Notes on Web/Enterprise Technologies

cjeong@live.com

Table of Contents

1 Foundations

1.1 HTTP

- current standard: HTTP 1.1 (RFC 2616)
- application protocol for distributed, collaborative, *hypermedia* information systems
- **HTTP *"session"***: sequence of network request-response transactions
  - TCP connection (typically) to port 80
  - HTTP server listening on port 80 waits for client's request "message"
  - upon receipt of a request, server sends back a status line (e.g. `"HTTP/1.1 200 OK"`); body of the message is typically the **requested resource** or an error message (or else: *what else could be?* **CODE** *or* **CONTINUATION***?*)
- defines 9 **methods** (verbs) to indicate desired action to be performed on "*resources*"

1. **HEAD**: asks for the response identical to the one that would correspond to a GET request but without the response body; this useful for retrieving *meta-information* written in response headers, without having to transport the entire content
2. **GET**: requests a representation of the specified resource
3. **POST**: submits data to be processed to the identified resource (submitted data is included in the body of request)
4. **PUT**: uploads a representation of specified resource
5. **DELETE**: deletes the specified resource
6. **TRACE**: echoes back the received request, so that a client can see what (if any) changes or additions have been made by intermedite servers
7. **OPTIONS**: returns the HTTP methods that the server supports for specified URL; this can be used to check the functionality of a web server by requesting '*' instead of a specific resource
8. **CONNECT**: converts the request connection to a transparent **TCP/IP tunnel**, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy
9. **PATCH**: used to apply partial modifications to a resource

1.2 HTML

- HTML (hytertext markup language) is the main language for **web pages**, where HTM elements are the basic building-blocks of webpages
- **HTML element** consists of **tags** enclosed in *angle brakets* (e.g. `<html>`); HTML tags commonly come in pairs

1.2.1 HTML DTD (Document Type Definition)

- a **doctype** (document type declaration) at the beginning of HTML specifies which SGML-based DTD (document type definition) to use for validation of this document
- e.g. `<!DOCTYPE HTML "-//W3C//DTD HTML 4.01"..>`
- usually, this line is used by browsers on which mode to render the document (e.g. standard mode vs quirks (backward-compatible) mode)

1.2.2 XHTML

- To be **deprecated** (to be replaced by **HTML5**, which supports XML and is endorsed by W3C)

1.2.3 HTML5

- more support for XML; media-friendly; reduce the need for "Flash"
- **HTML5 = Markup + JavaScript APIs + CSS**

- HTML is mostly about "interconnection", now in HTML5, we are more concerned with **first-class handling** of **audio**, **image**, and **video**, etc. (not just inlining into webpages)
  - audio, video, etc. can be "manipulated" from JavaScript
- in summary, **graphic toolkit** will be overrided by HTML5 and HTML5 will be the presentation layer of applications
- also, page elements will be programmable (reified)
  - HTML page is a "iteratable" container of page elements; can be modified/updated
- **How HTML works**
  - the **browser** loads a document, which includes a markup in HTML and sytle in CSS
  - when browser loads a page, it creates **internal model of the document** which is a "tree of markup elements"
  - browser also loads **JavaScript code** and begins execution after the page loads
    * using JavaScript, we can interact with the page by manipulating the DOM, react to user or browser-generated events, or make use of all new HTML5 JavaScript APIs
    * HTML5 JavaScript APIs include: **sockets, web workers, forms, audio, video, local storage, offline caching, canvas, drag and drop, geolocation, forms**

1.3 XML

- XML (eXtensible Markup Language) is used to *transport/store* data while HTML is used to *display* data
- XML documents for a **tree structure**
- **XML element**: everything from start tag to end tag; can contain
  1. another element (recursively),
  2. **text**, or
  3. **attribute**
- using **attributes** can make update of XML schemas difficult
- **well-formed XML documents**: satisfies syntactic constraints (e.g. matches tags, exactly one root leement, proper nesting)
- **valid XML documents**: well-formed + conformity to **DTD**

1.3.1 XML DTD (Document Type Definition)

- used to define the structure of XML document, e.g.

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

1.3.2 XML Schema

- XML-based alternative to DTD

```
<xs:element name="note">
<xs:complexType>
  <xs:sequence>
    <xs:element name="to" type="xs:string"/>
    <xs:element name="from" type="xs:string"/>
    <xs:element name="heading" type="xs:string"/>
    <xs:element name="body" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
</xs:element>
```

- Pattern #1: **Russian Doll**

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="title"
                   type="xsd:string"/>
      <xsd:element name="price"
                   type="xsd:decimal"/>
      <xsd:element name="category"
                   type="xsd:NCName"/>
      <xsd:choice>
        <xsd:elemnet name="author"
                     type="xsd:string"/>
        <xsd:element name="authors">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="author"
                           type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

- Pattern #2: **Salami Slice**

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="tns:title" />
      <xsd:element ref="tns:author" />
      <xsd:element ref="tns:category" />
      <xsd:element ref="tns:price" />
    </xsd:sequence>
  </xsd:complexType>
```

```
</xsd:element>
<xsd:element name="title"/>
<xsd:element name="price"/>
<xsd:element name="category"/>
<xsd:element name="author"/>
```

- Pattern #3: **Venetian Blind**

```
<xsd:element name="book" type="tns:BookType"/>
<xsd:complexType name="BookType">
  <xsd:sequence>
    <xsd:element name="title"
                 type="tns:TitleType" />
    <xsd:element name="author"
                 type="tns:AuthorType" />
    <xsd:element name="category"
                 type="tns:CategoryType" />
    <xsd:element name="price"
                 type="tns:PriceType" />
  </xsd:sequence>
  <xsd:simpleType name="TitleType">
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="1"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:complexType>
```

- Pattern #4: **Garden of Eden**

1.3.3 CSS (Cascading Style Sheets) & CSS3

- not the most common method for formatting XML
- W3C recommended using **XSLT** instead
- *However, what about CSS3 w/ HTML5?*

1.3.4 XSLT (eXtensible Stylesheet Language Transformations)

- sort of language used to transform XML data into displayable HTML-format data
- e.g.

1.3.5 XML DOM (Document Object Model)

- DOM defines a standard way of accessing and manipulating XML documents (**get, change, add, delete XML *elements***)
- DOM is an *internal(-to-browser) representation* for XML document, represented as a tree-structure
- **DOM nodes**:
  - document node
  - element node
  - test node
  - attribute node
  - comment node
- browsers have built-in XML parser:
  - **parser converts XML into a JavaScript accessible object (XML DOM)**

1.3.6 Creating XML on Servers

- XML can be statically available as a file (e.g. `books.xml`)
- XML can be dynamically created
  - from ASP
  - from PHP
  - ...

1.4 Handling XML using Java

- XML parser: **SAX** (Simple API for XML) is a de-factor standard for XML handling in Java
- SAX is available as part of JavaSDK (`org.xml.sax`)
- for further reference on XML handling in Java (e.g. validating, etc.), see "Java SOA Cookbook" and "Java and XML"

2 Web Browsing

2.1 Browser-Side Scripting

2.1.1 JavaScript

- originated from **ECMAscript**

2.1.2 JQuery

2.1.3 JSON (JavaScript Object Notation)

- lightweight text-based open standard designed for *human-readable* data exchange
- used for representing data structures and associative arrays
- ex. use in AJAX

```
var my_json_obj = {}
var http_req = new XMLHttpRequest();
http_req.open("GET", url, true);
http_req.onreadystatechange = function() {
  if (http_req.readyState == 4 &&
      http_req.status == 200) {
     my_json_obj =
       JSON.parse(http_req.responseText);
  }
}
```

- Inside Java program, there's JSON library which builds JSON object (**Builder** pattern)

```
mystr = new JSONStringer()
          .object()
             .key("JSON")
             .value("Hello, String!")
          .endObject();
          .toString()
```

creates

```
{"JSON", "Hello, String!"}
```

2.1.4 DHTML (Dynamic HTML)

- not a language or standard; it's mix of technologies to make websites more dynamic

2.1.5 AJAX (Asynchronous JavaScript and XML)

- not a language or standard; it's mix of technologies to make websites more dynamic and interactive
- **= fat client!**
- **involved technologies**: XHTML, DOM, JavaScript, CSS, XML

2.1.6 Google Web Toolkit

- corss-compiles Java into JavaScript
- google chrome Web Apps

2.1.7 Google Dart

2.2 Server-Side Scripting

2.2.1 PHP

2.2.2 .NET

2.2.3 ASP

2.2.4 Node.js

2.2.5 Web Services

3 Authorizations and Authentications

3.1 OpenID

- way of identifying yourself no matter which website you visit
- open standard that describes *how users can be authenticated in a decentralized manner*, obviating the need for services to provided their own ad hoc systems and allowing users to consolidate their digital identities

3.2 OAuth

- allows users to share private resources (photos, contact list) stored on one site with another site without having to hand out credentials (id/password)
- e.g. 3rd party tool/website is granted access to my google PPT file even if it does not know my id/password
- **access delegation**

4 Enterprise Computing

4.1 Web Services

4.1.1 Trends

- **Containers**
  - old: EJB
  - new: Spring
- **Persistence Model**
  - old: EJB/JDBC
  - new: iBATIS, JDO, Hibernate
- **Web application framework**
  - old: Struts
  - new: Tapestry, Ruby on Rails
- **Java or PLs**
  - old: Java
  - new: Python, Scala, Ruby

4.1.2 Java-WS (Java API for XML Web Services)

- offers three basic choises for connecting to web services:
  1. **dynamic invocation**
  2. **proxy**
  3. **SAAJ**

4.1.3 WSDL (Web Services Description Language)

- XML-based language for describing web services and how to access them (e.g. how to locate)
- **WSDL ports** (`<portType>`)
  - defines a **web service** (or **operations**) that can be performed and the messages that are involved
  - **operation types**: **one-way, request-response, solicit-response, notification**

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

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

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

- **WSDL messages**
- **WSDL types**
- **WSDL bindings**

4.2 Business Processes

4.2.1 BPEL (Business Process Execution Language)

- origins of BPEL can be traced back to **WSFL** (Web Services Flow Language) and **XLANG**

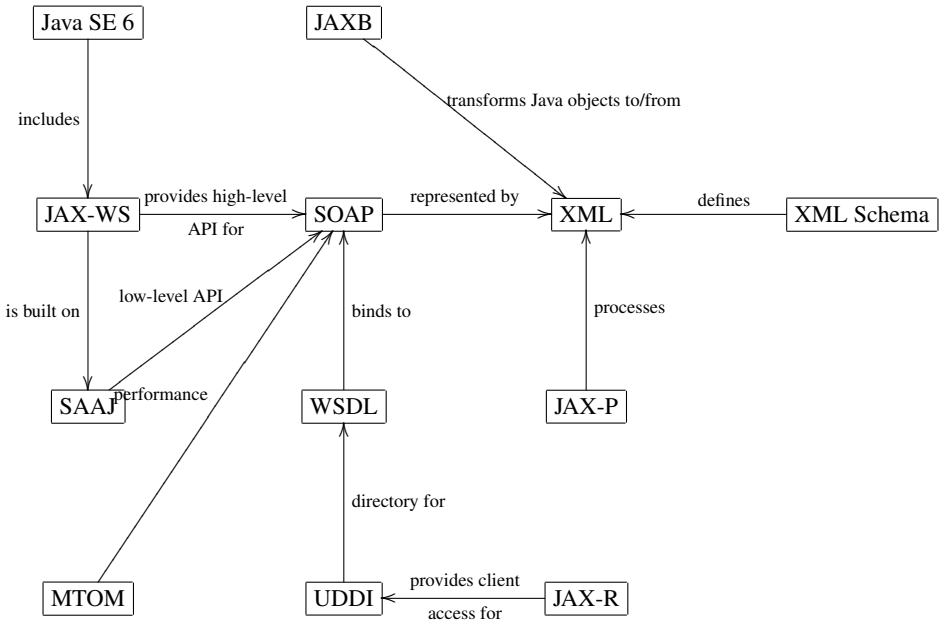

Figure 1: The world of JAX-WS (many of thesea re quite dead)

4.2.2 Workflows

4.3 SOA (Service-Oriented Architecture)

4.3.1 Service

A **service** is a *softwsare component* with the following properties:

- defined by a **interface** that may be **platform-independent**,
- available across a **network**
- carries out *business functions* by operating on *business objects*
- interface and implmenetation can be decroated with extensions that come into effect at runtime

4.3.2 SOA (Service-Oriented Architecture)

An **SOA** is a architecture that

- uses **services** as building blocks,
- facilitates enterprise integration and component reuse through **loose coupling**.

4.3.3 Types of services

- **entity service**: CRUD (create/read/update/delete) opreations over basic entities (e.g. customer, invoice, employee, product)
- **functional service**: performs a function such as sending an email, or logging, or notification, security, etc
- **process service**: represents a series of related tasks; can be represented as an orchestration by an ESB, with a coarse-grained contract that causes the process service to appear as a unified whole to clients

4.3.4 Service composition

A **composed service** is an aggregate of other existing services.

4.4 Enterprise Server-Side Langauges

4.4.1 Java

4.4.2 Python

4.4.3 Ruby

4.5 Enterprise Containers

4.5.1 EJB (Enterprise JavaBeans)

4.5.2 Spring

4.6 Enterprise Persistence

4.6.1 JDBC (Java Database Connectivity)

4.6.2 JDO

4.6.3 Hibernate

4.7 Web Applications Framework

- basically **MVC framework**
  - **Model**: data model + business logic
  - **View**: UI
  - **Controller**: glue (e.g. event system)
- **functionalities of web application frameworks**
  - **web template system**: static HTML + dynamically generated parts

        - e.g. **real estate website**: 500 static pages vs. 1 dynamic page + 500 records
    - **caching**: instead of dynamic generation every time, cache generated files
    - **database access and mapping**
    - **URL mapping**
    - **AJAX**: for interactive web applications (i.e. more intensive UI; more than just buttons)
    - **web services**
- **push-based frameworks**: Struts, Django, Ruby on Rails, Spring MVC
- **pull-based (a.k.a. component-based) frameworks**: Struts2, Lift, Tapestry, JBoss Seam, Wicket, Stripes

4.7.1 Struts

4.7.2 Tapestry

4.7.3 Ruby on Rails

-

5 Software Bus

5.1 D-Bus

5.1.1 Overview

D-Bus is a message bus system, a simple *way for applications to talk to each other*. In addition to the major role of **inter-process communication** and **remote procedure call** capabilities, D-Bus helps coordinate process lifecycle (or **workflow**); it makes it simple and reliable to code a "single instance" application or daemon, and to launch applications and daemons on demand when their services are needed.

D-Bus provides two kinds of daemons: a **system daemon** for events such as "new hardware device added" or "printer queue changed" and a **per-user-login-session daemon** for general IPC needs among user applications.

5.1.2 Language bindings

APIs for D-Bus, or **bindings**, are available in several languages – typically one per language. Each presents its own API as suits the language, hiding the details of working with D-Bus from the programmer to different extgents. The ideal is to fit the D-Bus API into the native language and libraries as naturally as possible.

5.1.3 Buses

There are two major components to D-Bus: a **point-to-point communication dbus library**, which in theory could be used by any two processes in order to exchange messages among themselves and **a dbus daemon**. The daemon runs a actual *bus*, a kind of "street" that messages are transported over, and to which any number of processes may be connected at any given time.

There are two kinds of buses: a single **system bus** for system-wide communication and a **session bus** used by a single user's ongoing GNOME session. A session bus normally carries traffic under only a single user identity, but D-Bus is aware of user identities and does support flexible authentication mechanisms and access controls.

5.1.4 Addresses

Every bus has an **address** describing how to connect to it. A bus address will typically be the filename of a Unix-domain socket such as "`/tmp/.hiddensocket`," but it may also be a TCP port where a bus daemon is listening on an IP-domain socket, or conceivably a descriptor for some other low-level communications scheme.

The dbus library is responsible for hooking up clients to the bus daemon. This process is said that a client process opens and uses a **connection** to the bus.

5.1.5 Connections

Every connection to a bus can be addressed on that bus under one or more names. These names are called the connection's **bus names**. (note that bus names are for "connections" not for "buses.") A bus name example is `com.acme.Foo`.

When a conection is set up, the bus immediately assigns an *immutable* bus name, called a **unique connection name**. One example is "`:34-907`".

5.1.6 Object Model

Message exchange on protocols like TCP or UDP is symmetric.

5.1.7 Objects

One end of any exchange on a bus will always be a communications endpoint that in D-Bus parlance is called an **object**. An object is created by a client process and exists within the context of that client's connection to the bus. The object is a way for the client process to *offer its **services** on the bus* – but one client may create any number of objects.

The bus imposes an object-centric view of communications, where any message carried by the bus is of one of three kinds:

1. **Requests** sent to objects by client processes.
2. **Replies** to requests, going from an object back to a requesting process.
3. **One-way messages** emanating from objects, *broadcast* to any connected clients that have registered an interest in them.

Thus at a higher level of abstraction, the bus supports two forms of communication that we could call **1:1 request-reply** going to an object, and **1:*n* publish-subscribe** coming from an object.

Every bus has at least one object, representing the bus itself. Clients can obtain information about the status of the bus by sending requests to this object. As you'll see later on, it represents the bus in other useful ways as well.

5.1.8 Proxies

Objects on the bus can be accessed through *references* that we call **proxies**. We call them that because a proxy is a local representation inside your own program of an object that is really

	identified by	looks like	chosen by
bus	address	`unix:path=/var/run/dbus/system_bus_socket`	system configuration
connection	bus name	`:34-907` (unique) or `com.mycompany.TextEditor` (well-known)	D-Bus (unique) or the owning program (well-known)
object	path	`/com/mycompany/TextFileManager`	the owning program
interface	interface name	`org.freedesktop.Hal.Manager`	the owning program
member	member name	`ListNames`	the owning program

accessed through the bus, and typically lives outside your program: you literally access the object "by proxy."

Since any *object "lives within" the context of a connection*, it takes a combination of that connection's bus name and the object's own name to find it. Once you have found the object you want, if you'll be using it again soon, you'll usually want to keep a proxy to that object around as a variable in your program. That will save you having to look up the object time and again.

5.1.9 Methods

When a client sends a request to an object, it sees this request as invoking a method on the object: the object is asked to perform a specific, named action. Normally, if a client tries to invoke a method on an object that the object does not provide, this will raise an error.

The method's definition may require certain information to be passed with the request as arguments (input parameters). For every request, a reply message carries the result back to the requester, along with either result data (output parameters) or, if the action could not be performed, exception information. Exceptions will contain at least an exception name and an error message.

Most D-Bus bindings make all this fit in with their environment's native mechanisms, hiding the finicky details of encapsulating parameters in messages and translating exception messages into exceptions (or whatever the native error-handling mechanism is). For example, passing a string argument to a method of some remote object will look to your program just like passing a string argument to a function in your own program. There is no need for tedious conversions and copying of the data into messages, and there is usually no need to concern yourself with the sending of the underlying message. The binding takes care of all that; the work of encapsulating your data into the messages is called **marshalling**.

There is one interesting difference with conventional function calls: when sending a request to an object, you don't necessarily have to sit around and wait for a reply. In more complex programs you'll usually find other useful things to do until the method completes. You may want to be ready to handle user interaction, for example, or availability of data from a file or a network connection; you may even have multiple method invocations "in flight" and want to handle the results as they come in, rather than in some pre-defined order. This style of invocation, where you go on to do other things while waiting for an answer, is called **asynchronous method invocation**.

5.1.10 Signals

The other form of communication is **signals**. These one-way communications come from an object and go nowhere in particular. Client processes can register an interest in signals of a particular name coming from a particular object. Whenever an object emits a signal, all interested clients will receive a copy of the signal. There may be one client receiving it, or there may be many–or nobody may be listening. There are no replies to signals: the object emitting the signal would not know how many replies to expect, or where to expect them from.

Signals can carry parameters, just like method invocations. Signals are used to *publish the occurrence of events that clients may be interested in*, such as the closing of some other client's connection to the bus. That particular kind of signal is sent by the object representing the bus itself. Because of this, the event can be announced properly regardless of whether the departing client closed its connection in an orderly fashion, or was killed, or crashed spectacularly.

5.1.11 Interfaces

So every object supports particular **methods** and may emit particular **signals**. These are known collectively as the **object's members**.

All of an object's members are specified in interfaces, which are *sets of declarations*. "Implementing" an interface amounts to promising to provide all methods specified in the interface, and announcing the availability of its signals for listeners. Each of these members must accept and/or provide parameters exactly as specified in the interface.

5.1.12 Message ordering

Requests from one connection to the same object are delivered in the same order in which they were sent. The same goes for multiple replies from one object to the same client.

This does not mean that all messages are always delivered in sending order. For example, if two client processes send requests to the same object around the same time, there is no documented guarantee that the object will receive them in the same order. Even when one client sends two subsequent requests to the same object, then waits for both replies, it is possible that the reply to the second request comes in before the reply to the first request. The "object" may really be a multithreaded server process with multiple requests being handled in parallel, or it may prioritize requests internally.

5.1.13 Activation

So far we've assumed that objects are created by active clients. There is another way of offering services on the bus: the bus daemon can be instructed to start (or activate) clients automatically when needed. Activation of a client can be triggered in two ways, both keyed by a well-known bus name that the activated client must obtain:

1. Through an explicit request to the object representing the bus itself.

2. By invoking a method on an object in the context of the client's well-known bus name.

The latter can be inhibited through an option in the method invocation message. Some bindings may try to activate an appropriate client when you create a proxy on a well-known bus name that is not currently in use; others may defer this until you use the proxy to invoke the method. The difference can matter if you listen for a signal coming from an object: if the client that should provide the object is not actually running, you could wait in vain!

6 Data and Databases

6.1 Databases and Storage

6.2 Data Warehouses

7 Cloud Computing

7.1 Hadoop

7.1.1 Hadoop core

7.1.2 HDFS: Hadoop file system

7.1.3 MapReduce

7.1.4 ZooKeeper: Distributed coordination service

7.1.5 HBase: Distributed, column-oriented database

7.1.6 Hive: Distributed data warehouse

- provides a query lanauge based on SQL on top of HDFS (i.e. over data stored in HDFS)

7.1.7 Chukwa: Distributed data collection and analysis

7.1.8 Pig: Dataflow language

7.2 OpenStack

-

8 Internet of Things

8.1 Wireless Sensor Networks

8.2 Arduino

8.3 Chips: MCUs vs FPGAs vs CPUs

8.3.1 AVR

8.3.2 ARM

8.4 FitBit

9 Virtualization

9.1 Virtualization

9.2 Hypervisors

10 Case Studies

10.1 Google

10.1.1 GFS

10.1.2 protobuf: Protocol Buffers

- Google's data interchange format
- a way of encoding structured data in an efficient yet extensible format
- Google uses Protocol Buffers for almost all of its internal RPC protocols and file formats

10.2 Facebook

- SDKs:
  - JavaScript + DOM + CSS
  - FLV
  - PHP, XHP
- Infrastructure:
  - Apache Cassandra (Distributed Storage)
  - Apache Hive (Data Warehouse)
  - FlashCache
  - Scribe: data aggregation

11 Design Patterns

11.1 Service Provider Framework

- 3 (or 4) components
  - **service interface**
  - **provider registration API**
  - **service access API**
  - **service provider API (optional)**
- e.g. in JDBC
  - service interface: **Connection**
  - provider registration API: **DriverManager::registerDriver(n)**
  - service access API: **getConnection()**
  - service provider APi: **Driver**

11.2 MVC Framework