Web Services & Asynchronous Programming

CS 475, Spring 2018
Concurrent & Distributed Systems
HW2 Discussion

HW2 Submissions per day, as of Wed Feb 21 11:31:42 2018

Days until assignment is due

Submissions per day

-24 -23 -22 -21 -20 -19 -18 -17 -16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1
HW2 Discussion

HW2 Submissions per-student, as of Wed Feb 21 11:32:11 2018

# of submissions per-student
Review: Remote Procedure Calls

```
addPerson("Prof Bell","ENGR 4422");
```

Client Stub

```
addPerson("Prof Bell","ENGR 4422");
```

Server Stub

RPC Magic

Address Book

Client

Server
Review: Shared Fate

• Two methods/threads/processes running on the same computer generally have **shared fate**
• They will either both crash, or neither will crash
Review: Split Brain

• When two machines in a distributed system can’t talk to each other, they might start believing different things
• Two sides can not reconcile view of world because they can’t talk to each other
• We call this a **split brain** problem
Review: RPC

• Procedure calls
  • Simple way to pass control and data
  • Elegant transparent way to distribute application
  • Not only way…

• Hard to provide true transparency
  • Failures
  • Performance
  • Memory access
  • Etc.

• How to deal with hard problem: give up and let programmer deal with it
Announcements

• HW3 is out!
  • Handout will be posted Friday

• Today: Web Services
  • FYI - XML/RPC + SOAP
  • (For real) REST
Announcements

• FYI - What are you doing next year?
  • Not graduating yet: You’ve almost reached the end of the line for classes - Research?
  • Graduating: Consider the opportunity to graduate again?
    • Great new-ish program at GMU:
      • Do you have a GPA of 3.30?
      • You are pretty much guaranteed to be admitted to MS-CS!
RPC on the Web

• How do we do RPC on the web?
• Challenges for scaling up (more clients) and out (heterogeneous clients)
  • Need to get beyond RMI (it’s Java only)
  • How do we find API endpoints?
  • How do we format requests?
  • How do we encode data?
Web Services

- At a high level: any application that invokes computation via the Web
- Several standards:
  - XML/RPC
  - SOAP
  - REST
- All are implemented over HTTP as a communication protocol

```
XML/RPC or SOAP or REST
HTTP
TCP
Network layer
Link layer
```
XML (Extensible Markup Language)

• For a long time, the standard solution for describing information exchange in heterogeneous systems

• XML documents have elements, and elements are demarcated with tags

```xml
<AccountList>
  <Account> 729-1269-4785 </Account>
  <Account type="checking"> 729-1269-4785 </Account>
</AccountList>
```

• **Markup language** like HTML but not simply for displaying pages

• Can be read by programs and interpreted in an application-specific way
JSON (JavaScript Object Notation)

- XML is a markup language, with a schema
- JSON is instead a **data interchange format**
- Only specifies how to represent objects as strings
  
  ```json
  { 
      "accounts": [{ 
          "id": "729-1269-4785"
        }, { 
          "id": "729-1269-4785",
          "type": "checking"
        }]
  }
  
  
  - Less verbose than XML
    - Less bytes to send the same data
    - Usually faster to parse
XML/RPC

- A specification for generic RPC, using XML as an interchange format

```xml
<?xml version="1.0"?>
<methodCall>
    <methodName>SumAndDifference</methodName>
    <params>
        <param>
            <value>
                <i4>40</i4>
            </value>
        </param>
        <param>
            <value>
                <i4>10</i4>
            </value>
        </param>
    </params>
</methodCall>
```

- Recall - XML is a markup language — tags and parameters

- Protocols (like in this case, XML/RPC) define what tags mean (e.g. methodCall)
XML/RPC

- Very simple specification
  - [http://xmlrpc.scripting.com/spec.html](http://xmlrpc.scripting.com/spec.html) (it’s ~ 2 pages)
- Does not have a standard way to specify interfaces or generate stubs
  - Compare to: RMI @Remote interfaces
- No standard for extending protocol, adding authentication, sessions, etc
SOAP

- Written in XML
- Extension to XML-RPC
- Defines mechanism to pass commands and parameters for RPC (like XML-RPC)
- Also defines standard for describing the services and interfaces (WSDL, or Web Service Definition Language)
- WSDL can be used to automatically generate stubs for client/server
WSDL

- Written in XML
- Defines a web services:
  - Operations offered by the service (what)
  - Mechanisms to access the service (how)
  - Location of the service (where)

```xml
<definitions name="MyService">
  <types>data types used</types>
  <message>parameters used</message>
  <portType>set of operations performed</portType>
  <binding>communication protocols and data formats used</binding>
  <service>set of ports to service provider endpoints</service>
</definitions>
```
SOAP

- SOAP protocol defines how RPC are sent over a network
- WSDL defines how a given service uses SOAP
- SOAP packs messages into an envelope with a header and body
- Envelope abstraction allows SOAP extensions to do more stuff (authentication, etc)

```
env:envelope (env means this is part of the SOAP description)
  env:header
    relmsg:sequence (relmsg means part of a reliable message component)
      relmsg:messagid
        143
  env:body
    m:exchange (m means this is part of the service)
      m:arg 1
        Hello
      m:arg 2
        World
```
Web Services Standards Overview

Interoperability Issues

Business Process Specifications

Management Specifications

Presentation Specifications

Metadata Specifications

Reliability Specifications

Security Specifications

Transaction Specifications

Resource Specifications

Messaging Specifications

SOAP

XML Specifications

innoQ
SOAP

- SOAP has LOTS of extensions (60+)
  - Reliable messaging
  - Security
  - Addressing
  - Transactions
- SOAP supports a lot of complexity **in the protocol itself**
- Problem: just to get a minimal, small example working, you need to do a lot of boilerplate
REST: REpresentational State Transfer

• Defined by Roy Fielding in his 2000 Ph.D. dissertation
• “Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do... I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience. That process honed my model down to a core set of principles, properties, and constraints that are now called REST.”
• Interfaces that follow REST principles are called RESTful
Properties of REST

• Performance
• Scalability
• Simplicity of a Uniform Interface
• Modifiability of components (even at runtime)
• Visibility of communication between components by service agents
• Portability of components by moving program code with data
• Reliability
Principles of REST

- Client server: separation of concerns (reuse)
- Stateless: each client request contains all information necessary to service request (scaling)
- Cacheable: clients and intermediaries may cache responses. (scaling)
- Layered system: client cannot determine if it is connected to end server or intermediary along the way. (scaling)
- Uniform interface for resources: a single uniform interface (URIs) simplifies and decouples architecture (change & reuse)
Uniform Interface for Resources

• Originally files on a web server
  • URL refers to directory path and file of a resource
• But… URIs might be used as an identity for any entity
  • A person, location, place, item, tweet, email, detail view, like
  • *Does not matter* if resource is a file, an entry in a database, retrieved from another server, or computed by the server on demand
• Resources offer an *interface* to the server describing the resources with which clients can interact
URI: Universal Resource Identifier

- Uniquely describes a resource
  - https://mail.google.com/mail/u/0/#inbox/157d5fb795159ac0
  - https://www.amazon.com/gp/yourstore/home/ref=nav_cs_yo
    StefanTilkov_RESTIDontThinkItMeansWhatYouThinkItDoes.pdf
- Which is a file, external web service request, or stored in a database?
  - It does not matter
- As client, only matters what actions we can do with resource, not how resource is represented on server
HTTP Actions

- Idea: define operations by using existing HTTP action verbs
- Describes what will be done with resource
  - GET: retrieve the current state of the resource
  - PUT: modify the state of a resource
  - DELETE: clear a resource
  - POST: initialize the state of a new resource
URI Design

• In theory, URI could last forever, being reused as server is rearchitected, new features are added, or even whole technology stack is replaced.

• “What makes a cool URI? A cool URI is one which does not change. What sorts of URIs change? URIs don't change: people change them.”
  • https://www.w3.org/Provider/Style/URI.html
  • Bad:  
    • https://www.w3.org/Content/id/50/URI.html (What does this path mean? What if we wanted to change it to mean something else?)

• Why might URIs change?
  • We reorganized our website to make it better.
  • We used to use a cgi script and now we use node.JS.
URI Design

- URIs represent a contract about what resources your server exposes and what can be done with them
- Leave out anything that might change
  - Content author names, status of content, other keys that might change
  - File name extensions: response describes content type through MIME header not extension (e.g., .jpg, .mp3, .pdf)
  - Server technology: should not reference technology (e.g., .cfm, .jsp)
- Endeavor to make all changes backwards compatible
  - Add new resources and actions rather than remove old
- If you must change URI structure, support old URI structure and new URI structure
Example URI Design

• The candy web service!
• Tracks information about candy
• http://api.jonbell.net/candy/twix
  • GET this URI to find out about twix bar
  • POST to the URI to set up a new twix bar
  • DELETE this URI to eat a twix
Describing Responses

• What happens if something goes wrong while handling HTTP request?
  • How does client know what happened and what to try next?
• HTTP offers response status codes describing the nature of the response
  • 1xx Informational: Request received, continuing
  • 2xx Success: Request received, understood, accepted, processed
    • 200: OK
  • 3xx Redirection: Client must take additional action to complete request
    • 301: Moved Permanently
    • 307: Temporary Redirect

Describing Errors

• 4xx Client Error: client did not make a valid request to server. Examples:
  • 400 Bad request (e.g., malformed syntax)
  • 403 Forbidden: client lacks necessary permissions
  • 404 Not found
  • 405 Method Not Allowed: specified HTTP action not allowed for resource
  • 408 Request Timeout: server timed out waiting for a request
  • 410 Gone: Resource has been intentionally removed and will not return
  • 429 Too Many Requests
Describing Errors

- 5xx Server Error: The server failed to fulfill an apparently valid request.
  - 500 Internal Server Error: generic error message
  - 501 Not Implemented
  - 503 Service Unavailable: server is currently unavailable
Async programming

• Interacting with web services begs for asynchronous programming

• Example:
  • Get a list from GitHub's API of the top 10,000 Java projects
  • As that list starts coming in, start requesting information on each commit of each of those projects
  • As that information starts coming in, request more information on each commit
Async Programming

- We probably **really** want to do this concurrently, but implementing it is tricky
- This is what promises are made for!!!
Async Programming Activity

• Case study (not GitHub): the Candy API
Async Programming Activity

Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar

thenCombine

Group all Twix
Group all 3 Musketeers
Group all MilkyWay
Group all MilkyWay Dark
Group all Snickers

when done

Eat all the Twix

in case of exception

Ask Prof Bell what to do