Servers
MIT 6.170 – 22\textsuperscript{nd} September 2014
Adam Chlipala, Mark Day, Daniel Jackson
Lectures are about big ideas

- Space
- Time
Contrasting with previous material

- Local, transient computation
  - Always running in browser
  - Little or no state retained between executions
- Now adding *distributed* computation
  - Spanning space
- ... Next adding *persistent* computation
  - Spanning time
Structure of next two lectures

• Today
  – Servers
  – Why servers?
  – How to build with Node.js

• Wednesday
  – Persistent servers
  – Why persistent?
  – How to build with Node.js + Mongo DB
This lecture

• Why servers?
  – Modularity revisited
  – When servers are / aren’t the right solution

• Web servers and services
  – How the plumbing works
  – Why it works that way: technology + history

• Routing: “matching” web operations with application calls
Next lecture

• Revisit servers from this lecture
  – Adding in databases!
  – Requires explaining/showing database
  – New set of “matching” issues
    • Now, between application ops and database ops
An alternative perspective

- From Fox & Patterson, *Engineering Software as a Service*

<table>
<thead>
<tr>
<th>“Height”</th>
<th>Concept(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000 feet</td>
<td>Client-server</td>
</tr>
<tr>
<td>50,000 feet</td>
<td>HTTP and URIs</td>
</tr>
<tr>
<td>10,000 feet</td>
<td>HTML and CSS</td>
</tr>
<tr>
<td>5,000 feet</td>
<td>3-tier and horizontal scaling</td>
</tr>
<tr>
<td>1,000 feet</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>500 feet</td>
<td>Active Record</td>
</tr>
</tbody>
</table>
WHY SERVERS?
Programs in a browser
What could go wrong?
Long-running or infinite loop event handler?

• Everything stuck
• Achilles heel of all "cooperative multithreading"

• Works OK when all code is controlled by one party – not so good cross-domain
What’s a server?

• Client and server interact only via messages
  – No shared memory (thus, no pointer passing)
  – No global variables
  – Separate execution environments
• Message exchanges define services offered by server
  – Typically offering same service to many clients
  – Predictable, consistent
High level model

Web browser

Internet

Web site
Servers, who cares?

• We already have objects
  – Private state
  – Well-defined methods

• Isn’t that enough?
Nobody’s perfect

• Errors in mostly-well-built code
  – E.g., omitting “var” is enough to create accidental cross-module sharing

• Some code isn’t well-built
  – Your code might be using it
  – It might be using your code
  – Good to have “enforced modularity”

• We might call this “voluntary use” of servers
“Voluntary” use of servers

• Choosing to separate more strongly
• Avoiding cooperation
• Protecting vs. untrusted parties
Other reasons...
Conversation: unaided vs. telecom

• Limits to how far you can make face-to-face work
  – Probably 20 feet as upper bound?
• Phone has limits too... but much longer distance
  – Main problem is propagation delay (geosync satellites) 44,000+ miles
  – Land lines easily cover 13,000+ miles with no problem
  – Note: Earth circumference ~25,000 miles
• Tradeoff: distance gains vs. expressiveness losses
• So what? Well...
Distance and autonomy

• Some stuff is far away
• Some stuff is outside your control
• That has implications for how you deal with that stuff
“Involuntary” use of servers

• Use a resource that’s not locally available
• Share a resource with non-local users
• Protect a resource from user errors
• Design a resource for scaling up to massive levels
• Common element: can’t really build with “shared variables” approach
Implications of distance and autonomy

• The remote side may not respond
  – In general, can’t tell if it’s alive or not
• The remote side may fail without notice
  – Distributed systems have “partial failure”
  – The whole Internet is *always* partially failed... but no-one complains
Design guidelines: adding server(s)

• When do you build a server?
  – When you have to ("involuntary")
  – What if everything could be built with ordinary objects?
    • Anticipate future sharing / concurrent access
    • Anticipate future remote access
    • Defend against unknown/untrustworthy users
    • Anticipate future scale-up
Design guidelines: deleting server(s)

• When do you eliminate a server?
  – Separation is meaningless: neither side of the app can function without the other
  – Trust model changes: no longer concerned about misbehavior
Programming problems

• New challenges
  – Accept incoming requests
  – Accept concurrent requests
  – Recover from unexpected conditions

• Programming idioms
  – Callbacks to handle incoming requests
  – Event loop model for concurrency
  – Catching exceptions and restoring order
Demo: Hello World Web Server
Hello World web server

• Easy to do
• Lots of complexity hidden
• We’ll revisit the complexity
Main idiom

• Create server with function
  – Function is called when request is received
  – Function is handed both the incoming request and the outgoing result
Demo: Hello World Telnet Server
Hello World telnet server

• Believe it or not, this is what the Internet was originally built for
• We’ll develop this a little further
  – See interaction/modularity issues with less overhead than a web server
Main idiom

• Network connection is event emitter
• We can add a listener for an event
  – Event has a string name
  – Listener is just a function
  – OK to have more than one listener
  – When the event occurs, any associated listeners are called
Failures are an issue

- More important in server context
- Server may provide shared service to many users
- Server may provide access to critical resource
Demo: Failing server & fixed
Demo: Web Server fronts Telnet Server
Web server in front of telnet server

• Don’t have to build everything into web server
• Can “webify” a resource
• A tiny version of how large scale apps are built in multiple tiers
  – Manageability
  – Separation of concerns
  – Scale up
Three-tier architecture
Alternative view
Multi-tier architectures

• Not a law of nature! Just an organizing principle.
• In large applications, each tier may consist of multiple servers
• We’ll revisit in next lecture
  – Effectively adding a database tier to these examples
Demo: Telnet with Replaceable Prefix
Telnet server with replaceable prefix

• Client operation changes long-term behavior (outside current call)
• Local state... still nothing very special
  – “Big deal!” We could have achieved much the same effect with objects and local calls
• This will be our starting point for the next lecture, when we make the local state persistent
Demo: Telnet with replaceable prefix, multiple clients
Multiple clients using shared telnet server

• One client’s actions affect what another client sees
• The start of something new (not just a weird modularity trick)
• Can now have 1 client change something and n others are affected
• Or have each of them affect the others
• Hint: This is the core of what you will have to do in P2
Multi-prefix telnet server

- Not just a single mutable item
- List might grow to arbitrary size
- Now need to be able to manipulate in more ways:
  - Create a new prefix
  - Read how many prefixes there are
  - Remove a prefix
  - Remove all prefixes (reset)
Servers as concept

• Multiple uses – be clear about what purposes your server(s) accomplish
• Node and JavaScript let us build servers with familiar tricks
  – Closures for code to be executed “when something happens” – like a request
  – Catching exception objects to survive problems
WEB SERVERS AND SERVICES
Understanding the Web

• Origins
  – Straightforward, but sometimes surprising

• Technology
  – Straightforward, but sometimes surprising

• Evolution & Future
  – Hard to characterize
Roots of the Web

• “Hypertext”
  – Vannevar Bush “Memex” 1945
  – Ted Nelson “Xanadu” 1960
  – Doug Engelbart “NLS” “Mother of all Demos” 1968

• Networks + storage + linking
Information Management: A Proposal

Tim Berners-Lee, CERN
March 1989, May 1990

This proposal concerns the management of general information about accelerators and experiments at CERN. It discusses the problems of loss of information about complex evolving systems and derives a solution based on a distributed hypertext system.

Overview

Many of the discussions of the future at CERN and the LHC era end with the question - “Yes, but how will we ever keep track of such a large project?” This proposal provides an answer to such questions. Firstly, it discusses the problem of information access at CERN. Then, it introduces the idea of linked information systems, and compares them with less flexible ways of finding information.
Invention of Web

- Tim Berners-Lee at CERN, 1989
- [http://www.w3.org/History/1989/proposal.html](http://www.w3.org/History/1989/proposal.html)
- “Yes, but how will we ever keep track of such a large project?”
Key elements

• Uniform Resource Indicator (URI)
  – Consistent identifier for generic “resource”

• HyperText Markup Language (HTML)
  – Express structure of document
  – Crucially, includes “anchor” element that can include URI to link to other resource

• HyperText Transport Protocol (HTTP)
  – Let browser and servers communicate so that browser can navigate universe of resources
URL vs. URI vs. ...

- **URL**: Locator
  - “web address”
  - Very common

- **URN**: Name
  - Stable despite location change
  - Very uncommon

- **URI**
  - Effectively synonymous with URL
Elements of URI

- http://localhost:2457/
- http://www.example.com/foo/bar?lang=en&order=up#top
- http://www.example.com/foo/bar?lang=en&order=up#top
Notable absences

• Original Web did not include CSS or scripting
  – HTML mixes structure and presentation
• Separation of structure, presentation, behavior has developed since then
• No cookies (more about this in this lecture)
• No HTTPS / SSL (security – later lecture)
Enough history, how does it work?

• What is happening with Hello World?
• What does the browser do?
• What does the server do?
Revisit: Hello World Web Server
Trace of Hello World server

• Lots of stuff happening behind the scenes!
• What is all of the network traffic?
• Why is it there?
HTTP Basics

• Request/response
  – Old style: tear down connection each time
  – Optionally: “keep alive” to lower cost of multiple interactions

• Requesting a *method* on a URI

• Responding with a *status* of executing that request

• Stateless
Potential confusion

- HTTP methods unlike JavaScript methods
  - HTTP: Small set of fixed names (GET, PUT, POST, DELETE)
  - JavaScript: make up whatever you want, reuse other implementations, etc.
HTTP protocol structure (simplified)

- session ::= request response
- request ::= requestLine header+ [body]
- requestLine ::= method path
- response ::= status header+ [body]

- Headers include info about request/response or about body
  - Huge vocabulary of headers, allowed in any order
- Body contains the “stuff” either being sent or received
Methods

• GET – fetch indicated resource
• HEAD – get summary of indicated resource (GET without body)
• PUT – supply (replace) indicated resource
• DELETE – remove/destroy indicated resource
• POST – insert/add/create indicated resource
Response codes

- 1xx informational
- 2xx success
- 3xx redirect
- 4xx client error
- 5xx server error

Common codes:
- 200 OK (success, requested entity in body)
- 404 Not Found (don’t know where that is)
- 303 See Other (it’s moved, try this place)
- 500 Server Error (app implementor blew it, nothing better to say)
Stateless?

• Processing of request R2 isn’t affected by identity of previous request R1
  – i.e. GET means the same thing whether it’s preceded by GET or by PUT or by POST
  – Doesn’t mean the *result* is unaffected by sequencing of requests
  – “Stateful servers accessed via stateless protocol”
What if you want session state?

• Example: being logged in
  – GET → Failure
  – Login → OK
  – GET → OK, here’s what you asked for

• How can we get the second GET to work differently?
Cookies

• Client holds cross-request state in “cookie”
  – Just key-value store
• Client supplies cookie along with requests
• Ideally cookie holds all state
  – No state management at server
• Realistically, cookie is handle for server-maintained state
Look again at traces
How is that actually happening?

• Server sends Set-Cookie headers:

  Set-Cookie: name=value
  Set-Cookie: name2=value2; Expires=Wed, Jun 9, 2021 10:18:00 GMT

• Browser sends Cookie headers:

  Cookie: name=value; name2=value2
What about failures and ordering?

• Safe vs. Idempotent vs. General:
  – Assume right thing happens for 1 execution of method
  – What is difference for N (N > 1) executions of same method?

• N x safe method $\rightarrow$ no changes

• N x idempotent method $\rightarrow$ same as 1 execution

• N x general method $\rightarrow$ who knows?
Safe Methods

• GET, HEAD should not change state
• OK to repeat execution
• No problem with reordering them
Idempotent Methods

- PUT, DELETE cause change of state
- Can be repeated (2+ executions same as 1 execution)
- Cannot be reordered
What about POST?

• Causes side effects
• Not reorderable
• Not safe to repeat
Server implementing methods
Original Web vs. Modern Web

• Evolved, not designed
• Increasingly like a distributed operating system
  – Complex security issues, many still unresolved
  – Partly due to origins as simpler system
• Some basic elements are still unchanged
MATCHING WEB OPS WITH THE APPLICATION
HTTP vs. JavaScript

- Resources – named by URIs
- Methods – GET, PUT, POST, DELETE
- Status – numeric codes plus text explanation
- Headers modify or supplement
  - Header mechanism used on both request and response
  - Specific header may only make sense for one direction

- Objects
- Functions
Let’s go back to our Telnet server

• What were the operations there?
  – Adding a new prefix
  – Counting the number of prefixes
  – Deleting a prefix
  – Resetting the set
Webifying the app

• Add prefix: POST collectionURI/add newPrefix
• Count elements: GET collectionURI/count
• Delete prefix: POST collectionURI/delete prefix
• Reset: POST collectionURI/reset
Design questions

• Why not DELETE?
  – Operating on collection... don’t actually want to delete the collection
  – Often a fuzzy line between DELETE of element vs. POST of collection
  – Browser usage of DELETE is uncommon
Bigger picture

• Application Programming Interface (API)
  – The set of “buttons” available for programs to “push”
    • May be procedures, methods, hooks, callbacks, etc.
  – APIs have the same issues as other modules
    • What’s hidden?
    • What’s exposed?
    • Are the operations easy to understand?
    • Are the operations easy to use?
    • Are the operations suited to the task?
    • Does the API have conceptual integrity?
APIs via Web protocols

- System-Oriented Architecture (SOA)
  - Subsystems only communicate via published Web APIs
  - Improves opportunities for reuse
  - We will return to this idea later in the term
Seeking a general approach

- We’d like to have a principled way of approaching the web/app interface
General paradigm: CRUD

- Create
- Read
- Update
- Delete
Principled approach for Web

• HTTP has limited command set: map command + URL to application commands
• “Routing” maps HTTP command and URL to application behavior module
• Note: unrelated to network routing that gets IP packets to the right destination
General approach

• Create ➔ POST
• Read ➔ GET
• Update ➔ PUT
• Delete ➔ DELETE
Demo: simple HTTP methods
Demo: Simple routing
Programming idiom

• Declarative association of routes with processing
  – Instead of procedural then-do-this

• Similar to our approach to incoming messages, data
  – Triggered procedures
Easy(?) case

• Collection of documents
• Each document is just a “bag of bits”
  – No internal structure to track
• Add new document
• Delete existing document
• Iterate over documents
• Count of documents
A plausible design

• Add new document: POST collectionURI (returns documentURI for new document)
• Delete document: DELETE documentURI
• Iterate over documents: GET collectionURI/all (returns object containing all documentURIs)
• GET documentURI (returns one document)
• Count of documents: GET collectionURI/count (returns object containing number)
What if those opaque documents are photographs?

• Need more than one collection
• Need to label documents
• Need to label collections
• Need to have document in more than one collection
• Need to distinguish “delete” from “remove from this collection”
• May want to query on labels (names, dates, tags)
Design issues: Server for “Life” boards

• Granularity of sharing?
• Flexibility of sizes between clients?
• Split of computation between client and server?
• Semantics of sharing between clients?
Persistence problem

• How to capture our ending point today?
• Want to recreate an adequate starting point for Wednesday:
  – Record ending slide number
  – Record last topic fully covered
  – At start of lecture: *briefly* review last topic covered
  – Restart any partly-covered topic (don’t try to start in middle)
• This is like what we’ll learn about with databases
Next lecture

• Revisit servers from this lecture
  – Adding in databases!
  – Requires explaining/showing database
  – New set of “alignment” issues between app and database