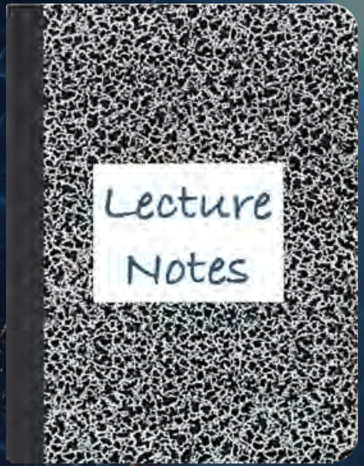
CS 417 – DISTRIBUTED SYSTEMS

Week 3: RPC & Web Services

Part 3: Web Services



Paul Krzyzanowski

Web Limitations

Web browser:

- Dominant model for user interaction on the Internet

But not good for programmatic access to data or manipulating data

- UI is a major component of the content
- Data and presentation are not separated
- *Site scraping* is a pain (and unreliable)!

We needed

- ***Remotely hosted services – that programs can use***
- Machine-to-machine communication

RPC Had Problems on the Internet

Interoperability	• RPC frameworks were often dependent on specific languages, OSes, platforms • RPyC is not even compatible between Python 2 & 3
Transparency	• We try to pretend remote calls are "just like local" • Reality: Have to handle errors and uncertainty
Firewalls	• Most RPC frameworks used random ports (assigned by OS) • Firewalls would block them
State	• Distributed objects require state (object memory) • This makes load balancing and failover difficult
Non-RPC Interactions	• RPC gave us a functional interface • But we also want streaming data, notification of delays, and publish-subscribe

Distributed objects mostly ended up used in intranets of homogenous systems and low latency networks

This led to the idea of web services

Goal:
Create a way to expose functions and data over the web

Benefits of using a web infrastructure and HTTP for communication:

- Authentication mechanisms provided via TLS (https, digital certificates)
- Secure communication via TLS (https)
- Load balancing
- Human-friendly naming (URLs) vs. port numbers
- Firewalls can allow HTTP traffic (and many can allow/block specific URLs)

Two widely-used interaction models for web services: **RPC** and **REST**

Web Services

What is a web service?

Set of protocols by which services can be published, discovered, and used in a technology neutral form

- Language & architecture independent

Service Oriented Architecture (SOA)

SOA = Programming model

- Applications will typically invoke multiple remote services
- An app is the integration of network-accessible services (components)
- Each service has a well-defined interface
- Services are autonomous & loosely coupled

Neither service depends on the other: all are mutually independent

Neither service needs to know about the internal structure of the others

Benefits of SOA

- **Autonomous services**
  - Each service module does one thing well
  - Supports reuse of services across applications
- **Loose coupling**
  - **Isolation**: No need to know the implementation of services, just interfaces
  - **Migration**: Services can be located and relocated on any servers
  - **Scalability**: new services can be added/removed on demand
    … and on different servers – or load balanced
  - **Updates**: Individual services can be replaced without interruption

General Principles of Web Services

Coarse-grained	• Prefer fewer operations with large messages • Amortize the overhead of latency (avoid lots of small interactions)
Platform neutral	• Messages don’t rely on the underlying language, OS, or hardware • Standardized protocols & data formats • Payloads are text (usually XML or JSON)
Message-oriented	• Communicate by exchanging messages • This could be RPC request-response but also streaming reads
HTTP	• Use existing infrastructure: web servers, authentication, encryption, firewalls, load-balancers

XML RPC

XML RPC Goals & Properties

- Data marshaled into XML messages
  - All request and responses are human-readable XML
- Explicit typing
- Transport over HTTP protocol: solves firewall issues
- No IDL compiler support for most languages or robust ecosystem
- Not widely used. Popular deployments:
  - WordPress traditionally used XML-RPC for remote publishing
  - It's the only RPC that ships with Python (`xmlrpc.client`, `xmlrpc.server`)

Sample XML-RPC Python Code

server.py

```
from xmlrpc.server import. \
SimpleXMLRPCServer

def add(a, b):
    return a + b

def hello(name):
    return f"Hello, {name}!"

with SimpleXMLRPCServer(("localhost", 8000),
  allow_none=True) as server:
    server.register_function(add, "add")
    server.register_function(hello, "hello")
    print("XML-RPC server listening on " \
          "http://localhost:8000")
    server.serve_forever()
```

client.py

```
from xmlrpc.client \
import ServerProxy

srv =
ServerProxy("http://localhost:8000",
      allow_none=True)

print("add(2, 3) ->", srv.add(2, 3))
print('hello("students") ->',
            srv.hello("students"))
```

XML-RPC example (from the previous progam)

Request (not showing HTTP headers)

```
<?xml version='1.0'?>
<methodCall>
    <methodName>add</methodName>
        <params>
            <param><value><int>2</int></value></param>
            <param><value><int>3</int></value></param>
        </params>
    </methodCall>
```

Response (not showing HTTP headers)

```
<?xml version='1.0'?>
<methodResponse>
    <params>
        <param><value><int>5</int></value></param>
    </params>
</methodResponse>
```

XML-RPC ⇒ SOAP

- Extended XML-RPC
  - Standardized message structure: *envelope with header + body*
  - Added extensible headers: authentication, correlation IDs, transactions, …
  - Defined fault handling
- Supports different interaction types
  - Request-response (traditional RPC)
  - Request-multiple-response
  - Asynchronous notification
  - Publish-subscribe (through extensions)

WSDL: Web Services Description Language

- Web Services Description Language
  - Analogous to an IDL

- A WSDL document describes a set of services
  - Name, operations, parameters, where to send requests
  - Goal is that organizations would exchange WSDL documents
    - Feed WSDL document to a program to generate software to send and receive SOAP messages
  - WSDL is not meant for human consumption

Decline of SOAP

- Still used but mostly legacy
  - Enterprises liked the formal contracts and big-company support
- Requires extensive support for creating/parsing/routing messages
- Interoperability was inconsistent
- Difficult to understand
- Painfully verbose

REST

REST

REpresentational State Transfer

- The URI identifies the resource and parameters
- Four HTTP commands let you manipulate data (a resource):
  - **POST** (create)
  - **GET** (read)
  - **PUT** (update/replace)
  - **DELETE** (delete)

Create = PUT with new URI or POST to a URI that returns a new URL

CRUD: *Create, Read, Update, Delete*

Update = PUT with existing URI

- And sometimes others:
  - **PATCH** (update/modify) – modify part of a resource (PUT is expected to modify all)
  - **OPTIONS** (query) – determine options associated with a resource
- Message body contains only the data (contents) – usually in JSON

Example

Identify resources via URLs

```
GET /api/users             # List all users
GET /api/users/123         # Get user 123
```

```
{
    "id": 123,
    "name": "Alice Smith",
    "email": "alice@example.com",
    "created_at": "2024-01-15T10:30:00Z"
}
```

```
POST /api/users            # Create a new user
PUT /api/users/123         # Update user 123
DELETE /api/users/123      # Delete user 123
```

REST vs. RPC Design

RPC approach: Define procedures (operations) that act on data

- `getUser(123)`
- `createUser(name, email)`
- `updateUserEmail(123, "new@example.com")`
- `deleteUser(123)`

REST approach: Define resources and use HTTP methods

- `GET /users/123`
- `POST /users`
- `PUT /users/123`
- `DELETE /users/123`

REST Wasn't Perfect

- **HTTP/1.1 overhead** – text based, cookies, extra headers
- **No support for streaming data**
- **No schema & validation** – any JSON will be accepted
- **Text-based data overhead** – JSON parsing
- **Connection overhead** – One request per connection at a time

HTTP Evolution

The HTTP protocol evolved to make interactions more efficient

- **HTTP/1.0** - Closed connection per request
- **HTTP/1.1** - Introduced **keep-alive** – persistent connections
  - Requests are processed sequentially
  - **Head-of-line blocking** is a problem: large content holds up others
- **HTTP/2** – Multiplexes concurrent streams on one connection
  - Binary format for easier parsing
  - **Head-of-line blocking** less of an issue but possible with packet loss vs. being stuck behind a big request
- **QUIC** – HTTP/3 – Similar to HTTP/2
  - Uses UDP to avoid head-of-line blocking

A Move Back to Binary-Encoded RPCs

Web services are widely used

- Great for working across organizations over the Internet

Web services are not the best choice for high-performance, internal, or time-sensitive applications where latency, bandwidth, and processing overhead are concerns.

Go back to the efficiency of traditional RPCs, but

- Have interactions over HTTP/2
- Add interactions beyond procedure calls, like streaming

gRPC

Open-source RPC framework created by Google

Runs over HTTP/2 and provides:

- **Binary protocol**: more efficient to parse and less bandwidth
  - Uses **protocol buffers** for encoding data
- **Multiplexing**: Multiple requests can be sent in parallel over a single TCP connection with one request not blocking another
- **Header compression**: HTTP/2 HPACK removes overhead of HTTP headers
- **Stream prioritization**: More important data can be prioritized
- **Server push**: Servers can send data proactively to the client's cache

gRPC: Key Features

High performance	Uses Protocol Buffers vs. XML or JSON
Strongly typed	Interface definitions, extensible format, versioning
Bidirectional streaming support	Traditional RPC model AND ability to stream requests and/or responses
Deadlines/timeouts	More suitable for real-time systems and handling failures
Multiplexing	Single connection can handle multiple gRPC calls concurrently
Language agnostic	Supports multiple programming languages

There are many other RPC frameworks

- **GraphQL** (API style, not RPC)
  - Primarily designed as a query language for UIs
  - Clients can select the fields they need: you don't always need the full data structure
- **Apache Thrift** (developed at Facebook)
  - IDL + code generation
  - Multiple transports/protocols; mature ecosystems
  - Similar to, but not as widely used as gRPC
  - Used by Facebook, X, Evernote, Microsoft
- **Connect RPC** (Buf project)
  - Protobuf RPC that works cleanly over HTTP/1.1 or HTTP/2
  - Can use Protobuf or JSON encoding; simpler deployment than full gRPC

Common themes

- Shared schemas, deadlines/cancellation, streaming, and strong observability hooks

The End