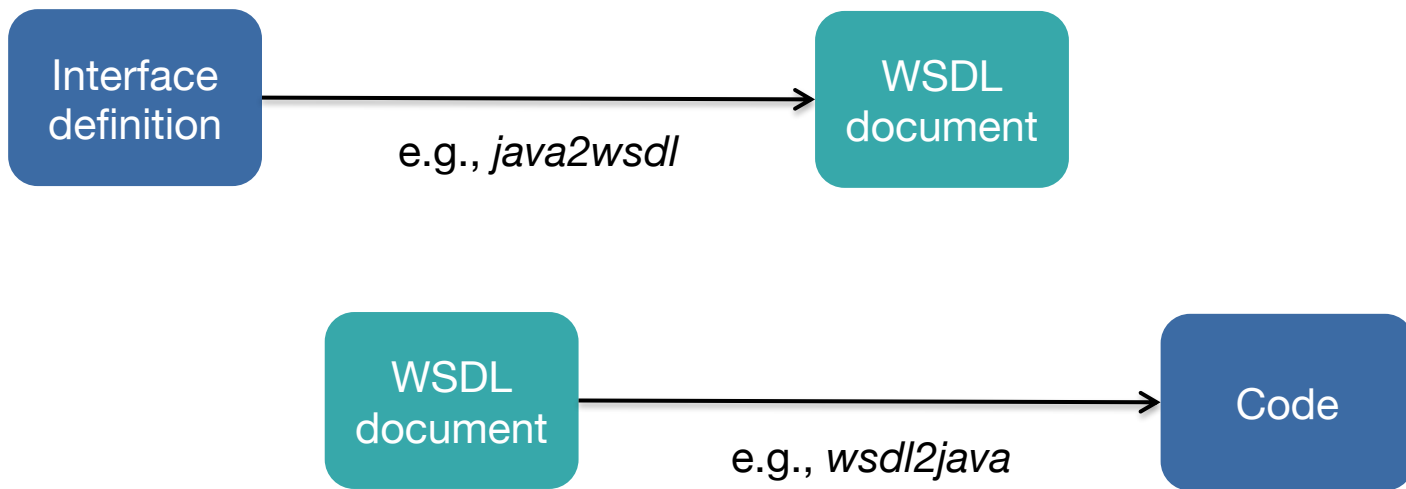
</binding>

</definitions>

# Java Web Services

# What do we do with WSDL?

It's an IDL – not meant for human consumption





# JAX-WS: Java API for XML Web Services

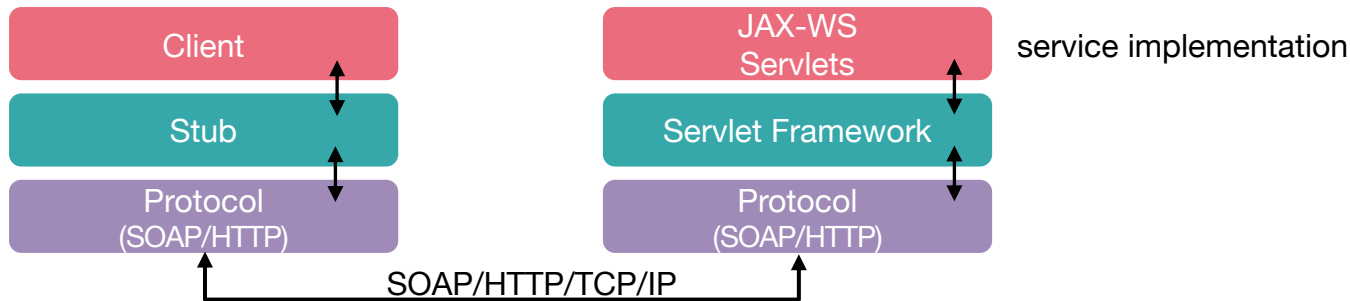
- Lots of them!
- Example: JAX-WS (evolved from earlier JAX-RPC)
  - Java API for XML-based Web-Service messaging & RPCs
  - Invoke a Java-based web service using Java RMI
  - Interoperability is a goal
    - Use SOAP & WSDL
    - Java not required on the other side (client or server)
- Service
  - Defined to clients via a WSDL document

# JAX-WS: Creating an RPC Endpoint

- Server
  - Define an interface (Java interface)
  - Implement the service
  - Create a publisher
    - Creates an instance of the service and publishes it with a name
- Client
  - Create a proxy (client-side stub)
    - *wsimport* command takes a WSDL document and creates a stub
  - Write a client that creates an instance of the service and invokes methods on it (calling the stub)

# JAX-RPC Execution Steps

1. Java client calls a method on a stub
2. The stub creates marshals the request into a SOAP message for the web service
3. The request is sent to the server.
4. Server gets the call and directs it to the framework
5. Framework calls the implementation
6. The implementation returns results to the framework
7. The framework marshals the results into a SOAP message
8. The sends the results back to the client stub
9. The client stub returns the information to the caller



# The future of SOAP?

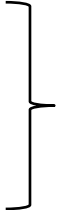
- Still used (mostly in large enterprises) but...
  - Required a framework – you will never create & parse messages yourself
  - Language support is not always great
  - Hard to understand & hard to use in many cases
  - Allegedly complex because “*we want our tools to read it, not people*”
    - *unnamed Microsoft employee*
  - Heavyweight: XML + verbose messaging structure
- Dropped by Google APIs in 2006
- Still used in many places, including Microsoft & Salesforce APIs

*But we wanted something lighter and easier...*

# REST

# REST

## REpresentational SState TTransfer

- Resource and parameters specified in URI
- Four HTTP commands let you manipulate data (a resource):
  - **POST** (create)
  - **GET** (read)
  - **PUT** (update/replace)
  - **DELETE** (delete)

**CRUD** primitives: *Create, Read, Update, Delete*
- And sometimes others:
  - **PATCH** (update/modify) – modify part of a resource (PUT is expected to modify all)
  - **OPTIONS** (query) – determine options associated with a resource
    - Rarely used ... but it's there
- Message body contains the data – usually in JSON format

# REST Principles

- Data is sent and received via HTTP
  - It is usually structured as a JSON or XML document (but can be anything, such as a comma-separated list)
- Interactions are stateless (sessions do not persist between requests)
- Interfaces should make use of URLs (including parameters) and the HTTP CRUD primitives
  - **POST** (create), **GET** (read), **PUT** (update/replace), **PATCH** (update/modify), **DELETE** (delete)

# Resource-oriented services

## Blog example

- Get a user's blogroll – a list of blogs subscribed by a user

`HTTP GET //myblogs.org/listsubs?user=paul`

- To get info about a specific blog (id = 12345):

`HTTP GET http://myblogs.org/bloginfo?id=12345`



# Resource-oriented services

- Get parts info

HTTP GET //www.parts-depot.com/parts

- Returns a document containing a list of parts

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
  <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
  <Part id="00347" xlink:href="http://www.parts-depot.com/parts/00347"/>
  <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
</p:Parts>
```

# Resource-oriented services

- Get detailed parts info:

HTTP GET //www.parts-depot.com/parts/00345

- Returns a document with information about a specific part

```
<?xml version="1.0"?>
<p:Part xmlns:p="http://www.parts-depot.com"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <Part-ID>00345</Part-ID>
  <Name>Widget-A</Name>
  <Description>This part is used within the frap assembly</Description>
  <Specification
xlink:href="http://www.parts-depot.com/parts/00345/specification"/>
    <UnitCost currency="USD">0.10</UnitCost>
    <Quantity>10</Quantity>
  </p:Part>
```

# Examples of REST services

- Various Amazon & Microsoft APIs
- Facebook Graph API
- Yahoo! Search APIs
- Flickr
- Twitter

- Open Zing Services – Sirius radio

`svc://Radio/ChannelList`

`svc://Radio/ChannelInfo?sid=001-siriushits1&ts=2007091103205`

- Tesla Cars

POST `https://owner-api.teslamotors.com/api/1/vehicles/vehicle_id/command/flash_lights`

# Example – newsapi.org

`http://newsapi.org/v2/everything?`

`q=tesla&from=2023-02-01&sortBy=publishedAt&apiKey=API_KEY`

```
{
  "status": "ok",
  "totalResults": 2400,
  "articles": [
    {
      "source": {
        "id": null,
        "name": "Motley Fool"
      },
      "author": "newsfeedback@fool.com (The Daily Upside)",
      "title": "America's Push for EVs Could Leave the Power Grid Feeling Drained",
      "description": "Electric vehicles existed in the early 1900s, believe it or not -- ...",
      "url": "https://www.fool.com/investing/2023/02/05/americas-push-for-evs-could-leave-the-power-grid-f/",
      "urlToImage": "https://g.foolcdn.com/editorial/images/719523/featured-daily-upside-image.jpeg",
      "publishedAt": "2023-02-06T01:00:26Z",
      "content": "For more crisp and insightful business and economic news, subscribe to\r\n..."
    },
    ...
  ]
}
```

# REST vs. RPC Interface Paradigms

Example from wikipedia:

## RPC

```
getUser(), addUser(), removeUser(), updateUser(),  
getLocation(), AddLocation(), removeLocation()  
exampleObject = new ExampleApp("example.com:1234");  
exampleObject.getUser();
```

## REST

```
http://example.com/users  
http://example.com/users/{user}  
http://example.com/locations  
userResource = new Resource("http://example.com/users/001");  
userResource.get();
```

# Improving browser interactivity: AJAX

# Web Clients: AJAX

- **A**synchronous **J**avaScript **A**nd **X**ML
  - Bring web services to web clients (JavaScript)
- Asynchronous
  - Client not blocked while waiting for result
- JavaScript
  - Request can be invoked from JavaScript (using **XMLHttpRequest**)
  - JavaScript may also modify the Document Object Model (DOM): the elements of the page: content, attributes, styles, events
- XML
  - Data sent & received as JSON or XML

# AJAX & XMLHttpRequest

- Allow Javascript to make HTTP requests and process results (change page without refresh)

```
var ajax = new XMLHttpRequest();  
ajax.onreadystatechange = function() {  
    // stuff to do when the request is ready  
}  
ajax.open("GET", "http://poopybrain.com/stuff.txt", true);  
ajax.send();
```

- Tell object:
  - Type of request you're making, URL to request
  - Function to call when request is made



# AJAX on the Web

- AJAX ushered in Web 2.0 – responsive web pages
- Early high-profile AJAX sites:
  - Microsoft Outlook Web Access, Gmail, Google Maps, Writely (Google Docs), ...

# The End

# The End

# The End