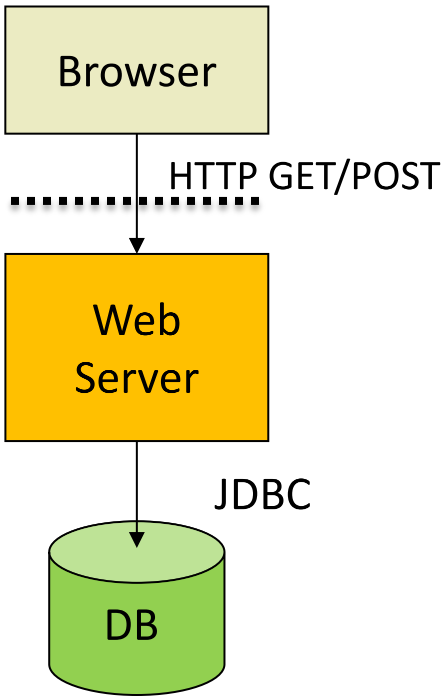
Web Services (WS)
and
Service-Oriented Architecture (SOA)
-
From Web Pages to Enterprise Computing

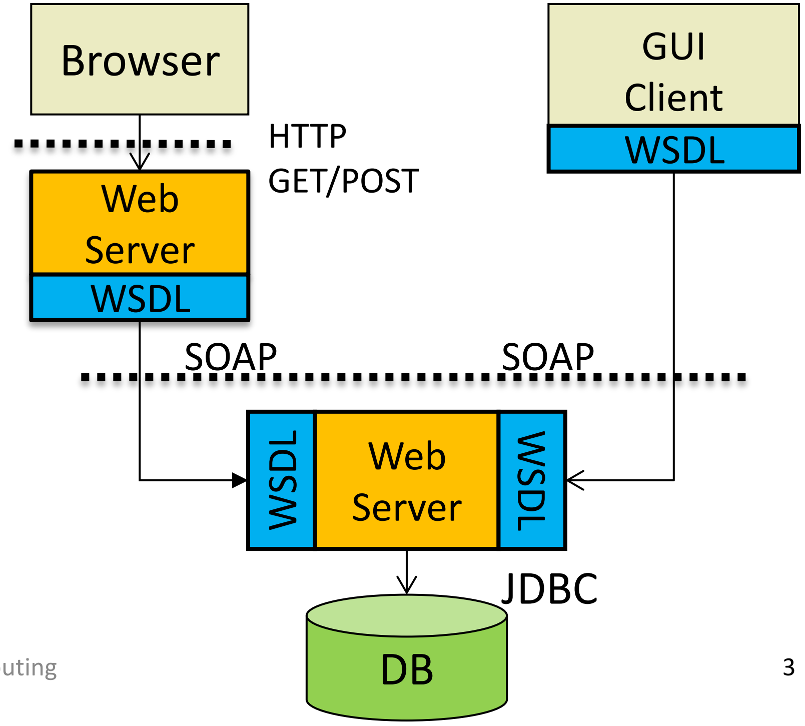
What is Good about HTTP?

- It is inefficient
  - Messages back and forth are all in plain text
  - Multiple rounds of interactions to complete a request
- However, it is so popular and important that …
  - It is supported in most enterprise servers
  - Most security agents and firewalls let http go through
    - After all, messages are plain texts, what harms could they do!?
  - It is a standard: independent of vendors, languages and platforms
- Result: HTTP is the information/message highway of enterprise systems
- Web is still regarded as a point-to-point, request-response system, returning “information” to the user.

Web Services

- Web services is for machine-machine communication
  - Interactions may be either through browsers or special clients
  - Platform, technology and programming language-independent
  - Built on existing Web standards: HTTP for transport





Web Service Components

- XML-based distributed services system based on:
  - Web Service Description Language (WSDL)
    - Describes how the service is to be used and for writing APIs
  - Simple Object Access Protocol (SOAP)
    - An envelope for transferring messages
  - Universal Description, Discovery and Integration (UDDI)
    - Registry for listing and discovering web services

Web Services Description Language

- Operational information about the service
  - Location (URI) of the service
  - Service interface: the set of functions supported by the WS and the formats (messages and their parameters) to request the service
- By parsing this WSDL file, other programs can invoke this Web service easily.
- Include meta-data on service capability for human users to read

WSDL

- *<types>* element
  - Define all complex data types not builtin in XML Schema
- *<message>* element
  - Messages used in WS
- *<portType>* element
  - Operation names, request and response messages
- *<binding>* element
  - Protocol and message format used by the web service (e.g., SOAP)

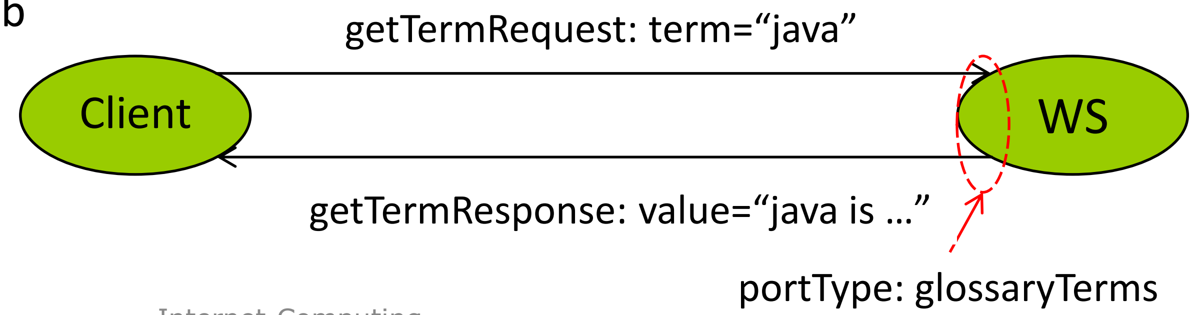
```
<portType name="glossaryTerms">
  <operation name="getTerm">
    <input  message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

Operation / method

Messages

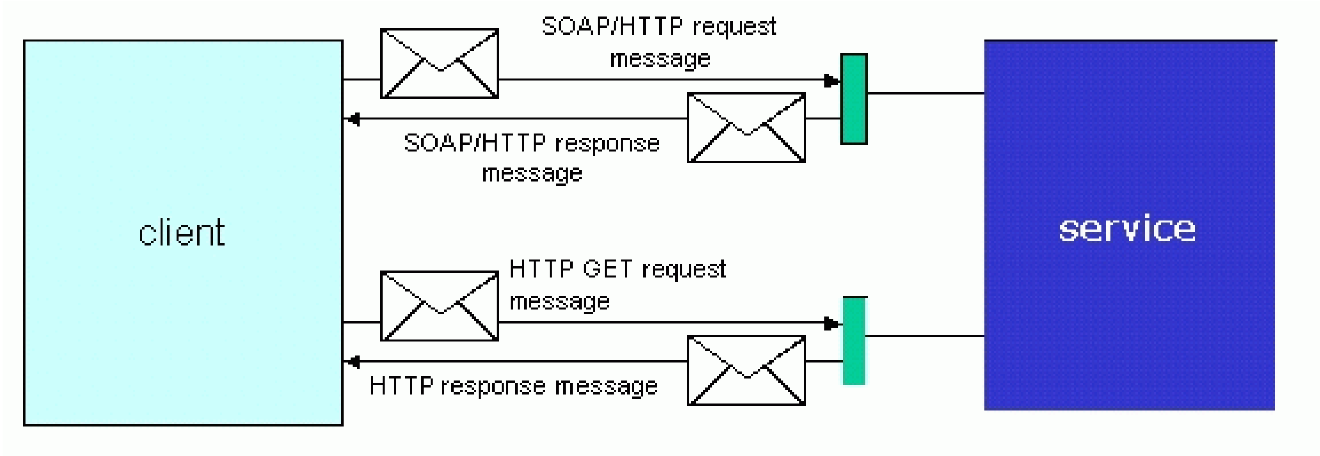
```
<message name="getTermRequest">
  <part name="term" type="string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="string"/>
</message>
```

Method parameters


getTermRequest: term="java"
Client
WS
getTermResponse: value="java is …"
portType: glossaryTerms

SOAP

- WS messaging format for request and response
  - Request invokes a method on a remote object
  - Response returns result of running the method
  - Think of RPC: function names, parameters and return types
- SOAP allows XML documents of any type, e.g.,
  - Send a purchase order document to the inbox of B2B partner
  - Expect to receive shipping and exceptions report as response
- Ship on HTTP



SOAP Request Message

```
<?xml version="1.0"?>
<soap:Envelope
 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.stock.org/stock">

    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>

  </soap:Body>
</soap:Envelope>
```

SOAP Envelope Namespace

Message Namespace

Message

SOAP Envelope

SOAP Response Message

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.stock.org/stock">

     <m:GetStockPriceResponse>
       <m:Price>34.5</m:Price>
     </m:GetStockPriceResponse>

  </soap:Body>
</soap:Envelope>
```

Message

Result returned in Body

SOAP Envelope

Companies Supporting Web Services

- Google Map API
- Amazon Web Service (AWS)
  - Amazon E-Commerce Service: Build your own store front on Amazon’s products
    - Search catalog, retrieve product information, images and customer reviews
    - Retrieve wish list, wedding registry…
    - Search seller and offer
- What can you do using both?
  - Call this “mashing” or service composition

Take Home Messages

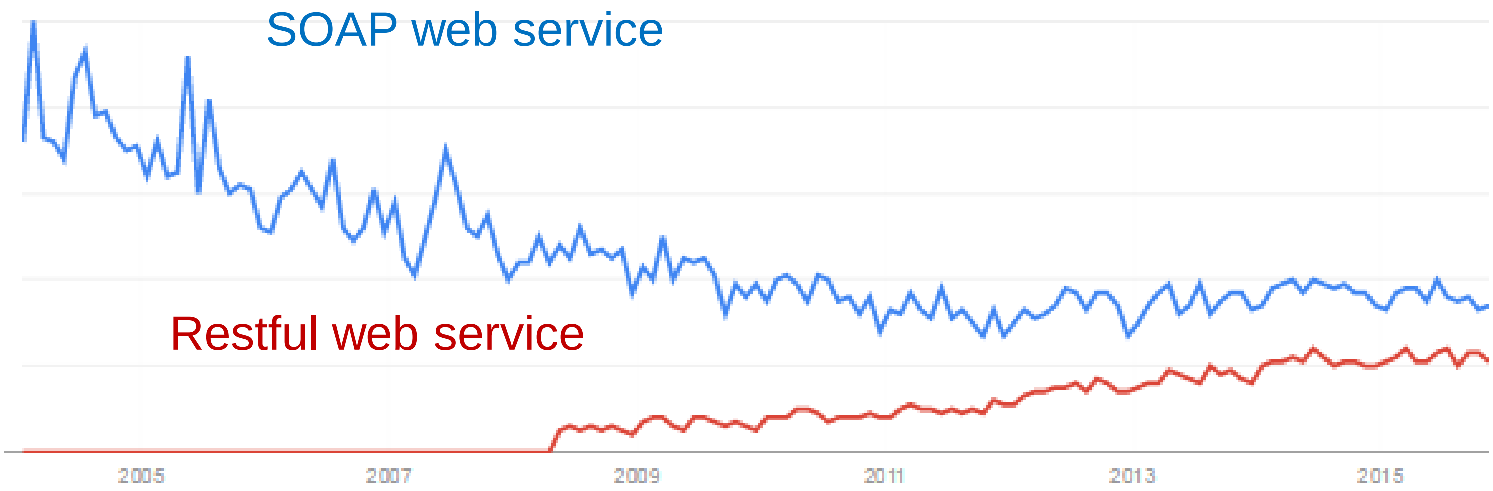
- HTTP from a freeway system of the internet: open, transparent, easy to use, reliable, …
- Large enterprise applications are broken down into “services”
  - That are loosely coupled and built on HTTP and XML data
  - Learn Service Oriented Architecture (SOA) in the future
- Applications can call external Web Services (Google Search API, Amazon, etc.) without worrying about where they are located, what platforms they use, …
  - All you need to do is to send a request and get a response
- Again, standard, standard, standard …
  - UDDI (Universal Description, Discovery, and Integration), SOAP and WSDL are XML based standards

Web Application Design Principles – REST (**RE**presentational **S**tate **T**ransfer)

Dik Lun Lee

Google Trend

- REST was coined by Roy Thomas Fielding in his 2000 PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures,“ University of California, Irvine
- Get increasing attention in web community



REST (**RE**presentational **S**tate **T**ransfer)

- REST is:
  - Not an architecture for building systems
  - Not a programming language or programming methodology
  - Not a framework, not a library, not a tool kit, …
- REST is a set of design criteria for interaction between two independent systems
  - It encourage a "new" way of thinking about the web (somewhat philosophical)
- REST is not tied to the ‘Web’ or HTTP, etc.
  - Just that HTTP 1.1 was designed with REST in mind and has been a very popular protocol
  - REST principles can be applied to other protocol

REST Principles

- Resources are identified by uniform resource identifiers (URIs)
- Resources are manipulated through their representations
- Multiple representations are accepted or sent
- Messages are self-descriptive and stateless

  - **Try to work with, not against, these principles**

Rest #1: Resources, Not Pages

- “Resource” is an abstract concept
  - Anything that is uniquely addressable and returns “some” information"
  - It could return a “page” that is statically stored in a server or dynamically generated by a server program
  - User will never "see" a resource, but rather a representation of it (e.g., in HTML; see later REST principle)
- Resources can be addressed with a URL (Universal Resource Locator) or URI (Universal Resource Identifier)
  - URL and URI can be considered the same; just conceptual difference
- A resource may have multiple URIs
- A URI must refer a unique resource
- URIs should be descriptive and have understandable structure

So, resource and URIs is 1:N

REST #2: Statelessness

- Every HTTP request is in complete isolation
  - The meaning of a request does not depend on prior requests
  - There is no specific ‘ordering’ of client requests (i.e. page 2 may be requested before page 1)
  - The server can restart and a client can resend the request and continue from where it was left off
- States are maintained as part of the content transferred from client to server and server to client

REST #3: Representations

- The client does NOT fetch a resource but one of the representations made available by a resource
- A representation of a resource is a sequence of bytes and headers to describe those bytes.
- The particular form of the representation can be negotiated between REST components:
- Client sets specific HTTP request headers to signal what representations it's willing to accept
  - **Accept:** XML/JSON, HTML, PDF, PPT, DOCX...
  - **Accept-Language:** English, Spanish, Hindi, Portuguese…

Is a web page (that can be displayed on your browser) a resource?

REST #4: Uniform Interface

- Provides 4 basic methods for CRUD (create, read, update, delete)

Method	Function	Response
GET	Retrieve representation of resource	Returns representation of resource
PUT	Update existing or create a new resource	Responds with status message or copy of representation or nothing at all
POST	Create a new resource under some 'parent' resource (e.g., Add new messages to forum)	Returns status message or copy of representation or nothing at all
DELETE	Delete an existing resource	Returns status message or nothing at all

- All of GET/POST/PUT/DELETE can be applied to all resources (of course, server can choose to ignore any one of them)
  - E.g., http://course.ust.hk?id=comp4021&op=delete
  - Or, http://course.ust.hk/delete?id=comp4021 (better but still not good)

Safety & Idempotence

- **Safety:** The request doesn't change state of resource; NO SIDE EFFECT
  - Making 10 requests is same as making one or none at all
  - GET and HEAD requests are **safe**
  - POST is NOT safe
- **Idempotence:** Executing the same operation multiple times is the same as executing it once
  - Deleting an already DELETE-ed resource is still deleted
  - Updating an already updated resource with PUT has no effect
  - GET, HEAD, PUT, DELETE are **idempotent**
- POST is neither safe nor idempotent
- Safety & idempotence are good for caching, bookmarking, reliability and scalability

Steps to a RESTful Architecture

1. Identify the data set for your application
2. Split the data set into resources
3. Name resources with URIs
4. Expose a subset of uniform interface
5. Design representation(s) accepted from client (Form-data, JSON, XML to be sent to server)
6. Design representation(s) served to client (file-format, language and/or (which) status message to be sent)

Surprise: You start with the resources and representations, NOT PHP, HTML, mySQL, etc.

Benefits of RESTful Design

- Clients can easily survive a server restart (state controlled by client instead of server)
- Easy load balancing – since requests are independent they can be handled by different servers
- Scalability: As simple as connecting more servers
- Stateless applications are easier to cache – applications do not have to worry about the ‘state’ of a previous request
- Bookmark-able URIs/Application States
- HTTP is stateless by default – developing applications around it gets above benefits (unless you wish to break them on purpose)

All operations can be misused: Use GET to update resource and use POST or PUT to retrieve representation of a resource

Take Home Messages

- REST is a set of design principles for client-server systems
  - Web in the 90’s was very simple and ad hoc, leading to web system developers to take shortcuts and do arbitrary things
  - REST attempts to set things straight
- REST is gaining popularity over w3c web service standard (too complicated)