CSE 124
Distributed programming and Remote Procedure Calls (RPC)

February 24, 2016, UCSD
Prof. George Porter
Today

• HW 5 out
  – Simple questions on datacenters
• Project 2
  – Deadline extended to last day of quarter
• Today:
  – RPC background and theory
  – REST
High-level overview
What makes RPC hard?

• Network vs. computer backplane
  – Message loss, message reorder, ...

• Heterogeneity
  – Client and Server might have different:
    • OS versions
    • Languages
    • Endian-ness
    • Hardware architectures
    • ...

Leslie Lamport

“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable”
Overview

• RPC Protocol itself
• Stub compiler / marshalling
• RPC Frameworks
Remote Procedure Call Issues

• Underlying protocol
  – UDP vs. TCP
  – Advantages/Disadvantages?

• Semantics
  – What happens on network/host failure?

• Transparency
  – Hide network communication/failure from programmer
  – With language support, can make remote procedure call “look just like” local procedure call
Identifying remote function/methods

• Namespace vs flat
  – Issues w.r.t. flat: how to assign?

• Matching requests with responses
  – What if the client or server fail during execution?

• Main idea:
  – Message ID
  – Client blocked until response returned w/ Message ID
When things go wrong

- Let’s assume client and server use TCP
- Client issues request(0)
- Client fails, reboots
- Client issues unrelated request(0)
- Server gets first request(0), executes it, send response
- Server get the second request, thinks it is a duplicate, and so ACKs it but doesn’t execute
- Client never gets a response to either request.
Boot ID + Message ID

• Idea is to keep non-volatile state that is updated every time the machine boots
• Incremented during the boot-up process
• Each request/response is uniquely identified by a (bootID,MessageID) pair
Client/Server Interaction

Client sends request
- Client waits for response (blocking)

Server wakes up
- Computes, sends response
- Response wakes up client

Server structure?
- Process, thread, FIFO
Reliable RPC (explicit acknowledgement)
Reliable RPC (implicit acknowledgement)
What about failures?

- Different options:
  1. Server can keep track of the (BootId, MessageId) and ignore any duplicate requests
  2. Server doesn’t keep state—might execute duplicate requests

- What are advantages/disadvantages?
## RPC Semantics

<table>
<thead>
<tr>
<th>Delivery Guarantees</th>
<th>RPC Call Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Retry Request</strong></td>
<td><strong>Duplicate Filtering</strong></td>
</tr>
<tr>
<td>No</td>
<td>NA</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Remote Procedure Call Issues

• Idempotent operations
  – Can you re-execute operations without harmful side effects
  – Idempotency means that executing the same procedure does not have visible side effects

• Timeout value
  – How long to wait before re-transmitting request?
Protocol-to-Protocol Interface

- Send/Receive semantics
  - Synchronous vs. Asynchronous

- Process Model
  - Single process vs. multiple process
  - Avoid context switches

- Buffer Model
  - Avoid data copies
Part 2: RPC Implementations
• Client stub indicates which procedure should run at server
  – Marshals arguments
• Server stub unmarshals arguments
  – Big switch statement to determine local procedure to run
Presentation Formatting

- Marshalling (encoding) application data into messages
- Unmarshalling (decoding) messages into application data

- Data types to consider:
  - integers
  - floats
  - strings
  - arrays
  - structs

Types of data we do not consider
- images
- video
- multimedia documents
Difficulties

- Representation of base types
  - Floating point: IEEE 754 versus non-standard
  - Integer: big-endian versus little-endian (e.g., 34,677,374)

```
<table>
<thead>
<tr>
<th>Big-endian</th>
<th>(2)</th>
<th>(17)</th>
<th>(34)</th>
<th>(126)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00000010</td>
<td>00010001</td>
<td>00100010</td>
<td>01111110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little-endian</th>
<th>(126)</th>
<th>(34)</th>
<th>(17)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01111110</td>
<td>00100010</td>
<td>00010001</td>
<td>00000010</td>
</tr>
</tbody>
</table>
```

- Compiler layout of structures
Taxonomy

- Data types
  - Base types (e.g., ints, floats); must convert
  - Flat types (e.g., structures, arrays); must pack
  - Complex types (e.g., pointers); must linearize
RPC Interface vs. Implementation

- **RPC Interface:**
  - High-level procedure invocation with arguments, return type
  - Asynchronous, Synchronous, ‘void’, Pipelined...

- **RPC Implementation:**
  - SunRPC
  - Java RMI
  - XML RPC
  - Apache Thrift
  - Google Protocol Buffers*
  - Apache Avro
  - REST (REpresentational State Transfer)
SunRPC

• Originally implemented for popular NFS (network file service)
• XID (transaction id) uniquely identifies request
• Server does not remember last XID it serviced
  – Problem if client retransmits request while reply is in transit
  – Provides at-least once semantics
• Port mapper service
XML-RPC

- XML is a standard for describing structured documents
  - Uses tags to define structure: `<tag> … </tag>` demarcates an element
    - Tags have no predefined semantics …
    - … except when document refers to a specific namespace
  - Elements can have attributes, which are encoded as name-value pairs
    - A well-formed XML document corresponds to an element tree
- `<?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>`

(thanks to Vijay Karamcheti)
XML-RPC Wire Format

• Scalar values
  – Represented by a `<value><type> ... </type></value>` block
• Integer
  – `<i4>12</i4>`
• Boolean
  – `<boolean>0</boolean>`
• String
  – `<string>Hello world</string>`
• Double
  – `<double>11.4368</double>`
• Also Base64 (binary), DateTime, etc.
XML-RPC Wire Format (struct)

- **Structures**
  - Represented as a set of `<member>`s
  - Each member contains a `<name>` and a `<value>`

```xml
<struct>
  <member>
    <name>lowerBound</name>
    <value><i4>18</i4></value>
  </member>
  <member>
    <name>upperBound</name>
    <value><i4>139</i4></value>
  </member>
</struct>
```
XML-RPC Wire Format (Arrays)

- Arrays
  - A single `<data>` element, which
  - contains any number of `<value>` elements
- `<array>`
  
  ```xml
  <data>
  <value><i4>12</i4></value>
  <value><string>Egypt</string></value>
  <value><boolean>0</boolean></value>
  <value><i4>-31</i4></value>
  </data>
  </array>
  ```
XML-RPC Request

- HTTP POST message
  - URI interpreted in an implementation-specific fashion
  - Method name passed to the server program

- POST /RPC2 HTTP/1.1
  Content-Type: text/xml
  User-Agent: XML-RPC.NET
  Content-Length: 278
  Expect: 100-continue
  Connection: Keep-Alive
  Host: localhost:8080

  <?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>
XML-RPC Response

- HTTP Response
  - Lower-level error returned as an HTTP error code
  - Application-level errors returned as a `<fault>` element (next slide)

- HTTP/1.1 200 OK
  Date: Mon, 22 Sep 2003 21:52:34 GMT
  Server: Microsoft-IIS/6.0
  Content-Type: text/xml
  Content-Length: 467
  <?xml version="1.0"?>
  <methodResponse>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>sum</name>
            <value>
              <i4>50</i4>
            </value>
          </member>
          <member>
            <name>diff</name>
            <value>
              <i4>30</i4>
            </value>
          </member>
        </struct>
      </value>
    </param>
  </params>
  </methodResponse>
XML-RPC Fault Handling

- Another kind of a MethodResponse
- `<xml version="1.0"?>
  <methodResponse>
    <fault>
      <value><struct>
        <member>
          <name>faultCode</name>
          <value><i4>500</i4></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Arg `a’ out of range</string></value>
        </member>
      </struct></value>
    </fault>
  </methodResponse>`
RMI Architecture

Java Client

Invoke Method A on Object B

Stub Object B

Distributed Computing Services

Java Server

Object B Method A

Skeleton Object B

Distributed Computing Services

RMI Object Registry
Maps object names to locations

RMI Transport Protocol

(thanks to David Del Vecchio)
RMI Example (Server)

Hello.java

```java
import java.rmi.*;

public interface HelloInterface extends Remote {
    public String say() throws RemoteException;
}
```

HelloInterface.java

```java
import java.rmi.*; import java.rmi.server.*;
public class Hello extends UnicastRemoteObject implements HelloInterface {
    private String message;

    public Hello (String msg) throws RemoteException { message = msg; }

    public String say() throws RemoteException {
        return message;
    }
}
```

```java
public static void main (String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    try {
        Naming.bind ("Hello", new Hello ("Hello, world!");
    } catch (Exception e) { System.out.println ("Server failed"); }
}
```
RMI Example (Client)

HelloClient.java

```java
import java.rmi.*;

public class HelloClient {
    public static void main (String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            HelloInterface hello = (HelloInterface) Naming.lookup("//(hostname)/Hello");
            System.out.println(hello.say());
        } catch (Exception e) {
            System.out.println("HelloClient exception: "+ e);
        }
    }
}
```

- The rmic utility generates stubs and skeletons
- Once started, server will stay active
- RMI daemon (rmid) can be used to create (activate) objects on the fly
Part 3: REST
(REpresentational State Transfer)
Background

• Roy Fielding Ph.D. dissertation, 2000
  – “Architectural Styles and the Design of Network-Based Software Architectures”

• RPC-like interface to web services
  – Based on HTTP, URLs, JSON, XML

• Access and update remote state via HTTP

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Request to read a Web page</td>
</tr>
<tr>
<td>HEAD</td>
<td>Request to read a Web page’s header</td>
</tr>
<tr>
<td>PUT</td>
<td>Request to store a Web page</td>
</tr>
<tr>
<td>POST</td>
<td>Append to a named resource (e.g., a Web page)</td>
</tr>
<tr>
<td>DELETE</td>
<td>Remove the Web page</td>
</tr>
<tr>
<td>TRACE</td>
<td>Echo the incoming request</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Query certain options</td>
</tr>
</tbody>
</table>
HTTP + RPC?

+ ?? = RPC?

- Must represent application state in a way that can be exposed to HTTP
- Idea:
  - Encode state representation as a URL
Representing state as URLs

GET /Addressbook/contacts

```json
[
{
  "id":1,
  "name":"George Porter",
  "phone":"858-561-1234",
  "room number":3104
},
{
  "id":2,
  "name":"Stefan Savage",
  "phone":"858-561-8172",
  "room number":3105
},
...]
```
Representing state as URLs

GET /Addressbook/contact/2

{
  "id":2,
  "name":"Stefan Savage",
  "phone":"858-561-8172",
  "room number":3105
}
Representing state as URLs

POST /Addressbook/contacts

Result:

200 OK

{
"id":3,
"name":"A. Turing",
"phone":"858-561-8212",
"room number":3108
}
REST examples “In the wild”

**URI**

Use the following URI to obtain Contact Objects or add a Contact Object for a user identified by the `guid` value.

https://social.yahooapis.com/v1/user/{guid}/contacts

**HTTP OPERATIONS SUPPORTED**

- GET
- POST
- PUT

**QUERY PARAMETERS SUPPORTED**

- format
- view
- start
- count
Example: Twitter

- Every aspect of twitter is available via REST interfaces
- Data encoded in JSON
  - Tweets, friends, Timelines, ...
- Developer interface to REST API via:
  - https://dev.twitter.com/rest/tools/console