Clarifying the Fundamentals of HTTP

Jeffrey Mogul
HP Labs – Large Scale Systems group
JeffMogul@acm.org

January 21, 2003

What's wrong?

Myth: the HTTP protocol is
- Simple
- Easy to understand
- Easy to extend

Reality: the HTTP protocol is
- Complicated (RFC2616: 176 pages, + other RFCs)
- Confusing
  - many mailing-list questions by implementors
- Hard to extend without conflicts
Goal of this talk

Examine fundamental definitions and models of HTTP
- Look at current models/definitions
- Show where these cause trouble
- Fix them

Not “write model first, then make HTTP fit the model”

Rather:
- deduce sound model, based on HTTP experience
- then, make sure protocol fits its implied model
Outline

- History and motivation
- HTTP's data type model
- HTTP's data access model
- Extensibility
- Other stuff
- Related and future work
My history and motivation

I have tried to articulate these issues before:

- 1995-1999: Within HTTP-WG
- 1999: USENIX invited talk:
  - “What’s Wrong with HTTP”
- 1999-2000: Writing (and rewriting) spec:
  - “Delta Encoding in HTTP”
- 2001: HotOS-8 position paper

Maybe this time I’ve gotten it right?
The importance of caching

HTTP: a network protocol and a distributed system
- Caching vital for distributed systems performance

Caching illuminates weaknesses in any specification:
- Caching must be semantically transparent
  - Wrong answers are worse than slow answers
  - Bad caches encourage cache-busting
- Caches are intermediaries, not end-points
  - Forces us to be explicit about the details
  - Can become inhibitors of extensibility
Is it a protocol or a distributed system?

Protocol designers:
- Think in terms of messages
- Treat implementations as black boxes
- Worry about interoperability & extensibility

Distributed systems designers:
- Think in terms of state
- Think hard about caches
- Worry about correctness and error conditions

We need to think both ways about HTTP
Outline

- History and motivation
- HTTP's data type model
- HTTP's data access model
- Extensibility
- Other stuff
- Related and future work
Data type model

What is a “data type model” for HTTP?
- Data types operated upon by the protocol
- Also includes transformations between data types
- **Not**: MIME types (these are opaque to HTTP)

Convenient to think of “message generation pipeline”
- Abstraction of steps carried out by server
- Nodes represent data types
- Edges represent transformations
HTTP's existing data type model

Fuzzy model, implicit in the spec:
- **Resource**: thing which a URL points to
- **Variant**: language-specific version of resource
- **Entity**: information (body + headers) in a response
- **Message**: carries entity or pieces thereof

Also, **Entity tag**:
- Assigned & sent by server, stored in cache entry
- Sent by client in conditional request (revalidation)
- If matches, cache entry still valid
Message generation pipeline

Things to note:
- There's a cycle in the pipeline
- Entity tag is assigned on an edge, not at a node
- Edge between variant and entity is not labeled
Problems with existing model

Specification of caching
- What does a cache store?
  - Not any of: resource, variant, entity, message

Partial results
- How to combine ranges, compression?
- How to define delta encoding?
- When does the server assign an entity tag?

These are both too complex to describe here!
Why “entities” don't have “entity tags”

Example: client requests both
- Compression as content-coding
- Byte range

GET /foo.html HTTP/1.1
Accept-Encoding: gzip
Range: bytes=1-1000

Question: which comes first?
- Order does matter! (compression changes sizes)
- HTTP specification gives no explicit answer
“Entities” vs. “entity tags,” part II

Answer: implied by the constraints of the specification:
- content-coding MUST precede range selection
- Otherwise, no consistent point to assign entity tag

Deduction from two constraints:
1. Entity tag MUST be assigned before range selection
   - else cannot match etags when combining 2 ranges
2. Entity tag MAY be assigned after content-coding
   - else servers cannot store pre-encoding files

So: content coding ≤ entity-tag < range selection
“Entities” vs. “entity tags,” part III

Given: content coding ≤ entity-tag < range selection
- e-tag cannot be assigned to output of range selection
- But, output of range sel. is an entity, by definition

So, not all “entities” can be given “entity tags”

Simplest consistent solution:
- No “entities” can be given “entity tags”
- The output of a content-coding is not an “entity”

Aha!

Expanded from: WWW2002 – Clarifying the Fundamentals of HTTP
More problems with existing model

Header categorizations
- What headers can a proxy cache modify?
- How to make this extensible?

Protocol specification complexity
- Lack of modularity
- Complex wording
An improved data type model

Key idea: add one new data type:

Instance: the entire result of successfully applying
GET to a given resource variant at a given point in
time. *(Defined more formally in the paper)*

Also add (optional) instance manipulations

- May generate partial results
- E.g., Range selection, Delta encoding, compression
- Add new IM and A-IM headers for labeling

Expanded from: WWW2002 – Clarifying the Fundamentals of HTTP
Improved message generation pipeline

- Selecting headers
- Content-codings
- Range, Delta, other instance manipulations
- Transfer codings

Assign instance tag here

Things to note:
- No cycles in pipeline
- “Entity tags” are really instance tags!
- Entity/instance tag is assigned at a node
- All edges are labeled
Why is the new model better?

Caching specification is much simpler
• Cache entries store instances (or parts thereof)
• “Entity tags” identify instances

Much cleaner handling of partial results
• Clients can specify ordering (using A–IM)
• Composition with compression is well-defined

Header categorizations make sense (mostly)
• Many more categories than RFC2616 (see paper)
• Rules for, e.g., “end-to-end” hdrs (almost) trivial
### Examples of header recategorization

<table>
<thead>
<tr>
<th>Field name</th>
<th>RFC2616</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>response-header</td>
<td>outer-header</td>
</tr>
<tr>
<td>Cache-Control</td>
<td>general-header</td>
<td>instance-header</td>
</tr>
<tr>
<td>Content-Encoding</td>
<td>entity-header</td>
<td>entity-header</td>
</tr>
<tr>
<td>Content-Length</td>
<td>entity-header</td>
<td>entity-header</td>
</tr>
<tr>
<td>Content-Type</td>
<td>entity-header</td>
<td>entity-header</td>
</tr>
<tr>
<td>ETag</td>
<td>response-header</td>
<td>instance-header</td>
</tr>
<tr>
<td>Expires</td>
<td>entity-header</td>
<td>variant-header</td>
</tr>
</tbody>
</table>

The paper includes a more complete table
Can we simplify the specification?

Possibly not:
● HTTP was designed by evolution
● Layering wasn't considered during design
● Caching seems to touch almost all data types

Possibly yes:
● Create modularity but without strict layering
  – add exceptions rather than try for perfection
● Require new extensions to follow the model
  – for example, label headers ("I-Digest")
Outline

- History and motivation
- HTTP's data type model
- HTTP's data access model
- Extensibility
- Other stuff
- Related and future work

Expanded from: WWW2002 – Clarifying the Fundamentals of HTTP
Access model

What is an “access model” for HTTP?
- *What* kinds of data can be accessed
  - E.g., mutable or not?; side-effects?; idempotent?
- *How* data is accessed
  - E.g., GET, PUT, POST, DELETE
- Not an “access-control” (protection) model

Most important for
- Non-human agents (robots, automated clients)
- Intermediaries (caches, proxies)
Problems with the access model

Too much is inferred that should be explicit:

- No access-model labeling, e.g.:
  - is it safe to replay the request?
  - will there be a new instance value in the future?

- No clear mapping to protocol requirements, e.g.:
  - can I do a PUT on this resource?
  - is it safe to pipeline requests on this resource?
"Static" vs. "dynamic" red herring

Typical special treatment to "dynamic" resources:
- Caching not allowed
- No replays (side effects expected)

This is nonsense!
- "Static" vs. "dynamic" is orthogonal to caching
- Should be a hidden detail of server implementation

Should only consider cost to generate the response:
- Can decouple from other access-model issues
- Dynamic might sometimes be cheaper than static

Expanded from: WWW2002 – Clarifying the Fundamentals of HTTP
Fixing the access model

Add labels
- Make everything explicit
- Never require inferences

Labels include
- “mutable”, “side-effects”, “assignable”, etc.
- Cost(s) to generate response (rather than “dynamic”)

Attach labels to instances, resources (maybe variants?)
Abuse of terms

Many people (even Famous Researchers) abuse terms:

- “URL” instead of “response” or “instance”
- “document” or “page” for single-resource items

Bad, bad researchers!

Examples from actual papers (perps anonymized):

- “the replacement algorithm decides which documents are cached and which documents are replaced”
- “it performs an IMS GET of a cached page”

Expanded from: WWW2002 – Clarifying the Fundamentals of HTTP
“Resources” vs. “pages”

But: humans care about “pages,” not “resources”

- Browsers render pages
- Useful URLs (as links or typed) point to pages

Terminology errors reveal conceptual gap in HTTP

HTTP has no way to refer to multi-resource pages:
- hard to keep a page internally coherent
- hard to produce snapshots without full archiving
- makes some trace-based analyses much harder :-)

Expanded from: WWW2002 – Clarifying the Fundamentals of HTTP
**Supporting pages in HTTP?**

- “History mechanisms” take a baby step
  - Recognizes that “back” requires coherent view
  - Doesn't help with forward progress
- Cookies can be applied to specific paths
  - Could allow server to detect certain errors
  - Not generally useful to caches
- “Collections” in WebDAV and DRP
  - Explicit support for multi-resource coordination
  - Not generally applicable to all HTTP pages
Outline

- History and motivation
- HTTP's data type model
- HTTP's data access model
- Extensibility
- Other stuff
- Related and future work
Extensibility

HTTP has one simple, powerful extension mechanism:
- “Ignore all headers you don't understand”

But lacks good support for complex extensions:
- Do we both agree to use a given extension?
- Do we agree what it means?
- Does entire path (including proxies) support it?
Extensibility problems

Consequences of current shortcomings include:
- Servers basing their responses on User-agent
- Javascript that “knows” what each browser can do

Failed attempts at solutions include:
- OPTIONS method
  - no detectable syntax or useful semantics
- RFC2774 “HTTP Extension Framework”
  - too complex
  - stuck as an “experimental” RFC
Why HTTP version numbers don't help

HTTP messages include a version number, but:

- Too many optional features associated with version
- Many different revisions for each version
- Some proxies lie about the version number
- Hop-by-hop only; says nothing about full path
What are the possible solutions?

How do we discover extensions?
- Trial and error
- Negotiate
- Declare capabilities

How do we name extensions?
- Implicitly (trial and error)
- Decentralized and flexible (as in RFC2774)
- Centralized and slow to change
A proposed extension mechanism

Choose:
- Declare capabilities (of every hop)
- Centralized naming

Use RFC numbers as extension names:
- Compact, non-ambiguous namespace
- RFCs are immutable (by IETF rules)
- IETF requires unambiguous meaning for RFCs
- Define “profile” RFCs for compact naming of sets
- (Proposed 1997 by J. Cohen, S. Lawrence, & me)
Outline

- History and motivation
- HTTP's data type model
- HTTP's data access model
- Extensibility
- Other stuff
- Related and future work
Variants really mess up the protocol, because they:

- Confuse the binding between URLs and resources
- Make caching much more complicated
- Combine multiple issues into one mechanism:
  - Non-automatable issues (e.g., French or German)
  - Device-specific properties (e.g., screen size)
- No clear model in the specification

I tried to write about variants, and gave up

Probably someone needs to start from scratch
Intermediaries

HTTP needs/tolerates/suffers intermediaries for:
- Speed: caching, server acceleration, transcoding
- Functionality: security, transcoding

HTTP/1.1 includes explicit proxy support, but:
- Many intermediaries are invisible but not transparent
- Spec doesn't handle transcoding very well

HTTP needs a more careful approach to intermediaries
- Some IETF work in progress (WREC; OPES recs.)
Protocol support for user-interface concerns

Theory:
- clear boundary between HTTP protocol and UI
- (after all, some HTTP clients have no UI!)

Reality:
- server & client need to communicate about UI state
- already some spec. words re: security state
Improving UI support

Example: history mechanisms ("back" & "forward")
- Spec. distinguishes these from caching
- But most implementations blur the distinction
- Forces annoying site-design glitches
  - E.g., "do not use the `back' button" warnings
  - E.g., no-store for "security" on shared browsers

Better approach: explicit server control over UI
- Separate from cache-related features
- E.g., "prevent this page from appearing in history"
Outline

- History and motivation
- HTTP's data type model
- HTTP's data access model
- Extensibility
- Other stuff
- Related and future work
Related work

Fielding & Taylor (2000, Intl. Conf. on Software Eng.)
- “Principled design of the modern Web architecture”
- Idealized model for interactions on the Web
- Points out some flaws in HTTP
- “Representation” model blurs too many distinctions

Eastlake (2001, Internet-Draft)
- “Protocol versus document points of view”
- Argues against taking just one of these points of view

Expanded from: WWW2002 – Clarifying the Fundamentals of HTTP
More related work

Baker (2001, Internet-Draft)
- “An abstract model for HTTP resource state”
- Effectively a data access model for HTTP/1.1
- Improves clarity without actually changing spec.

Several “HTTP Next Generation” efforts:
- Usually attempts to redesign HTTP from ground up
- None have gone very far
Future work

Data type model needs:
• Testing on other extensions
  – e.g., CDNs, coherent caching
• More work on arcane header-specific rules

Unfinished business:
• Variants (naming, semantics, and caching)
• Extension model (naming and semantics)
• Transcoding (and intermediaries in general)

Clean up the spec, in general!
Messages to take home

The HTTP spec needs some cleanup:

- Leave the *protocol* alone, but fix the *words*
- Explicit, careful models help with rigor and clarity
- “Entity tags” are really “instance tags”
- Think harder about how composition of features

Eliminate ambiguity:

- Never require inferences or heuristics
- Add tagging if necessary
- Extensions need a simple, precise namespace