Project Report
WebSocket

Otto Allmendinger

10. October 2012

Contents

Chapter 1

Introduction

1.1 WebSocket

WebSocket is a new web technology for full-duplex communication between web browsers and web servers[1]. Normally a web browser fetches data via HTTP GET requests and transmits data to the server via a HTTP POST request, which then can also contain response data. The downside is that each request and response causes considerable overhead[2], since the HTTP protocol has been designed for transmitting web pages and not real-time data. There also is no mechanism for a server to push data to the client without the client initiating an HTTP request.

There are workarounds for these problems, like long-lived GET and POST requests, but these have their own downsides.

WebSocket is comparable to TCP sockets in that it offers a standing connection that allows sending and receiving information without a request-response cycle.

1.1.1 Opportunities and problems

The WebSocket standard is part of a larger trend of the web client becoming more powerful. One possible application of WebSocket in combination with the *DOM Storage API*[3] is the use of client-side data management where the server can push data to the client and the client can send small messages to the server with little overhead.

A common scenario is a client reading some entries from the server, performing a possibly time-consuming task with the read data, and storing a result on the server. The question then arises: what should happen when the input data changed in the meantime?

One way to deal with this is by locking the input values and not allowing other clients to change them until the task is completed.

This has several downsides: keeping locks is expensive and not always necessary. Additionally, there is no way for another client to know if a write lock is actually used or only accidentally granted.

[1] https://developer.mozilla.org/en-US/docs/WebSockets
[2] http://blog.kaazing.com/2010/02/24/5-signs-you-need-html5-web-sockets-part-2/
[3] https://developer.mozilla.org/en-US/docs/DOM/Storage

1.2 Optimistic concurrency with Kung and Robinson

An alternative to locking is the use of transactions as described in Kung and Robinsons paper *On optimistic methods for concurrency control* [1].

The paper presents a simple protocol for applying transactions: each transaction has a *read set* of entries that are expected to not have changed, a set of entries to be written in case the input values haven't changed, and a transaction number *startTn* that indicates when the reads have been made.

When applying the transaction, the server checks if there have been any changes to the *read set* since *startTn* by checking the writes of the transactions between *startTn* and the current transaction number (the number of the least recently applied transaction). If this condition holds true, the changes declared in the *write set* are applied, the transaction number is increased and the transaction is stored internally in order to validate new transactions.

Kung and Robinson also defined the operations *create* and *delete*, which can be omitted in this project because low-level memory management is not a priority on modern systems.

1.2.1 Offline transactions

In *Mobile Computing* by Prof. Dr. Th. Fuchß [2], a method of maintaining a consistent database is described where the client is disconnected from the database for a period of time while still generating transactions. Instead of applying the transaction immediately, the client stores the transaction locally and requests a reintegration of all locally isolated transactions on reconnect.

1.2.2 Examples

Applications of single transactions

startTn	Reads	Writes	Valid	assigned Tn
0	a	b: 1, c: 2	true	1
0		b: 2	true	2
0	a, b	b: 3	false	-
2	a, b	b: 3, a: 1	true	3

Table 1.1: application of single transactions

The third transaction failed because *b* has been modified since *startId* 0. The final data table on the server is *a: 1, b: 3, c: 2*.

Reintegration of a collection of transactions

The next tables describes a scenario where offline isolated transaction are reintegrated into the remote database. Note that the transaction number (#) is only used internally and is not relevant for the reintegration, where a global *startId* is passed alongside the collection of transactions.

#	Reads	Writes
1	a	b: 1
2		c: 2
3	a, b	b: 3
4	c	b: 4

Table 1.2: isolated transactions

The assumed transactions on the remote database

Tn	Writes
4	a: 1, b: 1
5	a: 2
6	b: 5

Table 1.3: remote writes

The reintegration of these transactions starting at $startTn = 4$ checks modifications by other modifications beginning at the transaction with the id *5* (to stay consistent with the use of *startTn* in the single transaction application mode). Only transactions #2 and #4 succeed. The local transactions #1 and #3 fail due to the remote transaction #5. The resulting database entries are *a: 1, b: 4, c: 2*

Chapter 2

Technologies

The following sections will describe the technologies needed to implement a simple web application that allows creation and submission of transactions and proper handling of isolated transactions that are submitted while being disconnected from the server. The database is a simple key-value store on the server side, the client will consist of a simple form to add entries to the *read* set and add pairs to the *write* set.

The client should also display a table of the locally isolated transactions and the transaction application attempts on the remote database. This isn't strictly necessary for production use but helpful in a demonstration project.

The validation and application of transactions is done centrally on a server that accepts *WebSocket* requests. Clients must call the remote methods `applyTransaction` and `reintegrateTransactions` to apply their transactions.

2.1 CoffeeScript

CoffeeScript[1] is a language that compiles to JavaScript. The JavaScript language is widely considered as lacking and partially flawed, which has spawned many other languages that aim to provide languages with better semantics that then compile to JavaScript. Most of the syntax is easily understandable for anyone who has programmed in C, Java or JavaScript, but there are some aspects that should be explained in detail:

Implicit local scope In JavaScript, values are implicitly global if not declared with the `var` statment. CoffeeScript has the inverse approach and treats values as implicitly function-scoped unless explicitly declared as global by attaching them to the global object (e.g. `window` in the browser). This means that the `var` keyword can be dropped.

Shorthand for *this* The `this` operator in JavaScript returns a reference to the current function. CoffeeScript provides a shorthand by allowing to write `@attr` instead of `this.attr`.

[1] http://www.coffeescript.org

For more information on the JavaScript `this` operator, refer to the Mozilla Developer Network[2]

Improved function declaration notation The way to declare a function in JavaScript is to type out the keyword `function`, which gets tiresome quickly. Example in JavaScript

```
var add = function (a, b) { return a + b }
```

CoffeeScript offers the shorthand `->`

```
add = (a, b) -> return a + b
```

The "fat-arrow" notation `=>` preserves the scope of `this`

```
func = ->
    @attr = 1
    x = => @attr
    x() # is 1
```

Implicit return Another improvement is considering the last return value of a statement in a function as the return value of the function. To improve our previous example, it is now possible to write

```
add = (a, b) -> a + b
```

Significant whitespace As indicated in the last example, C-style curly braces are also optional and can be replaced with significant whitespace. Code blocks are then indicated with indentation:

```
foo = ->
    # some code
    bar = ->
        # some other code
        # part of the "bar" scope

    # this is part of the "foo" scope
```

Object notation Another instance of optional curly braces is an improvement on the object notation: JavaScript Objects are hash tables that can be declared by the notation

```
var o = {a: 1, b: 2}
```

CoffeeScript also accepts the forms

```
o = a: 1, b: 2
```

and

[2] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/this

```
o =
    a: 1
    b: 2
```

Nearly everything is an expression Many constructs that are traditionally regarded as statements in other languages can be used as expressions in CoffeeScript. For example

```
max = (a, b) -> if (a > b) then a else b
```

Implicit parentheses The last significant deviation from C-style syntax is the use of implicit braces. Instead of writing

```
result = max(a, b)
```

you can use the lighter notation of

```
result = max a, b
```

2.2 Underscore.js

JavaScript's first-class functions allow for a functional style of programming. Unfortunately, common primitives for functional programming like *map* and *each* are lacking or inconsistently implemented across browsers. The JavaScript library *underscore.js* aims at patching the deficiencies of standard JavaScript and providing a toolkit for functional programming.

Underscore methods are accessible through the underscore identifier: _
Example:

```
result = _.map [1,2,3,4], (x) -> x * 2

# result is [2, 4, 6, 8]

result = _.all [1,2,3,4], (x) -> x < 4

# result is false since (4 < 4) is false
```

A commonly used pattern for creating an object from another object uses the function `_.reduce`. From the documentation

> `_.reduce(list, iterator, memo, [context])`
>
> Boils down a list of values into a single value. Memo is the initial state of the reduction, and each successive step of it should be returned by iterator. The iterator is passed four arguments: the `memo`, then the `value` and `index` (or `key`) of the iteration, and finally a reference to the entire list.

```
sum = _.reduce(
  [1, 2, 3], ((memo, num) -> memo + num), 0
)
# sum = 6
```

To read all values form a database and return a dictionary with `key: value` mappings, `_.reduce` can be used as follows:

```
result = _.reduce(
 readKeys,
 ((obj, key) ->
  obj[key] = database.read key
  obj),
 {}
)
```

We will use this library on the client side as well as on the server. The functional programming style greatly improves the readability of the described algorithms.

2.3 Node.js

The server stack for our project is based on *Node.js*[3], a popular stand-alone implementation of JavaScript based on Chrome's *V8 JavasScript Engine*[4] engine.

Since the client-side API of WebSocket can only be used in JavaScript, this allows the server to be written in the same language as the client and reduces mental context switching when writing web applications. Another feature is sharing code and libraries on the client and the server, which improves code reuse and readability.

JavaScript is designed to run as a single thread, concurrency is achieved by using events. This reduces the amount of locking and thread management on the server and the client and thereby contributes to ease of development and overall code quality.

2.4 Zappa.js

The web application framework *Zappa.js*[5] is a CoffeeScript-friendly adaptation of *express.js*[6], a web application framework on top of *node.js*. A simple web application written in *Zappa.js* can be written as:

```
require('zappajs') ->
  @get '/': ->
    @render index: {layout: no}

  @view index: ->
    doctype 5
    html ->
      head -> title "Hello World!"
      h1 "Hello World!"
```

[3] http://www.nodejs.org
[4] http://code.google.com/p/v8/
[5] http://www.zappajs.org
[6] http://www.expressjs.com

A useful feature of *Zappa.js* is the seamless integration of client-side code that can be written in CoffeScript

```
require('zappajs') ->
  # (... see previous sample)

  # expose script via path "/client.js"
  @client '/client.js': ->
    # greet client with an alert message
    alert "Hello World!"
```

2.5 Socket.io

The JavaScript library *socket.io*[7] provides an abstraction over the raw WebSocket API on the client as well as on the server side. This allows developers to use the same primitives in both environments, greatly improving the usability of WebSocket.

The basic *socket.io* structure in JavaScript on the server looks like this:

```
var io = require('socket.io').listen(80);

io.sockets.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });

  socket.on('my other event', function (data) {
    console.log(data);
  });
});
```

The client can communicate with the server and respond to server-side events via the construct

```
var socket = io.connect('http://localhost');
socket.on('news', function (data) {
  console.log(data);
  socket.emit('my other event', { my: 'data' });
});
```

The communication in *socket.io* is event-oriented. Connecting to the server triggers the `connection` event. The callback function defined in the example uses 'socket.emit' to send the `'news'` event to the client, which logs the transmitted data (`{hello: 'world'}`) to the console and in turn triggers `'my other event'`.

[7] http://www.socket.io

2.5.1 Socket.io integration in Zappa.js

Utilizing the tight integration of client-side JavaScript running on the browser into the server-side JavaScript, Zappa.js offers a tighter integration layer on top of *socket.io*

```
require 'zappajs', ->
  @on message_to_server: ->
    console.log @data
    @emit message_to_client: 'pong'

  @client '/client.js': ->
    @connect()
    @emit message_to_server: 'ping'
    @on message_to_client: -> console.log @data
```

In this example, the client connects to the server and sends the message of the type `message_to_server` containing the value `'ping'` using the method `@emit` (property of the function `@client`).

The server has registered a handler for the message type `message_to_server` using the `@on` method, which in turn emits the message `message_to_client`.

It is also possible to return a response for an emitted message using the `@ack` method, if the client emit method receives a function as the last parameter

```
@on message_to_server: ->
  @ack "pong"

@client '/client.js': ->
  @connect()
  @emit messaget_to_server: "ping", -> console.log @data
```

Here the client will display *pong* in the browser console.

Chapter 3

Implementation

3.1 Database.coffee

Let's have a look at the most important methods of the Database class implementing the algorithms described by Kung and Robinson.

```
class Database
  # ...

  addTransaction: (transaction) ->
    _.each(
      transaction.writes,
      (value, key) => @write key, value
    )
    @transactions[transaction.tn = @tn += 1] = transaction
```

This method applies the changes described by a transaction (contained in the dictionary `transaction.writes`) and adds the transaction to the internal transaction lookup table. The transaction is also assigned a transaction number `transaction.tn`, that is generated by incrementing the internal transaction number counter of the database. The return value of this method is the transaction itself, since the return value of an assignment is the assigned value.

```
class Database
  # ...

  validate: (transaction, base) ->
    transaction.valid = _.all base, (t) ->
      _.isEmpty _.intersection(
        _.keys(t.writes), transaction.reads
      )
```

This method validates a transaction against a list of other transactions, called `base`, and checks if the write set of any of the base methods intersects with the read set of the transaction that is to be verified. This method highlights the power of the *underscore.js* library discussed earlier. Note that the `transaction.valid` flag is being set here.

```
class Database
  # ...

  applyTransaction: (transaction) ->
    baseTransactions =
      @getTransactionRange transaction.startTn + 1, @tn

    if @validate transaction, baseTransactions
      @addTransaction transaction

    transaction
```

The method `applyTransaction` validates a single transaction and adds the transaction to the database if the validation against the relevant transactions is successful. The last statement of the method is the expression `transaction` which simply returns the processed transaction.

```
class Database
  # ...
  reintegrateTransactions: (startTn, transactions) ->
    base = @getTransactionRange startTn + 1, @tn

    _.each transactions, (t) =>
      if @validate(t, base)
        @addTransaction t

    transactions
```

This method is used for reintegrating a list of transactions that have been made by a client that has been offline. It is similar to the method `applyTransaction`, with the difference that it explicitly requires a startTn (the last valid transaction number known to the client) that is independent of the *startTn* value of any of the transactions that are being applied.

This concludes the core of the Kung and Robinson algorithm. What follows is the outline of the integration into the web service

3.2 Server.coffee

The following code only contains the necessary methods for applying single transactions and reintegrating a batch of transactions. This is independent from providing the html interface and the necessary client code, as well as a read-only access to the database. A possible bottleneck in this design is the reintegration of a large number of transactions, which would interfere with read access and the serving of the client interface. One possible solution would be using three separate processes for reading data, serving the client files and applying and reintegrating transactions.

```
require('zappajs') ->
  # initialize empty database
  database = new Database {}

  @on
    # atomic read of a list of keys
    read: ->
      @ack
        tn: database.tn
        values: _.reduce(
          @data,
          ((o, key) ->
            o[key] = database.read(key) or null; o),
          {})

    applyTransaction: ->
      transaction = @data
      database.applyTransaction transaction
      # reply with transaction status
      # and new transaction number
      @ack(
        valid: transaction.valid
        tn: database.tn
      )

    reintegrateTransactions: ->
      {startTn, transactions} = @data
      transactions = database.reintegrateTransactions(
        startTn, transactions
      )

      # reply with dictionary of transaction statuses
      @ack _.reduce(
        transactions,
        ((obj, t) -> obj[t.tn] = t.valid; obj),
        {}
      )
```

3.3 Client.coffee

The client access to the remote database is provided through the class `RemoteDatabase`. Both methods `applyTransaction` and `reintegrateTransactions` accept accept callback functions as the last parameter that will be execute once the server reply (`@ack`) has arrived. If the transaction is invalid (`transaction.valid == false`), the client can decide how to proceed.

```
require('zappajs') ->
  @client '/client.js': ->
    @connect()

    # make function-bound methods available
    # inside other functions
    socket_emit = @emit
    socket_on = @on

    class RemoteDatabase
      # @isolatedTransactions is expected
      # to be of type Backbone.Model
      constructor: (@model, @isolatedTransactions) ->
        @online = true

      disconnect: ->
        @online = false
        # clear the offline transaction buffer
        @isolatedTransactions.reset()

      reconnect: (callback) ->
        @online = true
        socket_emit(
          'reintegrateTransactions',
          startTn: @lastValidTn
          transactions: @isolatedTransactions.map((t) => t.toJSON()),
          -> callback @data
        )

      applyTransaction: (transaction, callback) ->
        if @online
          socket_emit(
            'applyTransaction',
            transaction.toJSON(),
            callback
          )
        else
          @isolatedTransactions.add transaction
          callback()
```

Chapter 4

Conclusion

WebSocket offers new and exciting methods of developing applications and pushes HTML applications in areas that have traditionally been filled by native applications. The possibilities of real-time interaction and concurrent manipulation of datasets by different clients necessitates technologies and frameworks that allow conflict detection, resolution and support for offline operations. This project demonstrates a simple method of achieving these goals in a way that is easy to understand due to the use of modern web technologies that blur the line between server and client code and change the way concurrency is handled when writing networking code.

Bibliography

[1] H.T. Kung and J. Robinson, "On optimistic methods for concurrency control" ACM Transactions on Database Systems, 6(2), June 1981

[2] Prof. Dr. Thomas Fuchß, "Mobile Computing - Grundlagen und Konzepte für mobile Anwendungen", Hanser Verlag, 2009