RESTful Web Services

An introduction to building Web Services without tears (i.e., without SOAP or WSDL)

John Cowan
cowan@ccil.org
http://www.ccil.org/~cowan

Copyright

- Copyright © 2005 John Cowan
- Licensed under the GNU General Public License
- ABSOLUTELY NO WARRANTIES; USE AT YOUR OWN RISK
- Black and white for readability
- Gentium font for readability and beauty

The Pitch

Would you like something cleaner than SOAP? Something less impenetrable than WSDL? Something less confusingly intertwingled than the various WS-* bafflegab standards? ... Say, just what is this Web Services jazz anyhow?

The Pitch

It's all No Problem. It's all Easy as Pi. REST isn't some obscure thing that nobody supports; it's the way the Web already works, just formalized a bit and with some do's and don'ts.

The Pitch

By deconstructing what you already know about the Web, you can rebuild it into a set of principles for sound design, without worrying about it. No, it won't be "you push the button, we do the REST". But it'll be clean, secure, straightforward, extensible, discoverable, maintainable.

Talk is cheap

But that's what you're going to get today. After we're done here, go home and try it for yourself.

Uniform Resource Identifier

- I use the term “URI” (Uniform Resource Identifier) throughout
- If it makes you feel better, cross it out and use “URL” instead
- Contrary to all propaganda, there are no effective differences these days

Credits

- The guts of this presentation comes from the writings of:
  - Roy Fielding
  - Ryan Tomayko
  - Paul Prescod
  - Mark Baker
  - Jeff Bone *(conversus)*
  - All the contributors to RestWiki

Roadmap

- Web Services (12)
- What's REST? (18)
- The killer argument (7)
- Distributed Systems (13)
- What about ... (13)
- Clarifying "state" (6)
- From here to there (18)
- SOAP (9)
- Cleaning up (14)
- RESTafarian email (8)
- Related architectures (6)
- Final thoughts (3)

Web Services

What's a Web Service?

- A web service is just a web page meant for a computer to request and process
- More precisely, a Web service is a Web page that's meant to be consumed by an *autonomous* program as opposed to a Web browser or similar UI tool

What's a Web Service?

- Web Services require an architectural style to make sense of them, because there's no smart human being on the client end to keep track
- The pre-Web techniques of computer interaction don't scale on the Internet
- They were designed for small scales and single trust domains

The scope of the problem

- Computer A in New York ...
- ... tells computer B in Samarkand ...
- ... about a resource available on Computer C in Timbuktu
- None of them belongs to the same trust domain

Nouns

- URIs are the equivalent of a noun
- Most words in English are nouns, from *cat* to *antidisestablishmentarianism*
- The REST language has trillions of nouns for all the concepts in all the heads and files of all the people in the world

Verbs

- Verbs (loosely) describe actions that are applicable to nouns
- Using different verbs for every noun would make widespread communication impossible
- In programming we call this “polymorphism”
- Some verbs only apply to a few nouns
- In REST we use *universal* verbs only

GET: fetch information

- To fetch a web page, the browser does a GET on some URI and retrieves a representation (HTML, plain text, JPEG, or whatever) of the resource identified by that URI
- GET is fundamental to browsers because mostly they just browse
- REST requires a few more verbs to allow taking actions

Four verbs for every noun

- GET to retrieve information
- POST to add new information, showing its relation to old information
- PUT to update information
- DELETE to discard information

Not such a big deal

- The Web already supports machine-to-machine integration
- What's not machine-processable about the current Web isn't the protocol, it's the *content*

XML

- Using XML formats as your machine-processable representations for resources allows applying new tools to old data
- It also simplifies interconnection with remote systems
- XML has plenty of tools, as we all know

Why not just use plain HTML?

- Web pages are designed to be understood by people, who care about layout and styling, not just raw data
- Every URI could have a human-readable and a machine-processable representation:
  - Web Services clients ask for the machine-readable one
  - Browsers ask for the human-readable one

Well, not quite every URI

- The information on some pages is going to be too complex for machines to understand
- *Anna Karenina* has lots of meaning, but making it into a non-trivial Web service is an AI-complete problem

Are we doing this now?

- Most of us are are busy writing to layers of complex specifications
- Our nouns aren't universal
- Our verbs aren't polymorphic
- The proven techniques of the Web are being discarded for a pot of messages

What's REST?

So what's REST already?

- REpresentational State Transfer
- An architectural style, not a toolkit
- “We don't need no steenkin' toolkits!”
- A distillation of the way the Web already works

REST defined

- Resources are identified by uniform resource identifiers (URIs)
- Resources are manipulated through their representations
- Messages are self-descriptive and stateless
- Multiple representations are accepted or sent
- Hypertext is the engine of application state

REST style

- Client-server
- Stateless
- Cached
- Uniform interface
- Layered system
- (Code on demand)

Snarky question

- How are representations transferred, and why would I want a representation of something to be transferred to something else?
- Representations are all we really have (the shadows in Plato's cave)
- Representations are transferred by ordinary digital means – it's how we think about them that's new

Representation

- Resources are first-class objects
  - Indeed, “object” is a subtype of “resource”
- Resources are retrieved not as character strings or BLOBs but as complete representations

A web page is a resource?

- A web page is a *representation* of a resource
- Resources are just concepts
- URIs tell a client that there's a concept somewhere
- Clients can then request a specific representation of the concept from the representations the server makes available

State

- “State” means application/session state
- Maintained as part of the content transferred from client to server back to client
- Thus any server can potentially continue transaction from the point where it was left off
- State is never left in limbo

Transfer of state

- Connectors (client, server, cache, resolver, tunnel) are unrelated to sessions
- State is maintained by being transferred from clients to servers and back to clients

REST and HTTP

- REST is a *post hoc* description of the Web
- HTTP 1.1 was designed to conform to REST
- Its methods are defined well enough to get work done
- Unsurprisingly, HTTP is the most RESTful protocol
- But it's possible to apply REST concepts to other protocols and systems

Other protocols

- Web interaction using other protocols is restricted to REST semantics
- Sacrifices some of the advantages of other architectures
  - Stateful interaction with an FTP site
  - Relevance feedback with WAIS search
- Retains a single interface for everything

Existing HTTP uses

- Web browsing (obviously)
- Instant messaging
- Content management
- Blogging (with Atom)
- What's outside its scope?

What do REST messages look like?

- Like what we already know: HTTP, URIs, etc.
- REST can support any media type, but XML is expected to be the most popular transport for structured information.
- Unlike SOAP and XML-RPC, REST does not really require a new message format

Multiple representations

- Most resources have only a single representation
- XML makes it possible to have as many representations as you need
- You can even view them in a clever way, thanks to the magic of XSLT and CSS

Why hypertext?

- Because the links mirror the structure of how a user makes progress through an application
- The user is in control, thanks to the Back button and other non-local actions
- In a Web service, the client should be in control in the same sense

Web-based applications

- A Web-based application is a dynamically changing graph of
  - state representations (pages)
  - potential transitions (links) between states
- If it doesn't work like that, it may be *accessible* from the Web, but it's not really *part* of the Web

Code on demand

- Java applets weren't so hot
- Javascript is very hot
- The XmlHttpRequest object lets you do REST from *inside* a web page
  - Most browsers provide it nowadays, with a few annoying differences
  - It doesn't really require XML messages

A few simple tests of RESTfulness

- Can I do a GET on the URLs that I POST to?
- Iff so, do I get something that in some way represents the state of what I've been building up with the POSTs?
- HTML forms almost always fail miserably

A few simple tests of RESTfulness

- Would the client notice if the server were to be
  - restarted at any point between requests
  - re-initialized by the time the client made the next request
- These tests are not anything like complete

The killer argument

Arguments against non-REST designs

- They break Web architecture, particularly caching
- They don't scale well
- They have significantly higher coordination costs

Caching? Well ...

- The Web's caching architecture of the Web isn't always the Right Thing
- Using POST loosely to mean “don't cache” has been a good way of dealing with this problem
- Learning the stricter REST semantics of POST isn't just an extension of existing practice

Scaling? Well...

- What kind of scaling is most important is application-specific
- Not all apps are Hotmail, Google, or Amazon
- Integration between two corporate apps has different scaling and availability needs
- The right approach to one isn't necessarily the right approach to the other

The killer argument

- A service offered in a REST style will inherently be easier to consume than some complex API:
  - Lower learning curve for the consumer
  - Lower support overhead for the producer

What if REST is not enough?

- What happens when you need application semantics that don't fit into the GET / PUT / POST / DELETE generic interfaces and representational state model?
- People tend to assume that the REST answer is:
  - If the problem doesn't fit HTTP, build another protocol
  - Extend HTTP by adding new HTTP methods

But in fact:

- *There are no applications you can think of which cannot be made to fit into the GET / PUT / POST / DELETE resources / representations model of the world!*
- These interfaces are sufficiently general
- Other interfaces considered harmful because they increase the costs of consuming particular services

Be fruitful and multiply

- REST design appears to make web apps more likely to combine successfully with other web apps
- The resulting complexes of applications have a larger effect on the web as a whole
- REST tends to appear on the largest scales
- We don’t know in advance which apps will become large-scale

Distributed Systems

Distributed Systems

- Components (origin servers, gateways, proxies, user agents)
- Connectors (clients, servers, caches, resolvers, tunnels)
- Data elements (resources, resource identifiers, representations)

Components

- Communicate by transferring representations of resources through a standard interface rather than operating directly upon the resource itself
- Used to access, provide access to, or mediate access to resources
- Intermediaries are part of the architecture, not just infrastructure like IP routers

Some components

- *Origin servers:* Apache, IIS
- *Gateways:* Squid, CGI, Reverse Proxy
- *Proxies:* Gauntlet
- *User agents:* Firefox, Mozilla, Safari, IE

Connectors

- Present an abstract interface for component communication, hiding the implementation details of communication mechanisms
- All requests must be stateless, containing *all* the information necessary for the understanding of that request *without* depending on any previous request

Some connectors

- *Clients:* browsers, feedreaders, libraries, many specialized applications
- *Servers:* Apache, IIS, AOLserver
- *Caches:* browser cache, Akamai cache network
- *Resolvers:* DNS lookup, DOI lookup
- *Tunnels:* SOCKS, SSL after HTTP CONNECT

The connector view

- Concentrates on the mechanics of the communication between components.
- Constrains the definition of the generic resource interface

Resource modeling

- The value of components and connectors is mostly obvious
- Resources, representations, URIs, and standardized interfaces are more subtle matters

Resource modeling

- Organize a distributed application into URI-addressable resources
- Use only the standard HTTP messages -- GET, PUT, POST and DELETE -- to provide the full capabilities of that application

Some data elements

- *Resources:* the intended conceptual target of a hypertext reference
- *Resource identifiers:* URIs
- *Resource metadata:* source links, alternates
- *Representations:* HTML documents, JPEG images
- *Representation-specific metadata:* media type, last-modified time

Advantages of REST

- Its architectural constraints *when applied as a whole*, generate:
  - Scalable component interactions
  - General interfaces
  - Independently deployed connectors
  - Reduced interaction latency
  - Strengthened security
  - Safe encapsulation of legacy systems

Advantages of REST

- Supports intermediaries (proxies and gateways) as data transformation and caching components
- Concentrates the application state within the user agent components, where the surplus disk and cycles are

Advantages of REST

- Separates server implementation from the client's perception of resources (“Cool URIs Don’t Change”)
- Scales well to large numbers of clients
- Enables transfer of data in streams of unlimited size and type

The key insights

- Discrete resources should be given their own stable URIs
- HTTP, URIs, and the actual data resources acquired from URIs are sufficient to describe any complex transaction, including
  - session state
  - authentication/authorization

What about ...

GETs that won't fit in a URI

- Restricting GET to a single line enforces a good design principle that everything interesting on the web should be URI-addressable.
- Changing an application to fit GET's limitations makes the application *better* by making it compatible with the rest of the web architecture

Reliability

- The Web consists of many redundant resources
- It might be possible to find an alternate representation and transfer the session there
- Databases don't normally allow this
- The Web is a world of constantly shifting, redundant, overlapping network components in a wide variety of states.

Reliability

- You can do reliable delivery in HTTP easily at the application level
- The guarantees provided by TCP get you pretty far, and then you need just a bit more
- Connector reliability is solved by redundancy and other standard means that have nothing to do with REST

Reliability

- If at first you don't succeed, try, try again!
- The HTTP GET, PUT and DELETE methods are already idempotent, but the POST method creates new resources
- Multiple POSTs of the same data must be made harmless
- Put some kind of message ID in a header or in the message body

Reliability

- Clients aren't that good at generating truly unique message IDs
- Paul Prescod's solution:
  - The client POSTs to a URI asking for a unique server-generated message ID
  - The server returns an HTTP "Location:" header pointing to a newly generated URI where the client may POST the actual data.

Reliability

- The original POST is used only to generate message IDs, which are cheap.
  - Retire them (whether they have been used or not) after a few hours
  - Or hold on to them for weeks!

Reliability

- Wasted IDs are irrelevant.
- Duplicated POSTs are not acted on by the server
- The server must send back the same response the original POST got, in case the application is retrying because it lost the response

Asynchronous operations

- Send back notifications as POSTs (the client can implement a trivial HTTP server)
- Piggyback them on the responses to later requests
- No complete solution yet

Transactions

- The client is ultimately responsible
- Other designs aren't much better
- Database-style transactions don't scale well on the Web
  - Clients will start transactions and then forget about them
  - Ties up server resources
  - Locks out all other users

REST outside the Web?

- REST concepts apply in general to any system
- Some problems can be solved more cleanly or quickly with other non- or partially-REST approaches
- But then you can't really participate in the Web
- The larger or more foundational your system, the more you need REST

B2B

- B2B systems usually assume that POSTed documents disappear into each partner's internal business systems
- Business processes would actually work better if treated like a Web resource
  - An order is a resource
  - Shipments and payments are sub-resources
- Amazon gets this mostly right

Other protocols

- Other protocols are not organized around URIs the way HTTP is
- They break up the address space into pieces, some of which don't even have URIs
- HTTP was *designed* to manipulate resources labeled by URIs

Tunneling HTTP

- If you really do need non-HTTP transport, tunnel HTTP over that transport
- HTTP is pretty simple -- a couple of headers is all you absolutely need

Clarifying “state”

Two kinds of state

- Application state is the information necessary to understand the context of an interaction
  - Authorization and authentication information are examples of application state
- Resource state is the kind that the S in REST refers to
- The "stateless" constraint means that all messages must include all *application* state.

Resource state

- Changes in resource state are unavoidable
  - Someone has to POST new resources before others can GET them
- REST is about avoiding implicit or unnamed state; resource state is named by URIs
- Application state is required by the server to understand how to process a request

Session state

- Session state is also application state
- If you want a session, you often need smarter clients than a browser
- Specialized clients can manage both application and resource state

Sessions

- A purchasing client could send a single HTTP request mentioning everything it wanted to purchase in one message
- Shopping carts are for people, who have trouble keeping state in their heads

The purpose of statelessness

- Prevents partial failures
- Allows for substrate independence
  - Load-balancing
  - Service interruptions

Another kind of state

- Don’t confuse REST state with state-machine state
- REST state is the representation of the values of the properties of a resource
- State machines fit into REST when the states are expressed as resources with links indicating transitions

From where we are to where we'd like to be

The “OOP on the Web” theory

- HTTP is just a transport layer between objects
- Messages and objects are both opaque
- Objects jealously guard their private state

Smash the (private) state

- Eliminating private state lets us develop architectures that can scale to larger designs.
- REST systems transfer the entire state of the transaction at every state transition
- You can pick up where you left off by merely accessing the URI at a later time, regardless of client or server changes.

“My boss just wants it on time and under budget”

- An analogy: Our genes want *everyone* to reproduce
- But that doesn’t mean reproducing will always make *you* any happier
- If your want to build a web-accessible toolkit that a lot of people make use of, REST may help
- For a one-off project written by a small group of developers, REST may be irrelevant

RPC characterized

- Every object has its own unique methods
- Methods can be remotely invoked over the Internet
- A single URI represents the end-point, and that's the only contact with the Web
- Data hidden behind method calls and parameters
- Data is unavailable to Web applications

But in REST (just to rub it in)

- *Every* useful data object has an address
- Resources themselves are the targets for method calls
- The list of methods is fixed for all resources

REST and RPC

- REST is, in a sense, a species of RPC, except the methods have been defined in advance
- Most RPC applications don't adhere to the REST philosophy
- It’s possible to work with RPC-style tools to produce REST results
- Not that people actually do so!

Remote procedures

- Consider the stock example of a remote procedure called “getStockPrice”
- This isn't a resource (verb, not noun)
- It's not clear what what it means to GET, PUT, and POST to something called "getStockPrice"

REST just RPC renamed?

- But if we change the name from "getStockPrice" to "CurrentStockPrice" (a noun), all is well! ☺
- The differences between RPC and REST can be quite subtle
- If that were all, REST would be just a design style, not an architecture

There are no neutrals there

- REST is *incompatible* with "end-point" RPC
- Either you address data objects or you address "software components“
  - REST does the former
  - End-point RPC does the latter
- You can try to contort RPC protocols into working on data object URIs, but then you end up re-inventing a non-standard variant of HTTP

Can REST really beat RPC?

- If REST works and RPC doesn't, then yes!
- SOAP began as pure RPC and has been moving further and further away
- SOAP (and its parent XML-RPC) have been around for years and yet there is no killer app
- REST can point to the Web itself as proof that It Just Works

Two views of POST

- "POST lets you pass a whole lot of parameters and get something back, bypassing caches."
- "POST lets you create new resources that are related to old ones."
- The second is the REST attitude

REST: an alien notion

- RPC-over-HTTP is well-matched with current thinking
- Take an existing object model, and a little Web-specific glue, and simply export those interfaces to the Web
- The problems creep in down the road

REST sounds ominous

- Completely rethink your design in terms of generic interfaces
- Build servlet-style implementations of each resource
- Unpack and repack Request and Response objects
- The gluuuuue is up to yooooou.

REST sounds ominous

- Plenty of people do know how to develop servlets
- Still, most developers and data modellers think only in UML and OOP
- REST is potentially as significant a change as the transition from procedures to objects

“REST is ha-ard” --RPC Barbie

- It sometimes takes as much work to learn to use one tool well than five tools badly
- In the long run you are better off
- XML was ha-ard too for people used to HTML, flat files, and CSV
- “Some people refused to learn to use the telephone. They don’t work here any more.”

Paul Prescod shows us the REST way

```
POST /purchase_orders HTTP/1.1
Host: accounting.mycompany.com
content-type:
  application/purchase-order+xml
....
<po>...</po>
```

And then there's the SOAP way

```
POST /generic_message_handler
content-type: application/SOAP+XML

<soap:envelope>
 <soap:body>
   <submit-purchase-order>
     <destination>accounting.mycompany.com</destination>
     <po>...</po>
   </submit-purchase-order>
 </soap:body>
<soap:envelope>
```

Stacking the deck

- Namespace declarations would make the SOAP example much bigger
- XML is not magic pixie dust: sometimes plain text is all you need
- In the gazillions-of-transactions-per-second world, these things count
- Do more with less

SOAP

SOAP: neither fish nor fowl

- A base from which to build new protocols and tunnel them over existing application protocols (typically HTTP)
- A means to extend the semantics of those same application protocols

SOAP can be RPC or not

- Originally SOAP was a pure RPC transport like its ancestor XML-RPC
- More recent versions of SOAP promote the less problematic “document/literal” style, which is analogous to email:
  - No explicit method name
  - The recipient decides what to do

POSTing a SOAP message

- Wrap the body in a SOAP envelope
- POST it to an endpoint URI
- A response comes back, which you must unwrap
- Or you might get a fault, which overrides (older SOAP) or duplicates (newer SOAP) the HTTP response code

POSTing a SOAP message

- SOAP uses its envelope for what new HTTP headers could do
- SOAP provides the meta-metadata "actor" and "mustUnderstand"
- If the body of the SOAP message represents an entity that is being POSTed to something, at least part of the REST style is preserved

The advantages of SOAPless GET

- More tools out there that can do HTTP gets (proxies, spiders, browsers) than can interpret your SOAP method as a getter
- Resources that are gettable have URIs that can be linked to
- SOAP endpoints should at least provide an alternate interface that allows vanilla HTTP getting

HTTP is not a transport protocol

- If the body of a POST or PUT is not a piece of representational state, you're not doing REST
- HTTP already defines these methods and doesn't need new ones inside the POST body

HTTP is not a transport protocol

- SOAP abuses HTTP by treating it as a transport protocol like TCP
  - "HTTP only exists to carry bits, namely SOAP messages, with or without a method name"
- HTTP is an *application protocol;* it doesn't send bits, it transfers representational state

Web Method specification

- SOAP 1.2 exposes the HTTP method through the SOAP binding
- SOAP clients can use GET to retrieve SOAP envelopes that contain the state of the resource identified by the URI

Web Method specification

- Potentially radically different from the common uses of SOAP 1.1
- Will SOAP 1.2 applications automatically become more RESTful? Not a bit
- Most SOAP users will probably continue to use SOAP 1.2 in the same ways as SOAP 1.1.

Cleaning up current practice

Cookies

- A receipt for application state handed out by the server
- Using cookies is being stateful:
  - Not all application state is carried in the message
  - The cookie's referent is held on the server

Cookies aren't all bad

- At least there exists a reference to the state
- The request can be load balanced to some other server within the same trust domain for processing
- Beyond that trust domain, cookies don't mean anything to anybody
- That makes people paranoid about them

Cookie problems

- Cookies break *visibility*
  - Caches don't understand them
- Cookies are bad authenticators
  - They give up security for efficiency
- Clients often shut off cookies to provide real or imagined privacy

Keeping state in the cookie

- Lets URIs be independent of the user state
- But it destroys the client's understanding of state as presented by hypertext
- It breaks the Back button

Keeping a *reference* to state

- Storing state on the client provides REST's scalability.
- Sites with client sessions on the back end are usually several orders of magnitude less scalable than REST-based applications
- They also require much more complex back-end engines (J2EE, for example)

Keeping identity in the cookie

- Cookies are more efficient than proper HTTP authentication
  - servers and intermediaries simply ignore them for most URIs (e.g., inline images)
- But the server is relying on security by obscurity
- Cross-site scripting and cookie guessing are real dangers

Tunneling

- Using POST to send data that's supposed to mean something other than POST to the recipient is tunneling
- Administrators detest tunneling and for good reason
- Because SOAP is a meta-application protocol, tunneling is its middle name

Don't tunnel through port 80

- Firewalls and ports exist for a reason
- When you show up at the airport, if you claim that you are a pilot you'll probably get waved through more quickly. But ...
- It's dangerous to lie to the firewall systems put up by people working for the same company you do, trying to protect it from the outside world

Don't tunnel through port 80

- Security administrators *will* find a way to shut your RPC over port 80 down
- Then you'll have to add another layer of obfuscation
- In the long run the extra layer will no longer buy you a free pass through the firewall
- You end up with an arms race of escalating obfuscation and detection

Application protocols and safety

- Applications protocols provide safety guarantees by providing a fixed interface
- Only limited things can be done through the interface
- SMTP doesn't let you do anything but send mail
- It can't be used to retrieve files unless somebody explicitly installs software that allows such tunneling

Application protocols and safety

- SMTP doesn't include such tunneling features by default
- Consequently it is trusted and well deployed
- (Spam is not an SMTP problem *per se*)
- Fixed interfaces are secure, because software implementing them only does what it's designed to do

Use HTTP as HTTP

- Use HTTP because it is pragmatic
- Also use HTTP *as HTTP* so that it works with, not against the firewall software and firewall administrators
- Make each message as visible as possible to the firewall, and invisible and opaque to crackers
- Letting arbitrary requests tunnel through your firewall is asking to lose

Plain HTTP vs. SOAP on HTTP

- See Paul Prescod's examples again
- Which one can be readily filtered with security software?
- Which one can a sysadmin inspect and understand in a logfile?

Working with REST, not against it

- Reconsider your application's needs in terms of the provided interfaces and semantics
- Don't try to figure out how to subvert or extend HTTP to encompass what you think your application semantics are

RESTafarian Email: an example

RESTafarian Email

- If we were designing email from scratch on REST principles, what might it look like?
- This is *one possible way*, not the One True REST Way
- REST is nothing if not flexible, provided you stick to the few principles we've already seen

Mail servers keep outgoing mail

- To post an email, use POST!
- Your local outbound mail server exposes a URI where outbound messages can be posted
- Security makes sure only authorized users can post
- The mail never leaves the server until the sender or the recipient decide to delete it

Mailbox servers keep inbox state

- To read your mail, use GET to fetch a set of hyperlinks (nicely formatted) that represent incoming messages
- GETting one link sends you to the mail server that has the message and retrieves it
- DELETE removes messages you no longer want

Mailbox servers keep inbox state

- Archived messages are displayed in views you can GET
- Folderizing is POSTing a message containing a URI to the folder (which itself has a URI)
- Forwarding is almost like folderizing, but to someone else's inbox
- Higher-level services like searches are done by POST and create new resources that you can wait for or GET later

Mail notification

- Mail servers have to tell mailbox servers that mail is available
- Inbound servers expose a URI that can be POSTed to with a cheap message containing just a URI

No spam!

- Any recipient can delete a message, so just keeping one copy on the spammer's mail server won't work
- Spammers would have to keep zillions of copies on their mail servers
- That costs $$$$ and draws attention
- A spam no one gets to read isn't a spam

No spam!

- Of course a spammer can cheat by using a server that improperly ignores DELETEs
- But that only works once, as such servers get blacklisted (and they cannot trivially hide their identities)
- No social problem can be *completely* solved by technical fixes

“Post in haste, repent at leisure”

- SMTP mail once sent can't be retrieved
- Senders can use PUT or DELETE to modify or remove their mails even after posting them
- Of course, that doesn't change the state in the recipient's head

Related architectures

Systems vs. applications programming

- Systems programming emphasizes making the new domain fit into the existing generic interfaces
- Applications programming models the application domain precisely first, worries about integration afterwards (if at all)

Thoughts of a systems geek

- If applications programmers thought more like systems programmers, the world would be a better place
- If a problem is not interesting, generalize it until it is, then solve the general problem

The Unix Way

- Unix has destroyed all its competitors but one (to the point where many people can't even name those other competitors)
- The core of Unix is its software tools philosophy:
  - the ability to string together lots of little special-purpose tools with generic interfaces

The Unix Way

- Everything is a file
  - Files have a generic interface
  - All resources in the system could be accessed through these narrow interfaces
  - Some things were always exceptions
  - Unix networking broke this philosophy
  - The Plan 9 research OS restored it, doubled and in spades

REST from a Unix viewpoint

- Resources rather than files
- URI space instead of the filesystem
- A slightly different (even narrower) generic interface
- But the focus is the same: a generic shared abstraction, not point-to-point interface coordination.

Other coordination environments

- In Linda, you get and put anonymous tuples
- In UNIX shell programming, autonomous programs read and write from pipes
- Plan 9 extends the filesystem to be a universal namespace
- To write a device driver, you implement *open* and *close* and *read* and *write* and *ioctl* and ...

Final thoughts

Has RPC really failed?

- ONC and DCE RPC are the basis of:
  - Plenty of enterprise software
  - The widely deployed NFS
- CORBA and DCOM are in *lots* of industrial-strength enterprise software.

REST and WS-*

- In the end, WS-* is just *there*, like Windows
- REST people need to work to ensure that the WS-* stack is sufficiently rich to be useful to them
- Two different design styles, informed by different needs and values
- They should still share a technology base as much as possible (and no more)

You're my only hope

- The only thing that can really make REST work for us all is broad education in:
  - What, exactly, the REST style *is*
  - *How* to design to it
  - Why it's a Good Thing
- But that’s why you’re here