Painlessly Porting Network Applications to the Browser

Michael Carter

QCon

November 18th, 2009

Web vs. Desktop/Mobile

- Normally discussion centers on GUI capabilities
  - The Web is a great application GUI platform
  - Getting closer and closer to thick/native clients
  - And in some ways its better
- But what about network capabilities?
  - Stuck on HTTP
  - Applications are made to follow a document protocol model
  - Request/Response is the only communication mechanism
- What if the web were not HTTP constrained?

Web Model

- HTTP-Based Protocols
  - Usually JSON/XML based
  - Unspecified “De-facto” per-application protocols
- Session/authentication tied to cookies
- Session spans many separate TCP sessions and many HTTP requests.

Desktop/Mobile Model

- TCP-Based Protocol
  - Mail – IMAP/POP
  - Chat – XMPP/IRC/etc.
  - Collaboration – Obby/Wave/etc.
  - Games – Custom
  - etc.
- Session/Authentication tied to TCP Session

Which is better?

- Web is tied to HTTP, which is about *content*
  - It sure does content really well
  - It would be crazy (infeasible and impossible) to re-implement this stack just for one application
- The Desktop is not protocol constrained
  - More choices: FTP · IMAP · IRC · NNTP · NTP · POP · SMTP · SNMP · SOAP · SSH · Telnet · XMPP · etc.
    - Much More Flexible
    - Easier to choose wrong.

Any Desktop-Only Apps Left?

- Mail? No: Gmail, hotmail, etc.
- Chat? No: Meebo, gmail chat, etc.
- Editors? No: Bespin
- Games? No: Yahoo Games, flash games, etc.
- Real-time Multiplayer Games? Maybe...

- Overall, very few apps are Desktop-only.

More Important Questions

Why are there so few real-time web applications?

Why is it so hard to build a Gmail clone?

Can we salvage any desktop-era engineering knowledge?

Present day approach to Building a Gmail Clone

- The approach is to implement all of the necessary logic within a web framework

- Write a database schema for user accounts, mail archives, settings, etc.
- Choose a web framework/language: Rails, PHP, Django, Servlets, etc.
- Write server-side code to query remote mail servers, and expose that logic over HTTP
- Write code to maintain and publish online presence and chat messages; Spend hundreds of hours learning how to implement Comet.
- Iterate database schema and server-side logic to include threaded conversations, search, contact lists, filters, buddy lists, and more.

Desktop-era approach

- Write no server-side code. Configure out-of-the-box software for all server-side tasks.

- Choose an LDAP server for account/auth information and address book records.
- Choose an IMAP server for receivng, sorting, and saving mail.
- Choose an XMPP server for chat
- Choose an SMTP server for sending mail.

Protocol-Enabling the Web, a TODO List

1. Build a browser compatible socket
2. Expose it to JavaScript
3. ?
4. Profit

Building a Socket

- Flash **XMLSocket**
  - **Blocked** by forward proxies, firewalls, and other intermediaries
  - Not in all browsers
  - Javascript / flash bridge necessary
  - Cross-domain xml access control
- **WebSocket**
  - Supports all intermediaries (**only with encryption enabled**)
  - Not implemented in any browsers yet (check nightlies and patches)
  - Server opt-in acess control
- Comet Session Protocol (**CometSession**)
  - **Supports all browsers, all intermediaries**
  - Operates over http
  - Slightly **less efficient** than flash XMLSocket or WebSocket
  - Standard Server opt-in access control via Host header

Comet Session Protocol

- Specified as a wire protocol
  - Straightforward to implement a server
  - Separates browser hacks from the core spec
- Client generally molds server responses
  - Client determines whats needed to make the Comet Transport work in a particular browser
  - Server has no knowledge of specific browser hacks

CSP Client API

```
var conn = new CometSession()
conn.connect('http://www.example.com/csp')
conn.onread = function(data) {
    console.log('received: ' + data);
}
conn.onconnect = function() {
    conn.send('Hello World');
}
conn.ondisconnect = function(code) {
    console.log('lost connection: ' + code);
}
```

CSP Servers

- Production:
  - Python (pycsp)
  - Server-side javascript (js.io)
- Alpha/Beta
  - Erlang (kohoutek)
  - Ruby (orbited-ruby)
- In progress
  - C (libcsp)
  - C++/IIS
  - Java (servlet-csp)

Additional CSP Info

- **Spec**:
  - http://orbited.org/blog/files/csp.html
- **Mailing List**:
  - http://groups.google.com/group/csp-dev/
- **Latest**:
  - http://orbited.org/svn/csp/trunk

js.io project

- http://js.io
- Library for building Javascript network clients and servers
- File-global Module system
  - jsio('import foo'); console.log('foo is:', foo);
- Swappable transports
  - Write server code once, expose over **CSP**, **TCP**, **WebSocket**, and more.
  - Run the server in the browser for testing with the **postmessage** transport
- **CSP** Server and Client implementation

js.io Echo Server

```
connectionMade = function() {
    logger.log('connection made');
    this.transport.write("Welcome!\r\n");
}

dataReceived = function(data) {
    logger.log('received:', data);
    this.transport.write('ECHO: ' + data);
}

connectionLost = function() {
    logger.log('connection lost');
}
```

js.io echo server boilerplate

```
jsio('from jsio import Class');
jsio('import jsio.logging');
jsio('from jsio.interfaces import Protocol');

var logger = jsio.logging.getLogger('echo');

exports.EchoProtocol = Class(Protocol, function(supr) {
    this.connectionMade = function() {
        logger.log('connection made');
        this.transport.write("Welcome!\r\n");
    }
    this.dataReceived = function(data) {
        logger.log('received:', data);
        this.transport.write('ECHO: ' + data);
    }
    this.connectionLost = function() {
        logger.log('connection lost');
    }
}
```

Running the server

```
require('jsio'); // commonjs import line
jsio('import echo');

var logger = jsio.logging.getLogger('echo server');
var server = jsio.quickServer(echo.EchoProtocol);


jsio.listen(server, “tcp”, { “port”: 5555 })
jsio.listen(server, “csp”, { “port”: 5556 })
```

Running the server in the browser

```
<html>
  <head>
    <script src="jsio/jsio.js"></script>
    <script>

      jsio('import echo');

      var server = jsio.quickServer(echo.EchoProtocol);

      jsio.listen(server, "postmessage");

    </script>
  </head>
</html>
```

World Demo

Case Study: Mino

- Real-time network multi-player game
  - TCP-based protocol
  - iPhone / Objective-C based client
  - Python / Twisted based network server

iPhone Mino Client

Web Mino Client demo

Questions?

Michael Carter
CarterMichael@gmail.com

http://js.io

Porting the Client to js.io

- No changes to the server code whatsoever
- No capability differences in the clients
- Minor decrease in GUI performance
- No network latency performance impact.