Administrative Details

- Questions about Project 2?
  - **Stateful versus stateless** XML-RPC servers
  - Set your classpath in .local_bashrc
  - Milestone: Mar 6
  - EVERYTHING (code + paper) due: Mar 14

- CoDeen paper due now
Project 2 Overview

- You will make 1 server, 2 clients, and a DB
- Concurrent queries OK, but only one purchase at a time
- Create two simple clients in two different prog languages
- Focus more on stress testing and evaluation
- Questions?
Last Time

• Learned about remote procedure calls
• Focused on XML-RPC as a case study
Today's Outline

• Learn more about remote procedure calls (RPCs)
  • Finish discussing XML-RPC
  • We'll look at Java RMI briefly
  • There are many others: SOAP, SunRPC, etc.
  • Quick discussion about REST

• Talk about content distribution networks
Recap: Remote Procedure Calls (RPCs)

- Distributed (network) programming is challenging
  - Need abstractions to hide complexity
- Initially, network programming involved hand-coded messages – not a very natural programming model!
- In early 1980's, researchers noticed most distributed programming took form of remote procedure call
- Issues to think about when designing RPC library:
  - Choice of underlying transport layer protocol
  - Semantics (i.e., what is the “effect” of executing an RPC?)
  - Transparency (make "remote-ness" invisible to programmer)
  - Role of idempotent operations
Recap: Remote Procedure Call

1. Client calls the client stub, a local procedure call, with parameters for remote procedure
2. Client stub packs the parameters (marshaling) into a message and makes a system call
3. Client's local OS sends the message
4. Server's OS passes incoming msg to the server stub
5. Server stub unpacks the parameters (unmarshaling)
6. Server stub calls the server procedure. (The response traces the same steps in the reverse direction.)
Case Study: Apache XMLRPC

- XML is a standard for describing structured documents
  - Uses tags to define structure:
    - `<tag> ... </tag>` demarcates an element
    - Tags have no predefined semantics ...
  - Elements can have attributes
    - They are encoded as name-value pairs
    - A well-formed XML document corresponds to an element tree
- In XML-RPC, stubs “wrap” messages in XML
- At-least once semantics

```xml
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
  <methodName>sample.SumAndDifference</methodName>
  <params>
    <param><value><i4>10</i4></value></param>
    <param><value><i4>40</i4></value></param>
  </params>
</methodCall>
```
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), Date/Time, etc.
XML-RPC Wire Format (struct)

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

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

- Arrays
  - A single `<data>` element, which contains any number of `<value>` elements (with types)

- `<array>`
  
  `<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

- Runs on top of a “standard” web server (speaks HTTP)
- Messages sent as HTTP POST requests
  - URL interpreted in an implementation-specific fashion
  - Method name and params passed to the server program

POST / HTTP/1.1
Content-Type: text/xml
User-Agent: Apache XML RPC 3.1.3
Content-Length: 245
Connection: Keep-Alive
Host: localhost:8888

<?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: Apache XML-RPC 1.0
Content-Type: text/xml
Content-Length: 198

<?xml version="1.0"?>
<methodResponse>
<params>
<param>
<value>
<array>
<data>
<value><i4>50</i4></value>
<value><i4>30</i4></value>
</data></array>
</value>
</param>
</params>
</methodResponse>
XML-RPC Fault Handling

- Another kind of a MethodResponse

HTTP/1.1 200 OK
Date: Mon, 27 Feb 2023 21:52:34 GMT
Server: Apache XML-RPC 1.0
Content-Type: text/xml
Content-Length: 467

```xml
<?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>
```
From XML-RPC to SOAP

- Limitations of XML-RPC
  - Adds overhead to messages (lots of XML)
  - Allows only one method of serialization (marshaling)
  - Simple/weak security model

- Led to invention of SOAP
  - Simple Object Access Protocol
  - Uses XML, HTTP, and SMTP (email)
  - More general than XML-RPC (and has more features)
    - Can define the serialization (so complex objects can be encoded) – usually use WSDL
    - More elaborate security models supported
**SOAP**

- SOAP is more complicated than XML-RPC
  - Simple Object Access Protocol
  - Uses *namespaces* and objects
  - Formal "envelope" (SOAP header and body)

```xml
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
 <SOAP-ENV:Body>
   <m:GetLastTradePrice xmlns:m="Some-URI">
     <symbol>DIS</symbol>
   </m:GetLastTradePrice>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
REST

• REST (or ReST) stands for Representational State Transfer
  • Relies on a stateless, client-server, cacheable communications protocol
  • Typically runs over HTTP
  • Theoretically simpler than RPC mechanisms
• RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data
• Everything is accomplished using the URL
RPC vs REST

• REST vs RPC implementations like XML-RPC is a false dichotomy
• Can implement a RESTful interface using XML-RPC (although you probably wouldn't want to)
• REST is an *architecture style*, not a protocol
• While there are REST programming frameworks, you usually just “roll your own” using standard language features
Another Alternative: Java RMI

Invoke Method A on Object B

Stub Object B

Distributed Computing Services

RMI Object Registry
Maps object names to locations

Object B Method A

Skeleton Object B

Distributed Computing Services

RMI Transport Protocol
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {
    public Server() {}
    public String sayHello() {  return "Hello, world!";  }
    public static void main(String args[]) {
        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            Registry registry = LocateRegistry.createRegistry(8888);
            registry.bind("Hello", stub);
        } catch (Exception e) { System.err.println("Server exception: " + e.toString()); }
    }
}

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}

Server.java
Hello.java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    private Client() {}
    public static void main(String[] args) {
        String host = (args.length < 1) ? "localhost" : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host, 8888);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) { System.err.println("Client exception: " + e.toString()); } 
    }
}
Java RMI vs. XML-RPC

• Java RMI is arguably simpler
  • Programs look a bit more "normal"
  • Can serialize (by implementing Serializable) and return different objects
  • Only works with Java

• XML-RPC is more flexible
  • Can interact (easily) with other XML-RPC clients written in different languages
  • But procedure calls are somewhat limiting
  • Difficult to send “non-standard” objects
Take-Away Message

- RPCs handle the marshaling and unmarshaling of data automatically
- Also handle threading, message formatting, socket creation, etc.
- Designed to simplify network programming through familiar programmatic abstractions
- You should appreciate all of this after building your web servers!
Moving on...
Content Distribution Networks

• Proactive content replication
  • Content provider (e.g., CNN) contracts with a CDN

• CDN replicates the content
  • On many servers spread throughout the Internet

• Updating the replicas
  • Updates pushed to replicas when the content changes
Server Selection Policy

- Live server
  - For availability
- Lowest load
  - To balance load across the servers
- Closest
  - Nearest geographically, or in round-trip time
- Best performance
  - Throughput, latency, …
- Cheapest bandwidth, electricity, …

Requires continuous monitoring of liveness, load, and performance
Server Selection Mechanism

- **Application**
  - HTTP redirection

- **Advantages**
  - Fine-grain control
  - Selection based on client IP address

- **Disadvantages**
  - Extra round-trips for TCP connection to server
  - Overhead on the server
Server Selection Mechanism

• Naming
  • DNS-based server selection

• Advantages
  • Avoid TCP set-up delay
  • DNS caching reduces overhead
  • Relatively fine control

Disadvantage
• Based on IP address of local DNS server
• "Hidden load" effect
  • How many clients are represented by “local DNS server?”
• DNS TTL limits adaptation
CDN Example – Akamai

Akamai:
• Creates new domain names (CNAMEs or canonical names) for each client content provider
• CNAMEs map hostnames to other hostnames in DNS
  • e.g., i.a.cnn.net = CNAME custom.i.cnn.net.edgesuite.net
  custom.i.cnn.net.edgesuite.net = CNAME a1921.g.akamai.net

Customer/Content Provider:
• Modifies content
  • Embedded URLs reference new domains
  • CNN page's HREFs refer to i.a.cnn.net
• Pushes content out to Akamai as it changes
  • Or: Akamai pulls it on demand w/ usual caching mech.
  • Both CNN & Akamai have control over load distribution
How Akamai Works

• Clients fetch html document from primary server (CNN)
  • E.g. fetch index.html from cnn.com

• URLs for replicated content are replaced in html
  • E.g. `<img src="http://cnn.com/af/x.gif">` replaced with `<img src="http://a73.g.akamaitech.net/7/23/cnn.com/af/x.gif">`

• Client is forced to resolve aXYZ.g.akamaitech.net hostname
"Akamaized" response object has inline URLs for secondary content at (after resolving CNAMEs) a73.g.akamaitech.net and other Akamai-managed DNS names.

GET http://cnn.com
1 - DNS Lookup
2 - Fetch page w/ "Akamaized" content
3 - DNS Lookup for Akamai URLs
4 - Fetch content
How Akamai Works

• How is content replicated in CDN?
• Akamai only replicates static content (*)
• Modified name contains original file name
• Akamai server is asked for content
  • First checks local cache
  • If not in cache, requests file from primary server and caches file

* (At least, the version we're talking about today. Akamai actually lets sites write code that can run on Akamai's servers, but that's a pretty different beast)
How Akamai Works

1. End-user requests cnn.com (content provider).
2. CNN.com sends DNS request to root DNS server.
3. Root DNS server sends request to high-level Akamai DNS server.
4. High-level Akamai DNS server sends request to low-level Akamai DNS server.
5. Low-level Akamai DNS server sends request to nearby matching Akamai server.
6. Nearby matching Akamai server sends request back to low-level Akamai DNS server.
7. Low-level Akamai DNS server sends request back to high-level Akamai DNS server.
8. High-level Akamai DNS server sends request back to root DNS server.
10. CNN.com sends the response to the end-user.
11. End-user sends request for foo.jpg.
12. End-user gets foo.jpg from Akamai server.
Akamai – Subsequent Requests

1. User requests "index.html" from cnn.com
2. DNS root server resolves cnn.com to an Akamai server IP
3. Akamai high-level DNS server contacts low-level DNS server
4. Nearby matching Akamai server answers request
5. User receives "index.html"
6. User requests "/cnn.com/foo.jpg"
Impact on DNS Usage

- DNS is increasingly being used for server selection
  - What are reasonable DNS TTLs for this type of use
  - Typically want to adapt to load changes
- How does this affect caching?
Web Proxy Caches

- User configures browser: Web accesses via cache
- Browser sends all HTTP requests to cache
  - Object in cache: cache returns object
  - Else cache requests object from origin server, then returns object to client
Caching Example (1)

Assumptions

- Average object size = 100,000 bits
- Avg. request rate from institution's browser to origin servers = 15/sec
- Delay from institutional router to any origin server and back to router = 2 sec

Consequences

- Utilization on LAN = 15%
- Utilization on access link = 100%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + milliseconds
Caching Example (2)

Possible solution
• Increase bandwidth of access link to, say, 10 Mbps
• Often a costly upgrade

Consequences
• Utilization on LAN = 15%
• Utilization on access link = 15%
• Total delay = Internet delay + access delay + LAN delay
  = 2 sec + msecs + msecs
Caching Example (3)

**Install cache**
- Suppose hit rate is .4

**Consequence**
- 40% requests will be satisfied almost immediately (say 10 msec)
- 60% requests satisfied by origin server
- Utilization of access link reduced to 60%, resulting in negligible delays
- Weighted average of delays
  \[ = .6 \times 2 \text{ sec} + .4 \times 10 \text{msec} < 1.3 \text{ secs} \]
Problems

- Many (most?) HTTP objects are uncachable – why?
- Not easily solvable
  - Dynamic data → stock prices, scores, web cams, social media, etc
  - CGI/PHP scripts → results based on passed parameters
- Problems with obvious fixes
  - SSL → encrypted data is not cacheable
    - Most web clients don't handle mixed pages well → many generic objects transferred with SSL
  - Cookies → results may be based on passed data
Reliability and Security in the CoDeeN Content Distribution Network

Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek Pai, Larry Peterson

Princeton University
Content Distribution Networks

- Replicates Web content broadly
- Redirects clients to "best" copy
  - Load, locality, proximity
- Offloads work from origin servers
- Multiplexes load spikes
  - Reduces overprovisioning
- Ex: Akamai, Mirror Image, Speedera
What Is CoDeeN?

- Academic Content Distribution Network
- Forward/reverse proxies, redirector
- 100+ proxy servers on PlanetLab (in 2004)
- Continuous service, decentralized control
- Deployed for getting real traffic
Who Is The Target Audience?

• Now (2003)
  • Users wanting better performance
  • People seeking "anonymity"

• Next
  • Content providers seeking load sharing

• Later
  • General support for absorbing flash crowds
  • Avoid the "Slashdot Effect"
How Does It Work?

- Server surrogates (proxies) on most North American sites
  - Originally everywhere, but we cut back
- Clients specify proxy to use
  - Cache hits served locally
  - Cache misses forwarded to CoDeeN nodes
    - Maybe forwarded to origin servers
Goals of CoDeeN

• Provide open content distribution
• Improve web performance & reliability
• Platform for testing new innovations
  • Particularly in live environments
• Keep CoDeeN running 24/7
  • Security
  • Reliability
How Does CoDeeN Work?

Each CoDeeN proxy is a forward proxy, reverse proxy, & redirector.
Types of Security Problems

- Spammers
- Bandwidth hogs
- High request rates
- Content thieves
- Worrisome anonymity

Commonality: using CoDeeN to do things they would not do directly
The Root of All Trouble

origin

http/tcp

CoDeeN Proxy

http/tcp

(Malicious) Client

No End-To-End Authentication
Approaches to Security

- Desired: allow only "safe" accesses
- No research in "partially open" proxies
- Our approach
  - Rate limiting
  - Privilege separation
Rate Limiting

- 3 scales capture burstiness
- Exceptions
  - Login attempts
  - Vulnerability tests
  - Repetition, request spreading
Privilege Separation

Remote Client

Site A Client

Site B Proxy

Unprivileged Request

Site A Proxy

Privileged Request

Site A Server
Thoughts?