HTTP **Architecture** in Security Protocols

Wishful Thinking
“Just Use HTTP”

Photo by Arnaud Bertrande

Reality

Photo by John Haslam

Outline

Problem Statement

- Give information to help people use HTTP securely in protocol design
- Go beyond “use HTTP as a substrate” (BCP56) and cover “use HTTP”

Is there a Real Problem?

1. Standards using HTTP have been blocked by HTTP intermediaries
2. Application implementations ship without required HTTP features → interop FAIL
3. Existing tools made useless by stupid errors
4. Session hacks introduced security holes
5. Manageability (e.g. firewall filtering) seriously hampered by tunneling
6. Extensibility of HTTP lost in new standards

Pitfalls

- HTTP compliance is difficult to test particularly in the context of an application
- Many HTTP features are obscure
- HTTP intermediaries affect connectivity and interoperability but are obscure
- HTTP scalability and extensibility characteristics are easy to break
- Ignorance of HTTP architecture has cost us

Standards activity

- BCP56 mostly tries to discourage use of HTTP
- Let’s adopt a reality-based approach
  - Protocol police: accept that people extend HTTP in these ways
  - Protocol designers: accept that HTTP has intermediaries and backwards-compatibility issues
- Revising BCP 56
- Revising RFC 2616 – seven sections

Security benefits of HTTP fairway

Use Existing Software

- Less new code ≈ Less new bugs
- Easier to test, audit, manage
- Known security properties
- Lots of expertise already out there

Where does this go wrong?

Architectural Styles & Features

Caching	Messaging	Queued
Store and forward	Pipe and filter	RPC and RMI
Sessions	Pub/Sub	Transaction
Client/ server	Request/ response	Connection oriented
Federated	Proxied	Synchronous

Protocol Architectural Styles

Caching	Messag	[illegible]
Store and forward	Pipe an	RMI
Sessions	Pub/Sub	P2P
Client/ server	**Request/ response**	Connection oriented
Federated	**Proxied**	**Synchronous**

REST is all these things:

Characteristic	Explanation/ Consequence
Request/Response (Client/Server)	Useful mostly for client-initiated interactions
Stateless on server	Server is scalable if this property is preserved
Cacheable	Better performance
Uniform Interface	Proper use of uniform GET semantics enables caching; server control of URIs grants resiliency
Proxied	Proper support for intermediaries enables caching

REST

NOT just a buzzword

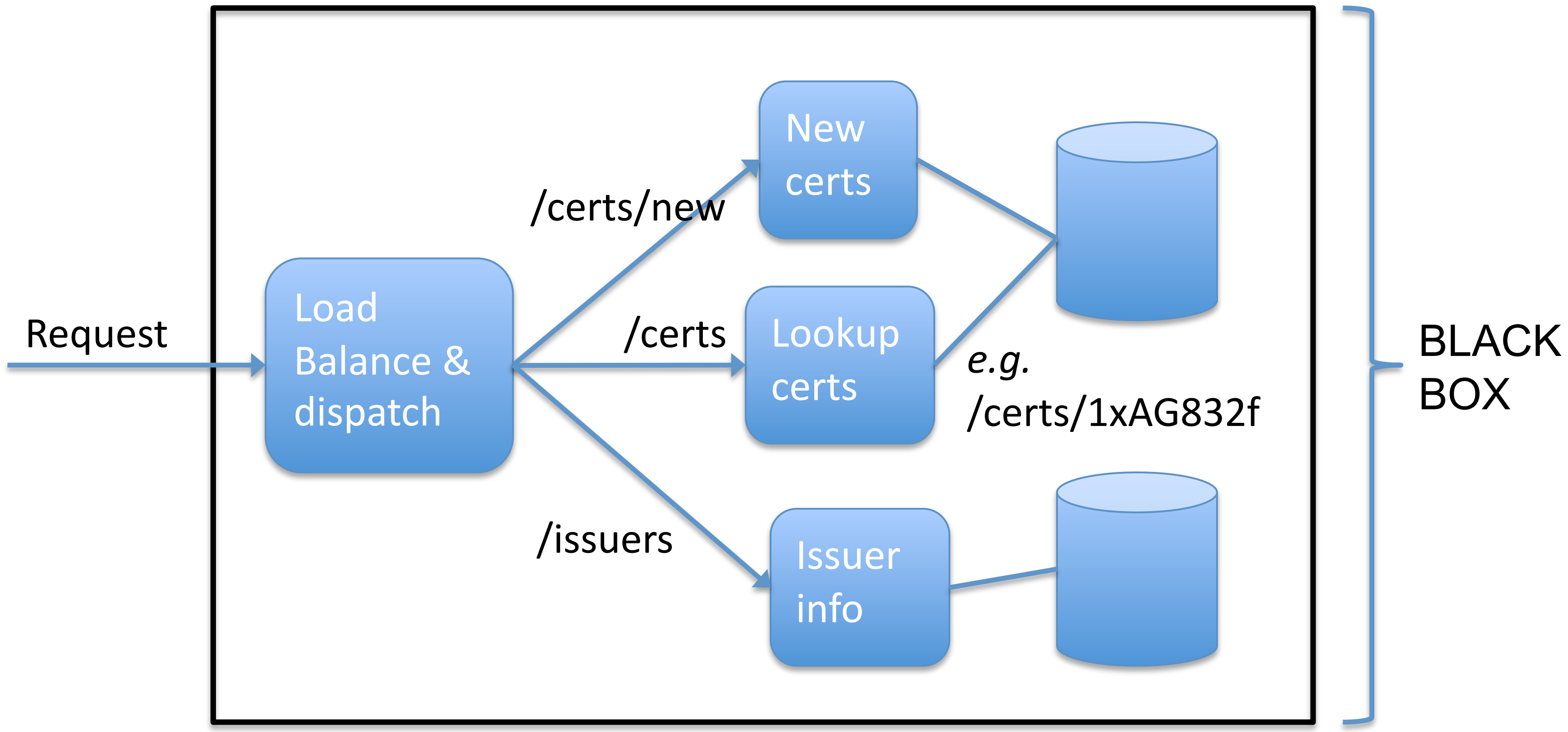
Quite useful for some application models

1. Can the application have persistent resources?
2. Can the resources be assigned persistent URLs?

➔ If yes to both: it could be a good match

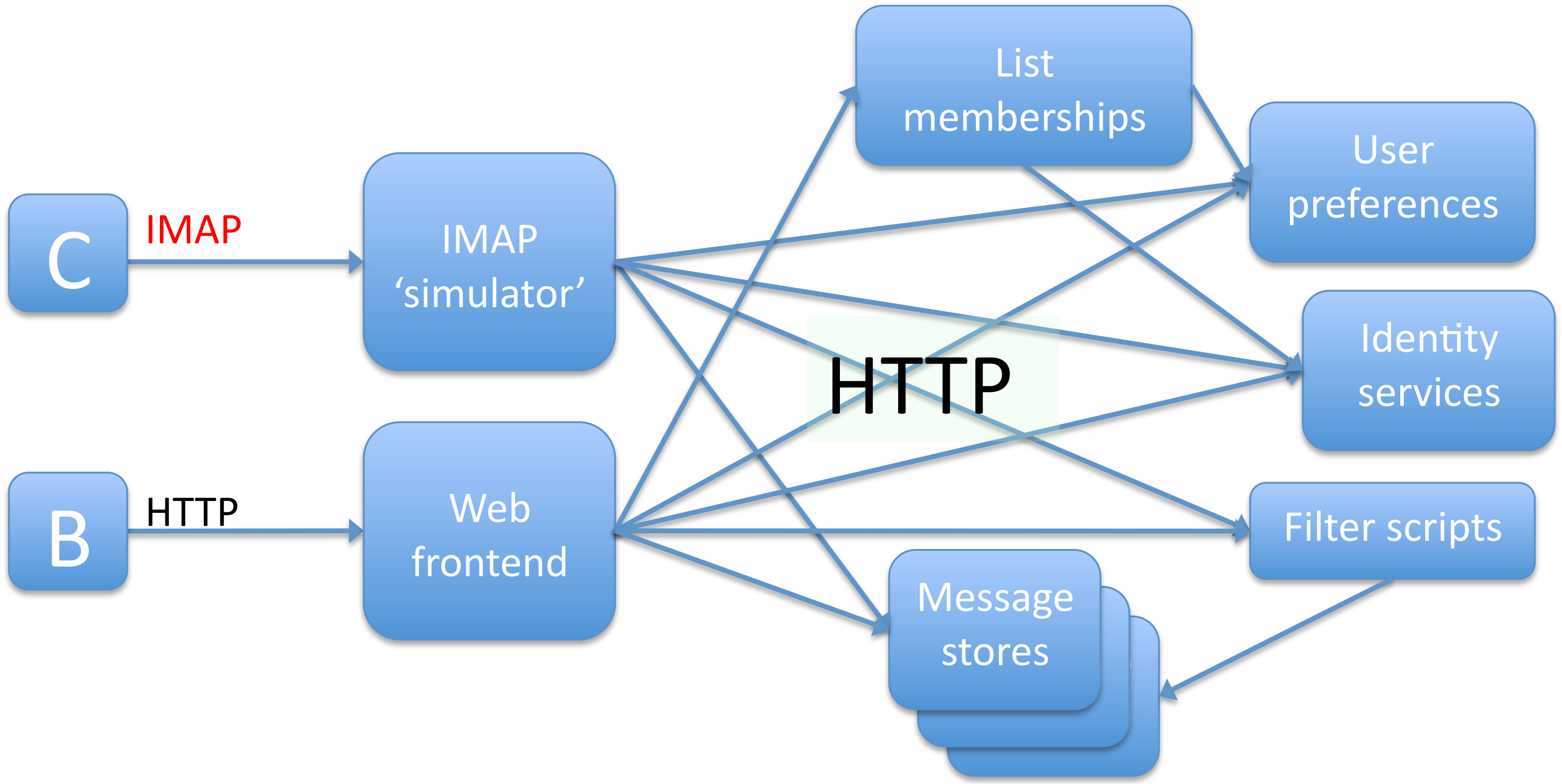
Benefits of REST + HTTP part 1

- Allows implementers to choose compartmentalization
  - URL namespace can align with components
  - Scalability is cheap with tiered system architecture



More on Internal Scalability

Why are we seeing HTTP in the backend? Hypothetical:



REST allows loose coupling and independently scaling services

Benefits of REST + HTTP part 2

- Extensibility along many axes is fairly clear
  - “You just bought yourself a future”
- Caching, conditional features often useful
- Extended HTTP features:
  - WebDAV permissions
  - WebDAV collection management
  - Atom feeds
  - Compression or deltas
  - PATCH

Benefits of REST + HTTP part 3

- Implementors leverage existing software
  - Web frameworks increasingly promote REST
  - Scalability tools like load balancers and memcache
- Deployed servers mesh well with Web GUIs
  - AJAX applications can use a REST-based standard to retrieve data
  - Data can be updated frequently while presentation and processing instructions are not
- Goes through firewalls

Drawbacks of REST

- A uniform interface is not optimized
  - Lower efficiency, because information is transferred in a standardized form rather than just exactly what that client needs at that second
- Not suitable for all application use cases
  - E.g. notifications from server to client
- Layered systems add overhead and latency for all new (uncached) data

Benefit or Drawback? Security

HTTP already integrates authentication and integrity

- Basic Auth and Digest auth
- TLS layering somewhat sketchy
- HTTP vs. HTTPS URI schemes awkward

NOT REST: RPC style

IETF sees proposals of the form: *"Protocol will use procedure calls (or messages) with bindings to SOAP, HTTP and other transports"*

- An approach that is fine for enterprise apps
- Bad for standards and interoperability because
  - ☹ Different transport choices
  - ☹ RPC has minimal interaction model
    - ☹ *Tosses out the resource model*
  - ☹ Procedure calls means tight coupling
  - ☹ Poor match to HTTP model

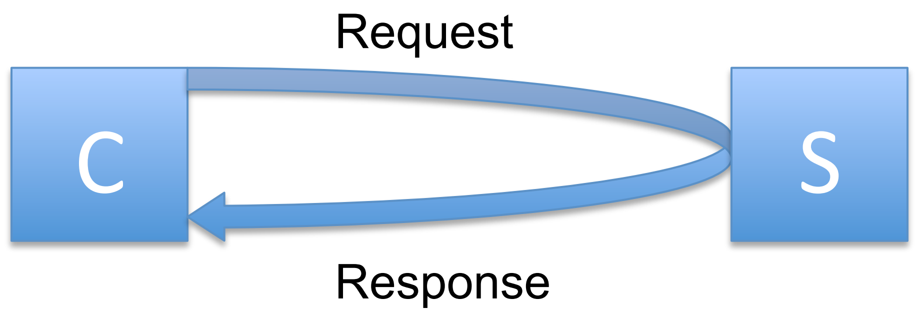
What the IETF knows about APIs

- APIs are harder to standardize than protocols
- APIs are tied to languages and platforms
- APIs are tied to implementation choices
- A decent protocol can last much longer than all but the very best APIs

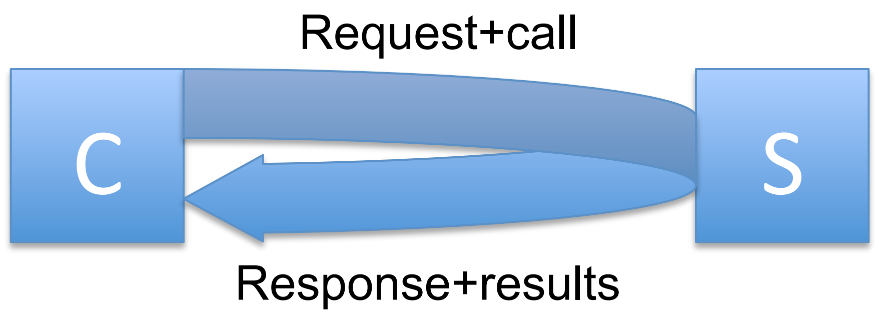
So why do we keep building programming interfaces that masquerade as protocols?

Interaction control vs. resource control

Message model gives client interaction control:

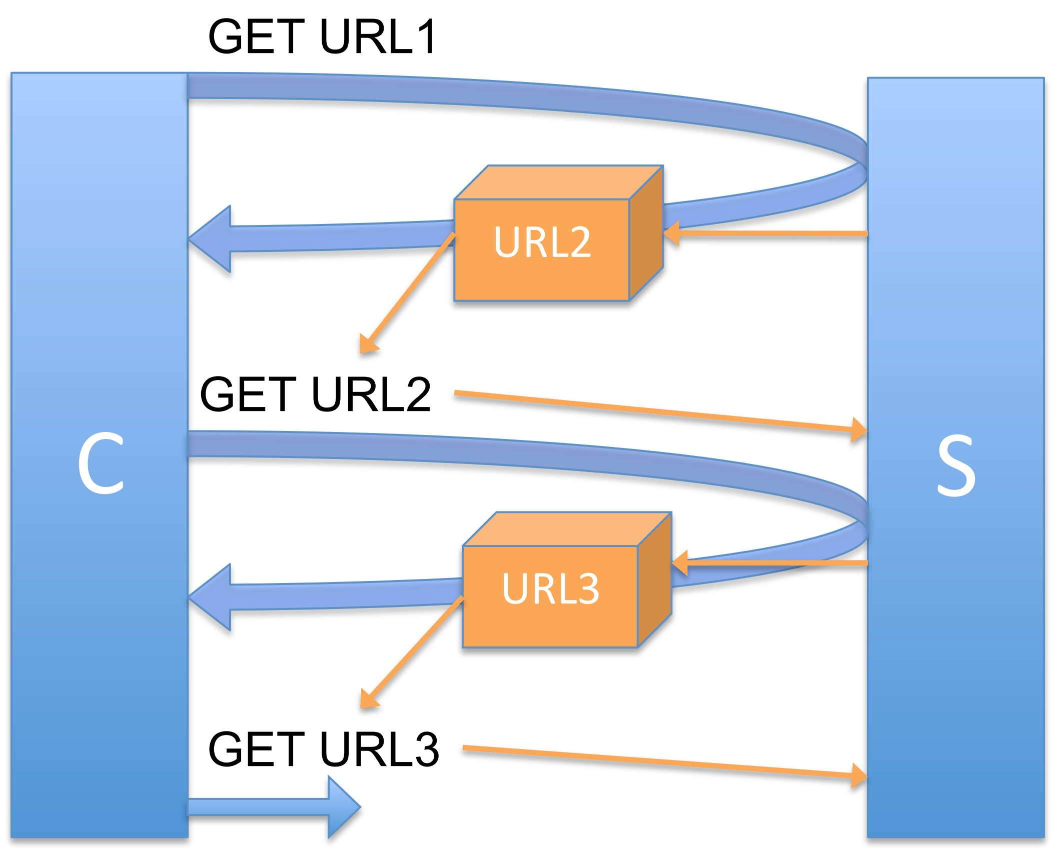


Adding procedure calls reinforces client control:



Resource Control

- REST gives resource control to the server



Contrast RPC Style

REST	RPC
GET /users	getUsers()
GET /newusers	getUsersSince(date d)
GET /users/l	getUsersMatching(string regex)
POST /newuser *<body>*	createUser(*params…*)
POST /data/u2/0037 *<body>*	updateUser(*params…*)

With REST: server controls naming; “newusers” and “users/l” could be discoverable resources with known syntax (basically fixed queries).

- Fixed queries can be optimized and cached
- … but you really have to know your use cases

"But I'm Using RPC Style Anyway"

In that case:

1. Don't use SOAP – it's not suited for standards (interoperable independent implementations)
2. Can use POST with a new MIME type that allows identification of your application
3. Disable caching, content negotiation
4. Consider carefully what role URLs will play

RMI Style may be slightly better

Remote Method Invocation, where object=resource

REST	RMI
GET /users	GET /users/
GET /newusers	POST /users/?since=d
GET /users/l	POST /users/?matching=regex
POST /newuser *<body>*	POST /users *<body>*
POST /data/u2/0037 *<body>*	POST /users/lisa *<body>*

- Note that the server is still losing control of the interaction
- Responsibility for constructing fast and scalable queries now is in the clients' hands– wrong place!

Compliance & other Rules

- Application servers still have to be HTTP compliant
- So do application clients
- Protocols need to use registries and use code points appropriately

Server Compliance

- MUST respond to the HEAD request properly (no body)
- MUST handle OPTIONS <path> and OPTIONS * requests.
- MUST use an error responding to unrecognized methods.
- MUST handle conditional headers on requests (If-* and If)
- MUST honour Content-* headers on requests.
- MUST handle the Range header or fail the request.
- MUST look for the Expect header and be able to do 100 Continue or fail
- MUST either support persistent connections and pipelining, or include the "close" connection option in every response.

Good news: a good server platform does all this

Client Compliance

- MUST include a Host header on requests
- MUST support end-of-message handling: chunked transfer-encoding, connection closing, and Content-Length.
- MUST either support persistent connections or include the "close” connection option in every request.
- If caching, MUST handle Vary, Cache-Control, Expires headers.
- MUST NOT automatically follow redirects for methods other than GET and HEAD.
- MUST handle 2xx responses as successes (202, 203, 205)
- MUST handle the 407 Proxy Authentication Required response and be able to use Proxy-Authenticate.

Good news: a good client library does all this

Protocol Design Compliance

- Status Codes
  - Use codes the same way they're already used to work well with HTTP client libraries.
  - Don't define new ones unless broadly applicable
    - Use the Status Code registry
  - Use internal status codes for application status
- Caching must be appropriate or disabled on every response to work with intermediaries

Security Considerations

- Security “Murphy’s Law”: If somebody can do something bad, they eventually will.
  - Thus: constrain implementers fairly carefully.
  - E.g. limit allowed MIME types
- New code ≈ new bugs
  - So avoid using new code
- Misunderstanding → misuse → hacks → holes

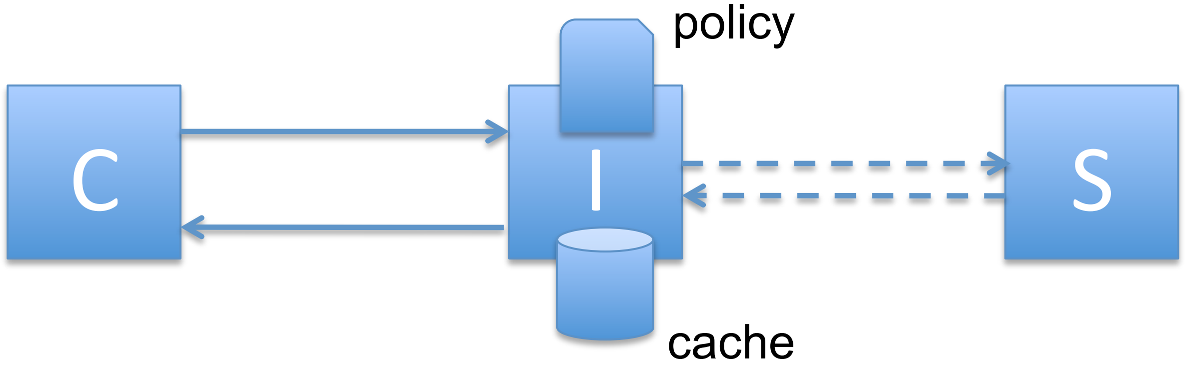
Authentication

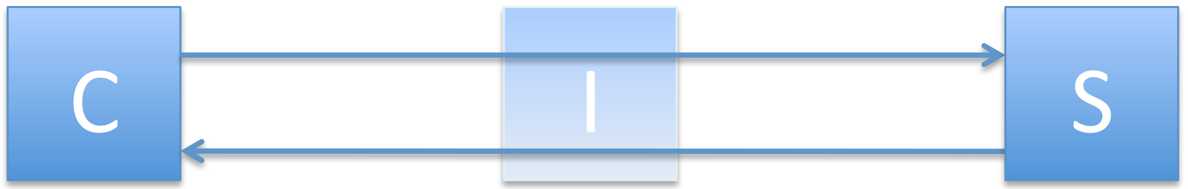
- Specify whether authentication via BASIC and DIGEST are required or forbidden
- Channel bindings: HTTP authentication is not bound to TLS encryption layer

State, sessions

- Stateless means there is no state
  - You can't release locks at the end of a "session"
- **Forbid Cookies** in applications that don't need sessions
- Require session tokens to be used securely if application really needs sessions
  - E.g. Cookies over TLS would maintain a session ID securely and cheaply

To layer or tunnel





CONNECT (all TLS traffic)

“Fairway” cheat sheet for safe use

- Full compliance to RFC2616
- Follow HTTP model where possible, using:
  - GET for cachable information responses
  - PUT to update resources
  - POST for custom processing
- Let servers control URIs
- Don’t use sessions, cookies; be clear about tunneling
- Require specific authentication and encryption

End notes

- Architecture is important even if opinions and taste differ
- This presentation is intended to continue the discussion, not to end it
- Consult with HTTP experts; e.g. ask for App Area review

Material for readers of this presentation

- References
- Choice quotes

References

- Architectural Styles and the Design of Network-based Software Architectures
  - Roy Fielding, 2000
- BCP56 On the Use of HTTP as a Substrate
- RFC2817 for CONNECT and TLS tunneling

At no time whatsoever do the server or client software need to know or understand the meaning of a URI — they merely act as a conduit through which the creator of a resource (a human naming authority) can associate representations with the semantics identified by the URI. In other words, there are no resources on the server; just mechanisms that supply answers across an abstract interface defined by resources.

It may seem odd, but this is the essence of what makes the Web work across so many different implementations. It is the nature of every engineer to define things in terms of the characteristics of the components that will be used to compose the finished product. The Web doesn't work that way. The Web architecture consists of constraints on the communication model between components, based on the role of each component during an application action. This prevents the components from assuming anything beyond the resource abstraction, thus hiding the actual mechanisms on either side of the abstract interface.

Roy Fielding, PhD. Dissertation

Wikipedia entry on SOAP

“Most uses of HTTP as a transport protocol are done in ignorance of how the operation would be modeled in HTTP. This is by design—similar to how different protocols sit on top of each other in the IP stack. But this analogy is imperfect; the application protocols used as transport protocols aren't really transport protocols. As a result, there is no way to know if the method used is appropriate to the operation. This makes good analysis at the application-protocol level problematic with sub-optimal results—for example, a POST operation is used when it would more naturally be modeled as a GET. The REST architecture has become a web service alternative that makes appropriate use of HTTP's defined methods.”