What's wrong with HTTP (and why it doesn't matter)

Jeffrey C. Mogul
mogul@pa.dec.com
Compaq Computer Corporation
Western Research Laboratory

June 1999

COMPAQ Western Research Laboratory

HTTP is *the* killer application protocol

- 75% of the bytes on Internet backbone
- $Billions of value for .com stocks
- Even my mom uses the Web

The HTTP protocol design is a mess

- Complex and often confusing specification
- Many basic concepts are wrong
- Numerous inefficiencies
- Even a spelling error

This talk will

- Explain the flaws in the HTTP design
- Explain why it still succeeds

Why this talk?

Better understanding of HTTP

- The good and the bad

Lessons for future protocol designers

- Mistakes to avoid
- Areas for future improvement
- Procedural issues

Disclaimer

This talk is entirely my own opinion!

I do not speak for

- The IETF's HTTP Working Group (HTTP-WG)
- Other authors of the HTTP/1.1 specification
- Compaq or its partners

and I expect some of these people will violently disagree with what I say.

Outline

History of HTTP

Overview of HTTP and HTTP/1.1

Fundamental mistakes

Procedural problems in the HTTP-WG

Why the bugs in HTTP don’t matter

A Brief history of HTTP

Pre-history:

- Hypertext dreams (but no protocol)
- Gopher protocol (but no hyperlinks)
- WAIS

At CERN:

- 1990: original deployment (Tim Berners-Lee)
- Jan. 1992: first known spec (on www-talk)
  - Only "GET" method
  - No documented headers
  - Generally known as ‘‘HTTP/0.9’’
- Dec. 1992: first www-talk mention of upgrade
  - MIME-compatible
  - HTTP headers

History continued

Internet Engineering Task Force (IETF):

- Mar. 1993: first Internet-Draft for HTTP/1.0
- Nov. 1993: revised Internet-Draft for HTTP/1.0
- Dec. 1994: first HTTP-WG meeting (‘‘BOF’’)
- May 1996: RFC1945: HTTP/1.0
  - Not an IETF standard
  - Reflected consensus of implementors
  - Generally recognized as problematic

HTTP/1.1 timeline:

- Nov. 1995: first Internet-Draft issued
- Jan. 1997: RFC2068 (‘‘Proposed Standard’’)
  - Actually finished in July 1996
- Nov. 1998: final HTTP/1.1 Internet-Draft
  - ‘‘Draft Standard’’ status approved
  - In RFC Editor’s queue since March 1999

Overview of HTTP and HTTP/1.1

Basics:

- Request-response protocol
  - No response without a request
  - No callbacks
- ASCII headers
  - MIME-like syntax
  - Special format for first line
- Binary data in optional *body*
  - HTTP treats body as bag-of-bits
- A request applies a *method* to a *resource*
  - Methods include: GET, PUT, POST
- HTTP uses TCP as transport (almost always)

Example: HTTP/1.0 exchange

Client sends:

```
GET /home.html HTTP/1.0
Accept: text/html, image/*
```

Server replies:

```
HTTP/1.0 200 OK
Date: Sat, 01 Aug 1998 06:59:59 GMT
Content-type: text/html

<HTML>
<HEAD><TITLE>
This is a title
</TITLE></HEAD>
This is an example.
</HTML>
```

Other HTTP/1.0 features

HTTP/1.0 included support for:

- Caching
  - `Expires`, `Last-Modified`, `If-Modified-Since`
- Internationalization
  - `Accept-Language`, `Accept-Charset`, etc.
- Authentication
  - with cleartext passwords!

HTTP/1.1: fixing(?) HTTP/1.0

Persistent connections & pipelining
- Multiple requests per connection
- No need to wait for preceding reply
- Requires explicit end-of-message mechanism

Clean up caching
- More careful model
- Entity tags
- Explicit control using `Cache-control`
- `Vary` header for negotiated resources
- `Warning` header for non-fatal errors

Partial transfers (‘‘`Range`’’ requests)
- For grabbing prefixes
- For resuming after an error

Extensibility
- `Via` header for hop-by-hop version info
- More careful rules for proxies and servers

HTTP/1.1, continued

Digest authentication
- Avoid cleartext passwords
- Some end-to-end message integrity

IP address conservation
- Send server hostname in `Host` header

Content negotiation
- Actual syntax specification
- Supports ‘‘quality factors’’ (preferences)

Negotiated use of compression, etc.
- More efficient use of bandwidth
- Allows use of new coding algorithms

More detail in WWW8 paper by Krishnamurthy, Mogul, and Kristol

Fundamental mistakes

Topics:

- Data model and the MIME miasma
- Extensibility
- Caching
- Header categories
- Status and error codes
- Transport issues
- Content negotiation
- Cookies & other social issues

My rating system

- A really annoying mistake:

- Bugs me that we didn't solve this:

- Might become trouble:

Data model and the MIME miasma

Original HTTP vision:

- ‘‘object-oriented protocol’’
- MIME-conforming and MIME-compatible
- Objects with many variants

Consequences:

- Terminology:
  - ‘‘Objects’’ and ‘‘methods’’
  - MIME ‘‘entity’’ for in-transit payload
- Possibility of ‘‘dynamic’’ resources
- MIME header rules (more or less)
- MIME content-type system
- URL does not always identify specific variant

COMPAQ Western Research Laboratory

Object-oriented? Not!

HTTP transfers current resource response

- Not the resource itself

Cached values don’t work like dynamic resources

No cache coherency for updatable resources

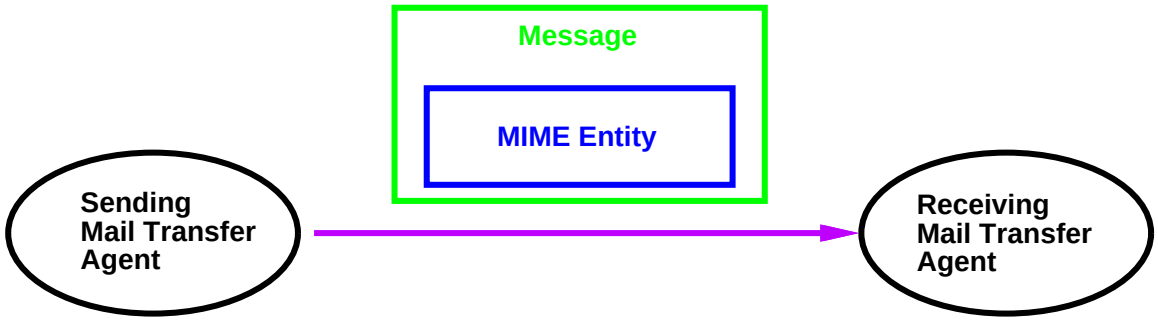
HTTP/1.1 no longer described as ‘‘objected oriented’’

So now what?

Entities and resources and instances, oh my!

Crucial mistake: false analogy with MIME entities

MIME model:

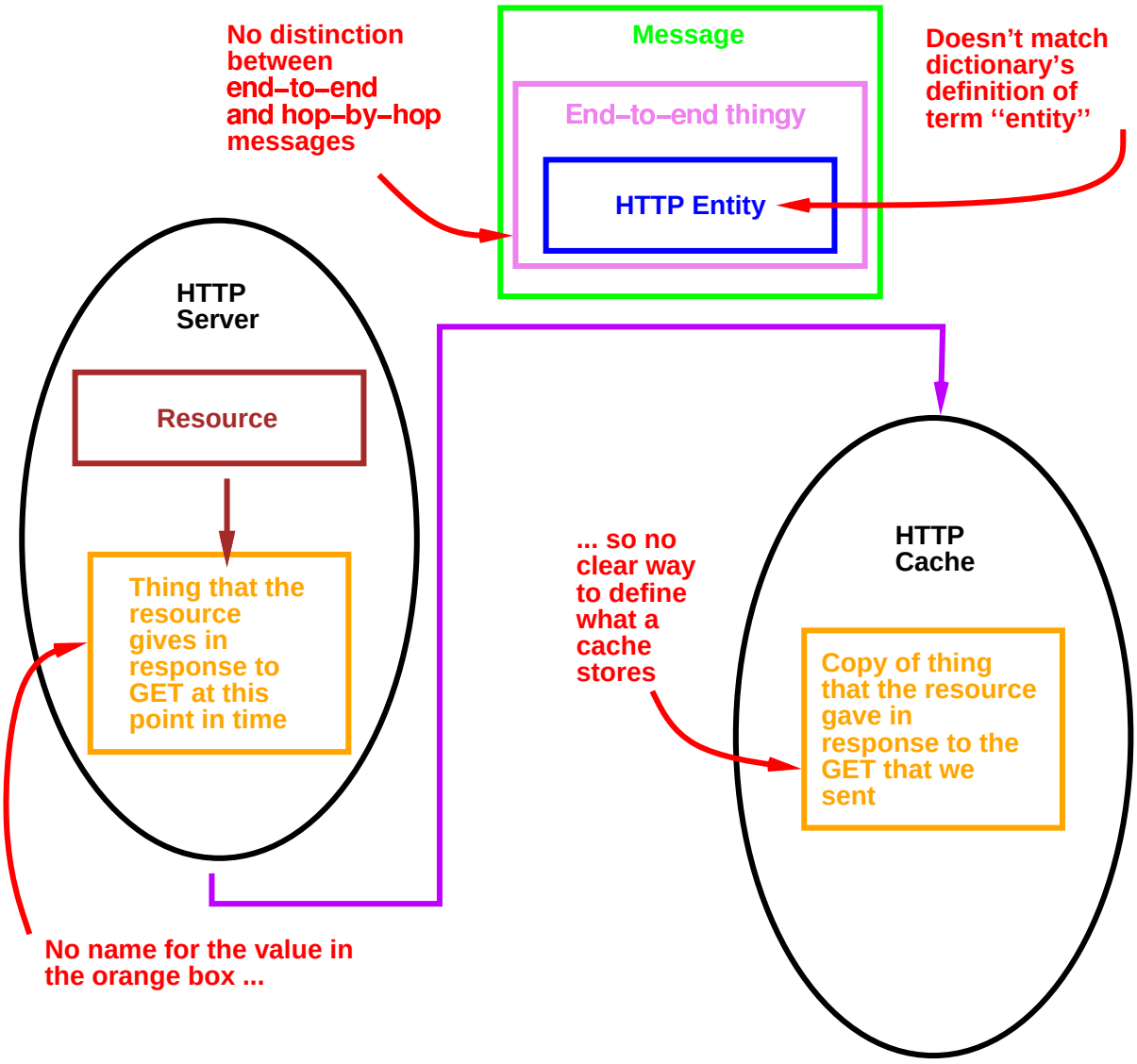


Note:

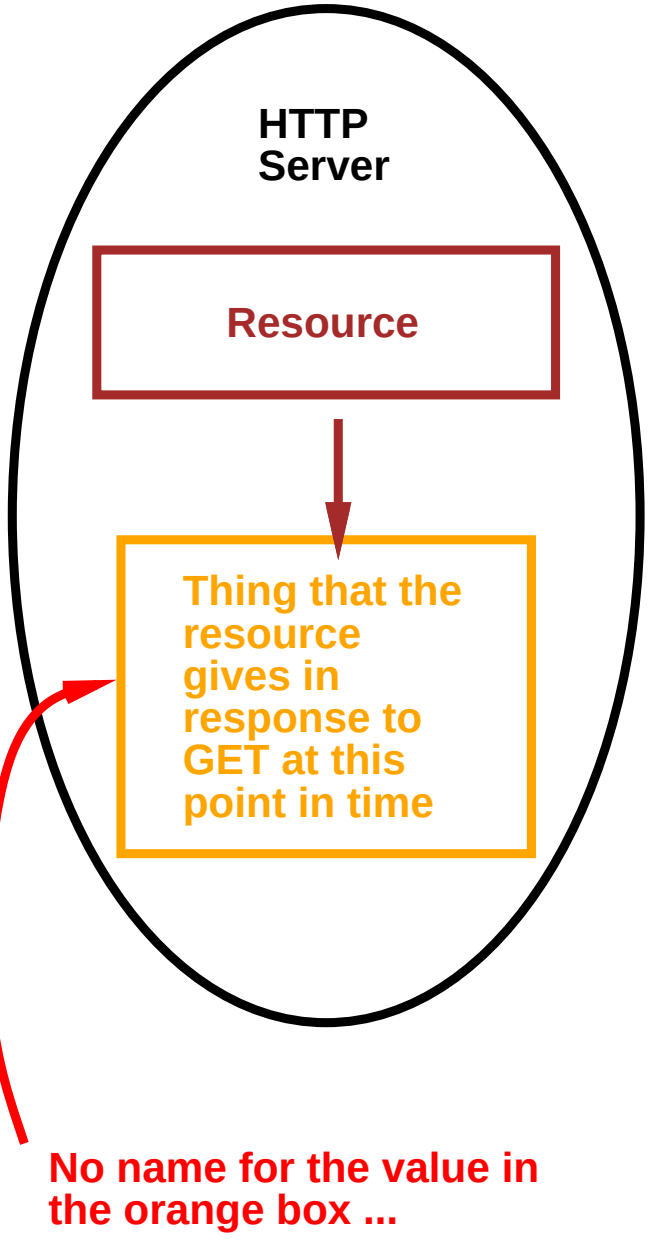
- No data types besides entities and messages
- End-to-end message = hop-by-hop message
  - Except for ‘‘Received-by’’ & warnings

MIME analogy falsely extended to HTTP

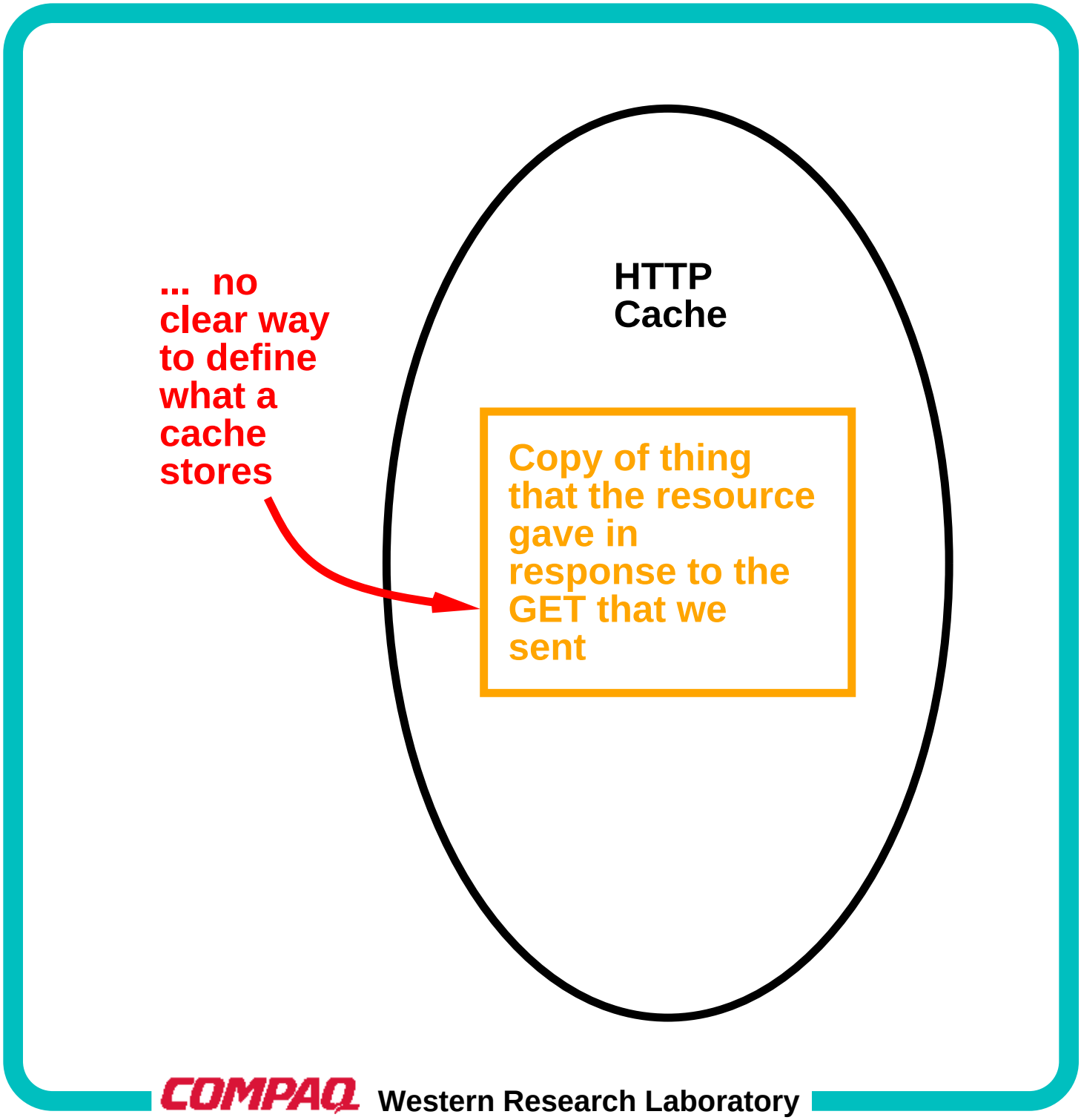
Official HTTP model:



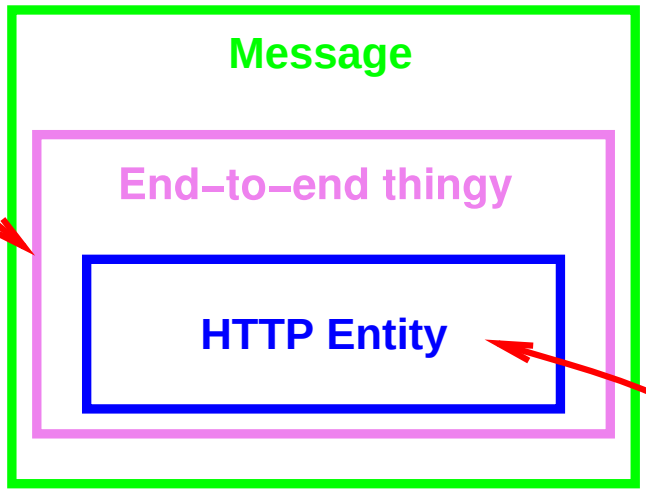
COMPAQ Western Research Laboratory



COMPAQ Western Research Laboratory


... no clear way to define what a cache stores
HTTP Cache
Copy of thing that the resource gave in response to the GET that we sent
COMPAQ
Western Research Laboratory

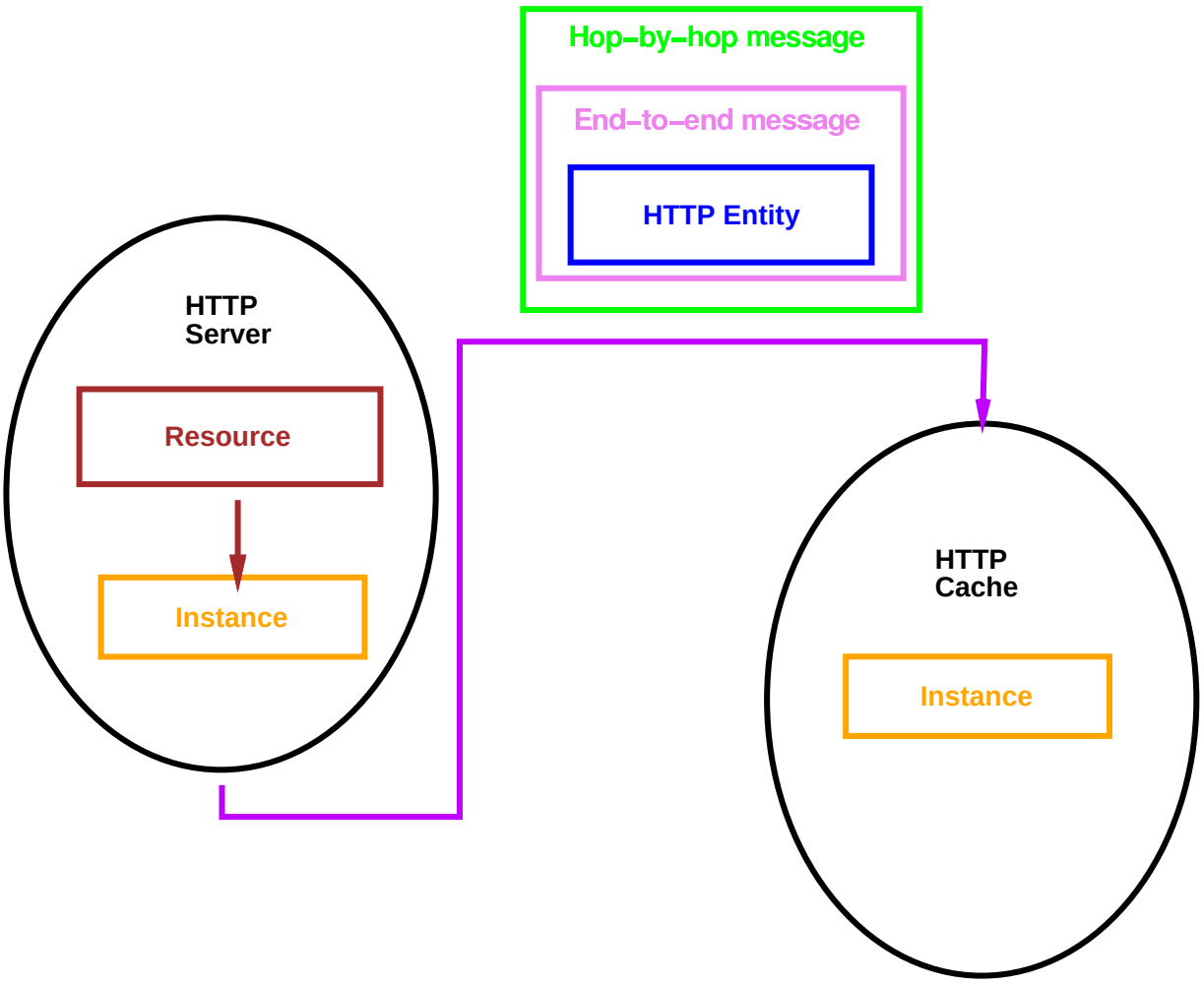
No distinction between end–to–end and hop–by–hop messages



Doesn’t match dictionary’s definition of term ‘‘entity’’

COMPAQ Western Research Laboratory

Clarified HTTP model: introduce ‘‘instances’’



Now we can discuss

- Transformations on instances
- What caches store

COMPAQ Western Research Laboratory

Data model - importance

Why is this important?

- Poor terminology leads to fuzzy thinking
- Poor terminology leads to underspecification
  - E.g., relationship between compression and byte-ranges
- Much confusion over whether headers apply to:
  - Server (or proxy)
  - Resource
  - Variant
  - Instance
  - Entity
  - End-to-end message
  - Hop-by-hop message

  with some headers filling several roles

Note: MIME's content-type system is fine with me.

Data model - example

Scenario:

1. Client loads first part of file, using `Range`
   - in response message #1
2. Client loads rest of file, using `Range`
   - in response message #2
3. Client assembles result from #1 & #2

Question: can we determine if result is correct?

What about `Content-MD5` header:

- Defined as ‘‘Digest of the entity-body’’
- ... so only applies to *message* #1 or #2

What we need:

- Digest of instance
- Not available in HTTP
  - because the concept wasn’t clear

Extensibility

Facts of life

- HTTP evolves
- Compatibility is mandatory
- ‘‘Flag days’’ are impossible

So: HTTP must be extensible:

- Interoperate with past implementations
  - Without degrading their behavior
- Interoperate with future implementations
  - No matter what the future brings
- Ambiguity kills

Three key issues

1. Protocol version numbering
2. Probing for extensions
3. The POST hole

HTTP Version numbers

HTTP version numbers verge on meaningless

- ‘‘HTTP/1.0’’ never really specified
- ‘‘HTTP/1.1’’ systems deployed before spec is finished
- Some proxies use the wrong version number
- Version number is officially hop-by-hop
  - though `Via` gives end-to-end version
- Optional features aren’t indicated by version

Basic problems:

- Distinction between hop-by-hop, end-to-end
- Lack of precision
  - optional features (MUST vs. SHOULD)
  - minor updates
  - ‘‘Proposed’’ vs. ‘‘Draft’’ vs. ‘‘standard’’
- Lack of formal binding to a specification

COMPAQ Western Research Laboratory

Version numbers - alternative?

Possible alternative approach:

- Use RFC #s instead of sequential versions
  - RFCs are immutable
  - Should be well-defined
- Create special ‘‘profile’’ RFCs as necessary
- Parameter indicates ‘‘conditional’’ compliance
- Use header (not first line) to carry info
  - Separate headers for e-2-e, hop-by-hop
  - Proxies leave audit trail (like `Via`)

Example:

```
HTTP/1.1 200 OK
Server-Version: RFC=2068, RFC=9914
Proxy-version: RFC=6234;cond
```

Maybe next time ...

Probing for extensions

Question:

- Does peer implement an extension?

Goals:

- Discover this reliably
- Fall back to standard protocol
- Don’t add too many round trips

One good thing: ‘‘ignore unknown headers’’ rule

- See if peer responds to new header
- Especially useful for optimizations

Complex extensions not adequately supported

- What precise extension (and options) is desired?
- How does this affect caching
- How are extensions named?
- What extensions do we expect, anyway?

Attempts:

- PEP (‘‘Protocol Extension Protocol’’): failed
- ‘‘HTTP Extension Framework’’: jury is still out

COMPAQ Western Research Laboratory

The POST hole

HTTP's POST method

- Sends bits to server
- ... but does not simply store into resource
- Basically, an arbitrary RPC
  - Typical use: complex HTML forms

A well-specified standard would ...

- allow ends to evolve without agreement
- allow proxies to intermediate

POST is just a big loophole

- No standard
- Too tempting to use instead of a true extension
- Problematic for security firewalls

Caching

Caching and proxies in HTTP/1.0:

- An afterthought, at best
- No cache coherency (especially for updates)
- No extensibility to new methods
- No detection of transparency failures

Transparency:

A cache behaves transparently when the response that you get from it is essentially the same as the response you would have received from the origin server.

Without transparency:

- Caches aren't trusted (and are bypassed)
- Users get wrong answers

How HTTP caching works

Basic model of HTTP caching:

1. Client A requests resource R via Proxy P
2. Proxy P forwards A's request to server for R
3. Proxy P receives server's response for R
   - Forwards response to A
   - Stores response for later use
4. Client B requests resource R via Proxy P
5. Proxy P retrieves & returns stored response

Conditional requests:

1. Server's response includes validator
   - e.g., last-mod date or (in 1.1) entity-tag
2. Client asks server if cache entry is fresh:

```
GET /foo.gif HTTP/1.0
If-Modified-Since:
        Thu, 03 Jun 1999 20:16:34 GMT
```

3. Server responds with either:
   - `200 OK` + full response body
   - `304 Not Modified` + no body

Caching - after HTTP/1.1

HTTP/1.1 improvements:

- Detection of transparency failures
- Better transparency, in general
  - More accurate mechanisms
  - Explicit control for exceptions

but these are still problems:

- Coherency for updatable resources
- Extensibility
- Complexity of rules & implementation
- Overall performance

Caching - coherency after updates

Problem: cache cannot detect update to resource

- If update is done local to server
- If update is done by client not using this cache

Implications:

- Expiration deadlines must be conservative
- ‘‘Distributed authoring’’ must disable caches

Several potential solutions:

- Callbacks (Chengjie Liu and Pei Cao)
- Volume-based validation (Cohen et al.)
- Cache applets (Cao, Jin Zhang, & Kevin Beach)

although all have drawbacks and limitations

Security: especially difficult

- Who is allowed to modify a cache entry?

Caching - complexity

HTTP/1.1 caching specification:

- Lots of caching-related rules
- Some are subtle
- Some are contradictory

This is due to

- Evolutionary design
- Competing notions of goodness

Cache implementations are complex:

- Because of complex specification
- Because of complex lookup mechanisms
- Because of inevitable engineering issues

Not clear how to resolve this

Caching - performance

Three reasons for caching

1. Faster response time
2. Lower bandwidth requirements ($$$)
3. Availability during disconnection

But

- Observed cache hit ratios are below 50%
- Byte-weighed hit ratios are even lower
- Lack of coherent ‘‘collection’’ concept

So simple caching has pretty much hit its limits

Can’t we do better?

Caching - paths to improved performance

Possible approaches:

- Prefetch, to hide latency
- Exploit (better) the bits already in the cache
- Decompose pages into static & dynamic parts
- Support coherent snap-shots

All require some level of HTTP enhancements

Also interesting: cache management issues

- Replacement hints
- Balancing the cost of disk I/O
- Cache cooperation
- Privacy considerations
- Hit-metering/ad-placement

would be nice to resolve these ...

Content negotiation

Important original broad goal:

- Make the Web international (multilingual)

Semi-failure, because of fuzziness about:

- Specific goals; especially, who's in charge?
  - User's preference
  - Site designer's preference
- Naming issues
- Confusion between negotiation axes:
  - content
  - presentation
  - implementation parameters
- Caching
- Importance of avoiding round-trips

Negotiation problem: deploying new codings

How to deploy new codings?

E.g.: client that groks ‘‘squish’’ compression sends:

```
Accept-encoding:gzip,compress,squish
```

Problem: must choose between

- Request header bloat

or

- Sluggish deployment of new codings

Underlying problem: no way for

- Client to ask server what it supports
- Server to tell client ‘‘*X* unsupported’’

Bad hacks:

- Server keys off of `User-agent` header
- Client sends `Accept: */*`

COMPAQ Western Research Laboratory

Status and error codes

HTTP responses carry one 3-digit status code; e.g.

```
HTTP/1.0 200 OK
HTTP/1.0 304 Not Modified
HTTP/1.0 404 Not Found
HTTP/1.0 501 Not Implemented
```

Problems:

- Some ambiguity in specification
  - ‘‘Which code should I use here?’’
- Only one code
  - What about non-fatal errors at proxies?
- Complex interaction with caching
- Not really extensible

Status codes - improvements

HTTP/1.1 adds `Warning` header; e.g.

```
HTTP/1.1 200 OK
Warning: 110 proxy.a.com
         "Response is stale"
Warning: 214 proxy.b.com
         "Transformation applied"
```

First digit controls cache behavior on revalidation

What might have been better:

- Status code as separate header(s)
- Specific indications for severity, caching
  - Consider automated clients
- Better multilingual support

Transport issues

Goal: efficient and reliable message transport

Issues:

- Inefficiency of ASCII header encoding
- Bias against compression
- Lack of clean connection-abort mechanism
- No buffer size limits/negotiation
- No multiplexing
- No multi-resource operations
- No atomic grouping of operations

Transport issues - HTTP/1.1 fixes

HTTP/1.1 has some improvements:

- Persistent connections, pipelining
- Compression negotiation
- Chunked encoding + careful rules
- "`100 Continue`" + `Expect` mechanism
  - This is complex, might be unreliable
- Weak form of atomic read-modify-write
  - Single resource only
  - Not really tested

Transport - efficiency

Inefficiency of ASCII header encoding

- Way too verbose, e.g.:

```
If-UnModified-Since: Thu, 03 Jun 1999 20:16:34 GM
```

  Mean header sizes (from traces):
  - Requests (with URLs): ca. 300 bytes
  - Responses: ca. 160 bytes

- Requires complex parser
- Could be simpler:
  - One or two bytes of header ‘‘name’’ code
  - Tokens with type, length code bytes
  - Binary codes for dates, integers, enums
- Could also be optional; either
  - Negotiated switch from ASCII to binary
  - Short abbrevs for header names, enums
- For example: replace header above by

```
IU: 3756E239
```

Transport - efficiency, continued

Bias against compression

- Default is uncompressed
  - images, media usually pre-compressed
- Poorly defined interation with other features
  - e.g., Range retrievals

Transport - reliability

Lack of clean connection-abort mechanism:

- If user hits Stop button, then:
  - TCP connection(s) lost
  - Possible loss of buffered data

No buffer size limits/negotiation

- Basic principle violated:
  - Can't send more than receiver can buffer
  - but HTTP has no explicit limits
- Specific bug found in HTTP/1.1 spec:
  - Involves proxies, chunking, HTTP/1.0
  - Required last-minute spec change
- HTTP should have
  - End-to-end, hop-by-hop limits
    - Reasonable default
    - + Negotiation mechanism?

Transport - message structure

No multiplexing

- Slow response stalls all others on connnection
- Would require some kind of transaction ID

No multi-resource operations

- Cannot operate on multiple resources, e.g.:
  - Multiple cache re-validations
  - GET-LIST of images on a page
- Can kludge using message headers
  - Not interoperable with extant proxies

No atomic grouping of operations, e.g:

- Get consistent set of resources
- Rename (via GET, PUT, DELETE)

(but WEBDAV adds this)

Cookies

Originally Netscape's *ad hoc* extension

RFC2109 proposed a standard

Problems:

- interoperation w/ non-standard cookie implementations adds complexity
- Privacy concerns

Conflict between technology, policy, & profit:

- IETF requires ‘‘Security Considerations’’
- but there's no consensus on how far to go
- Ad-supported sites have a lot to lose
    - Software vendors tend to play along

Other social issues

Copyright

- Is a cache liable for copyright violation?
- How can a proxy know what is legal?

Advertising

- Accurate counting without excess overhead
- Trust issues:
  - Are counts honest?
  - Are proxies surreptitiously replacing ads?
- Balance between:
  - Users want to refuse ads
  - Content-providers need ad revenue

Procedural problems in the HTTP-WG

- Length of process
- Porous criteria for feature inclusion
- Premature deployment
- Lack of resources for tedious work
- Some good points

HTTP-WG procedures

Length of process

- HTTP/1.1 took 4.5 years
- Lots of players joined relatively late
  - Or moved on (or got rich)
- Tendency to rush decisions
  - ... and then the process drags on
- Architectural issues tend to drift

Porous criteria for feature inclusion

- ‘‘Demonstrated necessity’’
- ... but sometimes based only in theory
- ‘‘Wait for HTTP-2.x’’ (or for HTTP/1.2)
- Multi-stage process might have helped

HTTP-WG Procedures - continued

Premature deployment

- RFC2068 (text: Aug. 96) treated as ‘‘standard’’
  - I.e., implementations deployed widely
  - Not a ‘‘standard’’ by normal IETF rules
- Precluded undoing design mistakes later on
- Special problem: RFC2068 inconsistencies
- Debate over whether to rename as ‘‘HTTP/1.2’’
  - Apparently, we won’t

Lack of resources for tedious work

- editorial work
  - checklists (required vs. recommended)
  - consistency
- test implementations
- rationale documentation

HTTP-WG Procedures - continued

Good points about the HTTP-WG process:

- The long process gave time to reflect
  - Many bugs found years after design phase
  - Overall, reached architectural consensus
- Vendors behaved themselves
  - No attempts to bias towards their code
  - Engineers cooperate better than marketers
- HTTP/1.1 much more specific than HTTP/1.0
  - Ambiguities rare
  - ... but still some ‘‘folklore’’ inferences
- Good balance of fixes vs. compatibility

Why the bugs in HTTP don’t matter

If technical excellence really mattered, then:

- FORTRAN
- Windows
- QWERTY keyboards
- VHS tapes

would be dead and buried

The bugs in HTTP don’t matter because:

- It works well enough
- It’s not the only game in town
- Revising it again would be too hard

HTTP - it works well enough

Lack of flexibility doesn't seem to be a problem

- Existence proof: zillions of Web sites

Poor cache coherence can be solved by

- Cache-busting
- ''For latest view, hit 'Shift-Reload' ''
- Not caring

Inefficiency might be irrelevant:

- Bandwidth keeps increasing
- CPU speed & RAM size follow Moore's Law
- Users are surprisingly patient
- Site designers will get a clue ... some day

''Wrong tool for the job'' argument

- Ignores human nature

COMPAQ Western Research Laboratory

HTTP - not the only game in town

‘‘HTTP doesn’t work for my application’’

- Intrinsically bad for:
  - Two-way initiation of operations
  - Real-time
  - Deferred delivery
- Argues for new protocol, not for fixing HTTP
- No single protocol can support every feature

For example:

- RealAudio/RTSP
- IRC, USENET, SMTP, NFS (old protocols)

‘‘HTTP Envy’’ (Mark Day’s term)

- ‘‘Driven by fantasies of HTTP-like ubiquity’’
- No reason to naively use HTTP for everything
- Properly leads to non-HTTP protocols

HTTP - too hard to revise it again

HTTP/1.1 took 4+ years

- Relative minor revisions

Barriers to new revisions of HTTP:

- Incompatibility would mean:
  - Hard to deploy new version
  - No experience to guide detailed design
- Incremental benefits, at best
  - Little incentive to deploy widely
- Large installed base
  - Older code stops dying off
- Last year's ‘‘trial’’ site now ‘‘mission-critical’’
  - Less tolerance for change, instability

Summary and Conclusions

Design by evolution

- Leads to adequate design
- But not optimal design

HTTP is

- Definitely not optimal
- Probably adequate

Caching - Prefetching

Prefetching principles

- Predict one or more future references
- Initiate prefetch if spare bandwidth available
- Cache prefetched result until needed

Prefetching issues

- How to make accurate predictions
  - Observations of this or other users
  - Parsing to find links and images
  - User-specified targets
- How to avoid congestion
  - Limit to non-shared links, when idle
  - Use feedback from TCP performance
  - Prefetch before scheduled uses
  - Ignore problem
- Protocol enhancements
  - Server-supplied hints
  - Mark prefetch requests as ‘‘low priority’’

Caching - Delta-encoding & Macros

Observations:

- Many resources change frequently
- Differences between instances often small

Therefore, in these cases

- Most of the ‘‘cached bits’’ actually useful
- Need: difference (delta) between instances

Delta-encoding:

- Transmit just the delta
- Requires extensions to convey instance info

HTML macros (Douglis, Haro, Rabinovich)

- Separate HTML into constant part, parameters
- Cache can hold constant (usually larger) part
- ‘‘Parameters’’ really just a macro call (Java?)

Caching - Duplicate suppression

Many responses have different URL, same content

Implications of such ‘‘duplicates’’:

- Cache miss yields data cache already has
- Cache ends up storing multiple copies

Automatic duplicate suppression would:

- Conserve bandwidth
- Reduce latency
- Possibly conserve cache storage

Proposals (all based on MD5 or SHA-1 digests):

- Link-level (Santos and Wetherall)
- DRP (van Hoff *et al.*)
- HTTP Duplicate Suppression (van Hoff & me)

Effectiveness? still an open question

- Probably 5% - 15% bandwidth savings

Caching - Coherent snapshots

Goal: avoid intra-page incoherence

- E.g., right text, wrong photo
- Or inter-page incoherence for collections

HTTP lacks mechanism for describing collections

- and for coordinated locking

Proposals:

- DRP: uses ‘‘index’’ Content-Type
  - MD5/SHA-1 digests for integrity
  - No locking mechanism
- WEBDAV: elaborate extensions
  - 96 pages
  - Does some other stuff, too