Designing realtime networking systems with JavaScript

by

David van Leeuwen

Rotterdam University of Applied Science

June 2011
Abstract

JavaScript applications with the use of HTML5 are becoming very powerful. It's allowing developers to use the APIs provided in HTML5 that let an application act more like a desktop application. It's not just that browsers have improved and the JavaScript engines. There are also frameworks that allow you to use better design patterns. With the use of websockets, applications can provide data in realtime. It allows users to chat, play and work together. If you are building such an application with the use of this technique, it forces you to think about a decent architecture.

For example, we have built a prototype called Anatomy Pro Am, which is a Facebook game that allows users to do diagnostics on real cancer patients. The users can chat and draw with up to 5 other Facebook users on CT-scans in realtime. We have made decision how to build this application and built everything in JavaScript with the use of several libraries.

The most important decisions were moving the state to the client with Backbone, so data was available fast. Another important part was using a NodeJS as the server, because it can handle a lot of data and connections simultaneously.

There are several conditions to be considered when building an application:
- If users care about that data to be presented in realtime, make sure that there is as less latency as possible.
- If data doesn't need to be presented in realtime, then don't.
- Make sure your architecture doesn't influence costs for a user.
- This technology is changing fast, make sure you keep in mind that you might want to change parts.
# Contents

1. Introduction 6
   1.1 Acknowledgement 7

2. Realtime networking architecture 8
   2.1 Communication 9
   2.2 State replication 9
   2.3 Channel manager 10
   2.4 Transport 11

3. Building an application 15
   3.1 Development stack 15
      3.1.1 Node.js 16
      3.1.2 Socket.IO 17
      3.1.3 Dnode 17
      3.1.4 BackBone.js 18
      3.1.5 CoffeeScript 19
   3.2 ORM 19
   3.3 Data storage 20
   3.4 Security 21

4. Anatomy Pro-Am game 22
   4.1 Concept 22
4.2 User testing

5. Conclusions

6. Bibliography

7. Appendix

7.1 I shot you first! Gameplay networking in Halo: Reach

7.2 Setting up a Node.js server with Socket.io

7.3 DNode: RPC / event driven networking [updated]

7.4 Clinic visit report
Chapter 1

Introduction

Creating web applications that allow users to interact with one another in realtime has been a challenge since the invention of the Web. Many important kinds of human communication, including shared writing, drawing, and game play are only possible when synchronous realtime interaction is possible.

While creating web tools for asynchronous interactions has been easy since the web's beginning, it has remained difficult to create web-based applications that allow users to play, work, chat, together without time-lags, excessive overhead, or the use of cumbersome, non-portable, and proprietary third-party browser plugins. The need for plugins to support realtime interactions is increasingly problematic, because the use of mobile technologies (such as iPhones) is increasing, and many such technologies do not allow the use of plugins. For the web to grow to support synchronous interactions, an escape from the prison of proprietary plugins is necessary. This thesis describes how we have created a web-based game that is rich with synchronous realtime interactions using emerging open and portable web technologies.

Supporting synchronous realtime interactions has been difficult because, historically, web-based applications were page-centric: A user's browser would download a web-page's content all at once and that content would remain static in the browser. If the user changed some application state, such as by clicking a form submission button, an entirely new web page would be downloaded from the server.

Over time, AJAX (originally short for Asynchronous Javascript and XML) has become common place. AJAX allows elements of web pages to be updated in place through background requests between a browser and a server that update elements of a visible page, rather than replacing the whole thing.

Both the historical method of whole page replacement and the AJAX method of page updating are based upon a request → response model. In this model, the browser opens
a connection to a server with a request, waits until the response is delivered, then times out. Even though AJAX allows web applications to feel more connected to the server, AJAX still requires web browsers to poll the server for new data, instead of letting the server push new data to its browsers.

Developers started to hack this, to create a way to make the server push new data to its clients, which wasn't the standard by any way. These techniques are collectively called Comet. Examples are polling, forever iframes, xhr-multipart and long-polling. With polling the server will respond immediately to a request, with an update or not. It will poll frequently. With long polling, the server waits to respond until there is an update. Rather than responding immediately after a request, it will keep the connection open. This simulates an open connection with the server. These techniques are really helpful and made the web more realtime, but created all sorts of other problems like latency and server overhead because of the amount of request and data that is transfers using HTTP requests.

Realtime networking over the web is not something new. It has been done by using plugins like Adobe Flash and Java. It allowed you to create a socket connection within these applications, but was frequently blocked because of firewall issues. These technologies are not built on top of a open standard and therefore not natively built into the browser of the client.

This thesis tries to emphasize how realtime networking can be implemented for the web. It covers common architectural patterns and will also show you how this can be implemented in an application.

Assume an application requires high-frequency transmissions of rapidly updated data. Under what conditions will realtime networking be useful for an end-user of the web?

1.1 Acknowledgement

While most part of this thesis is about programming in general, it is mostly focused on JavaScript development. Therefore basic knowledge about JavaScript is needed to understand other parts of this thesis.
Chapter 2

Realtime networking architecture

JavaScript is an increasingly central component of the web. It is a practical language with a broad history, but it is also a misunderstood language. While many only know JavaScript as a language where you can validate a web form or even make a image slider, it is much more than that. The language can be used for a number of different tasks, for example server side scripting or (3d) game development. Moreover, it now runs both on web servers and in browsers, allowing code libraries to be reused across contexts.

Realtime data systems can be implemented using a variety of technologies (e.g. C++, Java, Erlang, or JavaScript). This chapter describes general guidelines that can be helpful for creating a realtime application regardless of what tools are being used. Though all examples are written in JavaScript, the design patterns and algorithms used here are implementable in other languages. Similarly, while the examples that are provided derived from a web-based game the approaches shown are applicable for non-gaming and non-web applications.

At the Game Developers Conference 2011 in San Francisco, David Aldrigde, a developer at Bungie, gave a presentation about gameplay networking in Halo: Reach. These guidelines are similar in what he presented and a coverage of his talk can be found in appendix 7.1

Chapter question: what do we need to create an realtime networking architecture?
2.1 Communication

The realtime web is becoming more important. Companies like Google, Facebook and Twitter are starting to offer features in realtime. It is only logical that people are interested in this type of communication. For example, waiting for another person to response to a question that you asked, is kind of tedious. These days it is unacceptable to wait on breaking news, sport results or stock changes. People are trying to find the fastest source, because a delay of a couple of minutes will often cause a bad user experience. Data, search and other types of communication without delays is only natural.

Before starting building an application it is helpful to think about the architecture. A common pattern when building a realtime architecture, is that communication is triggered by events within an application. When the state of the application changes, it triggers an event and sends it out to the server. A state represents the current situation of the application. When an object changes and that object is important in a way, it will call a function that will notify the server that the state has changed. This way, an application becomes more flexible, because it's loosely coupled.

```javascript
object.on('data', function(n) {
    console.log('data: ', n);
});

object.trigger('data', myData);
```

In most cases, with event-driven architectures, there are event emitters and event listeners. In the above example, the first function listens to an object and returns data in a callback, and the second function emits to an object passes the data as a parameter.

2.2 State replication

Although game networking might seem to be different then networking for web. It mostly comes down to the same core concepts. The work of Mark Frohnmayer and Tim Gift [1] give more detail on these concepts. Replication is a big part of that. Their idea of the "Datablock stream manager" provides a way to let the client receive the latest state available, provided by the server. In most cases, this is useful during the initialization of the application. The server provides and guarantees to provide the
latest state to the client, and the client replicates this into its model. According to [1] the client copies it to a ghost object that represents the current state.

Similar ideas can be implemented with JavaScript. While everything in JavaScript exists out of objects, a model can be created and propagated by a call to a server either via websockets or plain old AJAX requests.

### 2.3 Channel manager

In order to propagate representational state, nodes in a distributed system need to be able to communicate with one another. On the web, where direct peer-to-peer communication is difficult, a central server node can orchestrate peer-to-peer communications by forwarding messages from one peer to another. This central node is called a Channel Manager.

Channel Managers should provide a way that there is a client-server communication layer and a client-to-client(s) communication layer. Basically our end goal is to send an object over a 'channel' to an individual object or a group of objects, instead of sending it to all objects. Every object can represent a connected client, which can be end-users of an application or another server. The channel itself is an object that keeps a reference to other objects, but it does not contain data. It will only transmit an object to another, by adding it to a queue.

A design pattern which is commonly used to create such an event-driven solution is called Publish/Subscribe or Pub/Sub. This is a messaging pattern where there are senders (publishers) send messages to specific receivers (subscribers). This creates a more dynamic networking system, by allowing you to let specific users listen to specific messages from specific senders. They don't know about each others existence and are therefore loosely coupled. They can operate normally without being able to know if there is anyone listening to or sending messages. This is really powerful when you create a scalable application with a lot of event-driven features.

To create a Pub/Sub in an application, the only thing that you need is to have an object keep track of an event name and a way to invoke them.
var PubSub = {
    subscribe: function(event, callback) {
        var calls = this._callbacks || (this._callbacks = {});
        (this._callbacks[event] || (this._callbacks[event] = [])).push(callback);
        return this;
    },
    publish: function() {
        var args = Array.prototype.slice.call(arguments, 0);
        var ev = args.shift();

        var list, calls, i, l;
        if (calls = this._callbacks) return this;
        if (list = this._callbacks[ev]) return this;

        for (i = 0, l = list.length; i < l; i++) {
            list[i].apply(this, args);
        }
        return this;
    }
};

PubSub.subscribe("miauw", function() {
    alert("The cat miauws");
});

PubSub.publish("miauw");

The above example is a simple implementation by Alex Maccaw [3], but it can be extended to include features such as maintaining multiple channels or even clients. The next step will be implementing something were this can be shared on the server and on the client.

### 2.4 Transport

Websockets are a part of the HTML5 specification. It provides bi-directional, full-duplex communication over a TCP socket. The API is designed to be implemented by browsers and web servers and can open persistent connections to transfer data. Although websockets are a standard at this moment, it is only implemented in the following browsers:

- Chrome >= 4
- Safari >= 5
• iOS >= 4.2
• Firefox >= 4*
• Opera >= 11*

*However, Firefox and Opera currently disabled their implementation by default for websockets because of certain security reasons¹. But it is something that is currently being sorted out and will be probably be solved within a matter of weeks.

Websockets basically allows a server to send and receive strings from connected clients, without the latency or server overhead that occur when you use Comet. The protocol for the browser is pretty straightforward and is handled by either ws://example.com or wss://example.com (secured):

```javascript
var socket = new WebSocket("ws://example.com");
```

The protocol is for establishing a websocket connection, which is handled by the browser and server is that a client sends a websocket handshake request and the server sends a websocket handshake response. When this is completed the browser en server can send messages back and forth, without the use of any additional HTTP headers for a low overhead.

---

This figure (1) shows the comparison of overhead between polling and websockets.

According to this figure and article, Web Sockets can provide a 500:1, or even 1000:1 overhead reduction. Which is of course a mayor improvement. Using such systems will heavily influence the cost of servers and traffic. Making realtime efficient is an important part of making it useful for everyone.
This figure (2) shows the comparison of the latency between polling and websockets.

As you can see in the figure above, websockets can send messages in a higher frequency. This will cause the latency to drop to a ratio of 3:1, depending on the amount of traffic. This is interesting of you're sending a lot of different types of small messages, where websockets will prove itself.
Chapter 3

Building an application

While trying to create the perfect architecture for a realtime application, it is often better to implement parts and choosing the right tools to do the job. It is not necessary to reinvent the wheel.

In this part of the thesis, we are going to take a look how we can implement some of the work above. We still assume that we are going to create a web-based game built with JavaScript and try not use non-standard technologies. The game will include realtime technology, for features like communication and interaction between users. Everything needs to be stored, so that users can return to there work on another time. But the most important part of all, is keeping all above points in mind. These conditions are not something that people came up with on one day, but is something we found that can be really helpful with any type of realtime application.

Chapter question: How can we implement those conditions with JavaScript?

3.1 Development stack

Creating a realtime application starts with creating a server that serves a websocket connection. There are currently a lot of implementations available in different languages:

- Node.js - node-websocket-server, Socket.IO
- Ruby - EventMachine, Cramp, Sunshowers
- Python - Twisted, Apache module
- PHP - php-websocket
- Java - Jetty
- Google Go - native
Another implementation is one that is developed by Google. It isn't a websocket connection, but a similar solution which trows out all standard HTTP requests and replaces them with SPDY\textsuperscript{2}. SPDY is implemented in Google Chrome and is used when visiting Gmail for example. It is still under development, but is an interesting technology which might be useful in the near future. But we are looking for a cross-browser solution, because we want to let it work with other and/or older browsers too. We are looking for a solution that works with most people, as they are the end-user of the web. A solution with some kind of fallback option would only be fair.

Choosing the right language or toolset is hard to decide. This thesis is about creating a realtime networking system with JavaScript, so the only solution would be Node.js, because it is the only solution where you can write in JavaScript. Fortunately, there are a lot of pro's for choosing Node.js. It's the newest, but also the most fast growing regarding realtime architectures. The other implementations are good too, but are mostly specifically built to do event-based programming on top of the current situation. With Node.js everything is event-based, so it seems that it's meant to do this kind of work.

### 3.1.1 Node.js

"Node's goal is to provide an easy way to build scalable network programs," is the first sentence on nodejs.org. It is built on top of Google's V8, which is an open source JavaScript engine that is also being used in Google Chrome. Node.js allows you to create a server, which is extendable by using modules, in an easy way. Compared to other webservers, it one of the best with many concurrent connections and handling a lot of input and output data. It is designed to be non-blocking, which makes development easier. Non-blocking means that every native method returns something immediately, which allows developers to work fast without thinking about processes that are running in the background. The only disadvantage is that it creates a lot of callback functions, which is known that it can be hard to read when the application grows.

```javascript
// connect is a middleware module which makes it easier to set up a server
var Connect = require('connect');

// setting up the server, we use port 3000 for our app
var server = Connect.createServer();
server.listen(3000);
```


16
So in three lines of code, there is a server running. It does not return anything, because it’s not catching anything.

### 3.1.2 Socket.IO

Socket.IO is a websocket implementation built as a module for Node.js. It provides a simple API to implement a realtime connectivity in every browser with a Node.js server. It support different kinds of transports, like Adobe Flash Socket, long polling and forever iframes. It uses these techniques as a fallback, so it can be used for a variety of platforms, e.g. mobile devices. It is very easy to implement and it will do most of the work for you.

```javascript
var socket = new io.Socket();

socket.on("connect", function() {
  socket.send("w00t, I just connected via websockets");
});

socket.on("disconnect", function() {});
```

The only exception is that everything you send over the socket, has to be a string. When you are working on a simple system, where you just want to share messages, this way of working is fine. If you want to do more, Dnode might be a good solution.

### 3.1.3 Dnode

Dnode is another module written for Node.js, but also available other languages like Perl, Ruby, Java and Python. With Dnode you can create asynchronous object-oriented remote procedure calls. This means that you call a remote function from one server to another over sockets. But it also works in the browser, doing remote calls from the browser to the server. It uses Socket.IO’s implementation to handle sockets in the browser, which makes it a lot easier. Furthermore, it allows any object to be passed as parameters, those will be stringified and parsed internally.

Let’s take an example where we calculate something on the server:

```javascript
var client = DNode(function (client, conn) {
  this.calculate(x) {
    var y = x * 2;
    // the client needs to implement a this.receive(result) method
    client.receive(y);
  }
  // server is the Node.js instance of a server
```
3.1.4 Backbone.js

Typical applications are written with servers, where the logic lives on the server. As we know, AJAX made a difference. The current trend is building your application in JavaScript. Backbone supplies a structure for creating these JavaScript applications. It is similar to Sproutcore and Cappuccino, which are other JavaScript frameworks. The only difference is, that Backbone is more modular, by using other libraries instead for other tasks, e.g. DOM manipulation or utility features that JavaScript doesn't supply by default. It doesn't persist in using certain features, instead you can choose however you want to use the framework. Backbone provides an easy MVC environment making it more usable for developers to create an application. It consist out of Collections, Models, Controllers and Views, similar to other frameworks like Ruby on Rails.

The interesting part is the use events within the collection. Whenever a model within a collection is updated, it automatically updates your view. If you use Jquery for example, you need to grab that element again, and change the element like you want it to be. But doing that for everything, when all the logic is handled in JavaScript, it's going to be difficult task. What Backbone also provides is RESTful persistence by default, where it does a call to your server whenever you change a model. So every collection represents a resource and using an API. Within Backbone you can use CRUD (create, read, update and delete) to work with your the resources. Internally it takes care of all the HTTP calls by using HTTP verbs (GET/PUT/POST/DELETE). So for example:

```
books.create({title: 'Harry Potter'});
// this will make a new 'book' model in the collection and will do a
POST to the server were you pointed your collection at
```

But if you hook it up to a view too, it will use that view and render it. As well for deleting views, changing views and more. So if you change a model in your collection, it will grab that specific view and change it. This can be very useful if you're handling a lot of logic client side in JavaScript.

BackBone is also available as a module for Node.js. By using BackBone on the server, it is possible to use collections and models the same way it is being used on the client. It is convenient if you think of data storage. The models can be shared across the server and client and can be implemented either way.
3.1.5 CoffeeScript

If you have another background then programming with JavaScript, you might think JavaScript is sometimes hard to read. CoffeeScript is a language that compiles into JavaScript. The reason why you should use CoffeeScript is to make it more readable and syntactically right. The language is more familiar for developers that use Ruby on Rails, Python or similar languages.

Instead of using curly braces to surround blocks of code, it uses indentation. You don't need parentheses when invoking a function with parameters. The var keyword isn't used anymore and operators have aliases. These are just some examples of the things that changed by using CoffeeScript. It makes your code more compact and cleaner at the end.

3.2 ORM

With JavaScript applications it means that the browser is responsible for the state and data management. Data is fetched from the server and stored in the memory of the browser. The disadvantage is that a application will load longer. The advantage though, when the page is done loading, everything that is stored in the memory is really fast because the data is available immediately.

ORM is short for Object Relational Mapping. In other languages, this means that your objects are tight in with a database. Whenever you change the attributes of that object for example, it will do the query to the database for you. With Backbone.js it works almost the same. Depending on how you implemented it, it will make calls to your server to make changes to a specific object/model. Normally, you would implement a kind of API on your server, where the client makes it's calls to.

With a realtime architecture, you can decide which models should be updated in realtime. And with a Pub/Sub mechanism, it is possible to decide which users should be notified. It is even possible to override the default Backbone methods, by changing Backbone.sync(). Every method you call on your objects/models (e.g. create, set and destroy), will be changed to send it to your server via websockets.
3.3 Data storage

Depending on what you want to store, you need to find a suitable solution to store data. In most cases, you are fine using a SQL database or something similar. If you create an API to couple it with a client side implementation like in Backbone.js, it would be very easy to use. But when you're infrastructure needs to handle a lot of small pieces of data, a key/value store like Memcached or Redis would suffice.

Let's take a look at an example were we could use DNode to handle all the CRUD commands, which originated from Backbone.js:

```javascript
class MemoryStore
  constructor: () ->
    @data = {}

  create: (model) ->
    if not model.id
      model.id = model.attributes.id = @guid
      @data[model.id] = model
    return model

  update: (model) ->
    @data[model.id] = model
    return model

  find: (model) ->
    if model and model.id
      return @data[model.id]

  findAll: () ->
    return _.values(@data)

  destroy: (model) ->
    delete @data[model.id]
    return model

  S4: () ->
    return (((1+Math.random())*0x10000)|0).toString(16).substring(1)

  guid: () ->
    return (@S4()+$S4()+"-"+$S4()+"-"+$S4()+"-"+$S4()+"-"+$S4()+$S4()$S4())
```

The above example uses CoffeeScript. When we create an instance of this class, you can use the same commands Backbone.js will let you use on Collection. This class would only store in the memory of the server, but it is possible to make it work with Redis or any other database. ID’s are assigned by the server to every model. If a model in Backbone.js doesn't have an ID yet, it knows that it isn't stored yet.
3.4 Security

Security can be a big issue for a lot of organizations, when they are thinking about implementing their own real-time infrastructure. When you don't secure a websocket connection, everybody can use it. The cross-domain-policy doesn't apply, because it's allowed to connect to a different source (via websockets). It can be very dangerous when you implement websockets and eval() incoming messages, without checking the content of that message. It is possible to do cross-site scripting and attack a site this way. This is basic JavaScript knowledge, but still possible to do.

There are a few things that should be taken under consideration when creating a real-time infrastructure. The most basic one is to make sure that you know what type of messages are being sent over the websocket. Check the contents and make sure you parse it right and add logging. This mistake can be made very easily and someone can hijack a websocket without any problems if this is not implemented. Probably the best way to secure a websocket connection, is by providing some kind of unique identifier to authorize a user. If a user is not authorized, he will be not able to send or receive any messages. A common way is to make this work is giving the user a token when he starts a session. The session is the time that a user is currently visiting the application. After the session the token wouldn't be available anymore. Even a more secure way is by adding some kind of login, where users are identifiable. If you use e.g. Facebook it is possible to letting users authorize first with a token and being able to use there Facebook ID for further calls.

It is still possible to do a man-in-the-middle attack, because data isn't encrypted on a normal websocket connection. It is probably not wise to bank account information information over a websocket. Luckily, the same implementation as HTTP implies for websockets. The WSS protocol is a secured websocket via SSL, which means that the data is encrypted.
Chapter 4

Anatomy Pro-Am game

Let's dive into an example were these technologies are being put in practice. The Anatomy Pro-Am game is a game developed by the Morgridge Institute for Research. It is an educational web-based game, which is using the Facebook platform to make a connection with social activities and a user database.

All the code can be found on: https://github.com/davidvanleeuwen/Anatomy-Pro-Am/

4.1 Concept

Why not have game that is fun and useful at the same time. Farmville or Cityville are popular games, which you can play with friends and help each other build the most awesome farm or city. Anatomy Pro-Am is a similar game, where you build an hospital and where you threat patients, like real doctors. But it's not all fictional, you are working with real patients and real cases. Your effort might even help a doctor make a better decision while working on a patient.

The current implementation is focused on doing diagnostic radiology, which means that users have to find cancer tissue in CT scans. In this part of the game, the user can invite fellow doctors (gamers) to examine a scan of a patient to determine where the unhealthy tissue is. These results can help doctors treat the patients better by having most of the work an ideas on the table made by a large amount of people. It’s a certain way to do crowd sourcing with a difficult task and using the right tools to help users work together.
4.2 User testing

The first version has a specific target audience. The first users would be medical students, because it will help them practice more and they know about the human anatomy. This would make it easier to test a prototype without making a part where we try to explain anatomy and how to find cancer tissue.

After doing heavy research how this can be done, by visiting several professionals (appendix 7.4), a cancer clinic and having a game jam with our group and professionals. We developed a prototype which was tested by a group of students, from the University of Wisconsin. Our main goal was to explore how students would communicate and use the game in general. The interesting part was finding out how they would determine together where the bad tissue is and discussing about the result.

The test was setup as a focus group, where the students were challenged to play the game and discuss the results with each other. Two students who were working together on one computer were challenged to talk out loud while they played the game. Everyone was recorded by one camera, and the one computer was recorded and the camera in the computer followed their conversation specifically.

There were a few things we noticed watching the students playing the game. First of all, the students were really focussed and trying their best. The reason for this might have been because they are motivated students. The students were also right in finding the cancer, some of them were a little bit off, but at the end they agreed with each other.

What stood out was that they didn't really realize that they were actually watching each other draw in the game. They did see the drawings that they made and talked with each other about them, but they didn't care about the fact that everything was realtime. It was just a 'normal' experience, were everything worked like it supposed to work, not with any delays.
Chapter 5

Conclusions

Implementing a realtime infrastructure is getting easier and is becoming standardized. As known companies are perceiving to use these techniques to get a richer and better user experiences, it might even be because it can increase revenue. By providing a faster load time, or providing information faster, it will become a better source for users. Studies have shown that implications can cause negative effects:

- If the response time at Amazon increases by 100 ms, it will drop sales by 1% (source: Greg Linden, Amazon).
- When it takes 500 ms longer to load Google, it will cause 20% less searches (source: Marrissa Mayer, Google)
- Around 5 to 9% will use the back button before the Yahoo page is ever loaded if it takes 400 ms longer to load (source: Nicole Sullivan, Yahoo!).

It's an interesting fact that this will cause a bad user experience. With the introduction of web sockets, it is possible to make more dynamic applications without a the server overhead and latency as other techniques. Therefor it will probably be more widely implemented.

To answer the main question of this thesis:

Assume an application requires high-frequency transmissions of rapidly updated data. Under what conditions will realtime networking be useful for an end-user of the web?

The conditions depend on how designers and developers implement the technology and what kind of application they are building. It depends on what users expect and this is different if you application is about communication, or if you are building a game or even something with statistics. But the with the experience we have with building Anatomy Pro Am, it only mattered to users if it has any negative consequences, e.g. if the application is slow. If it's just working 'normal', the user mostly doesn't care if his
message to another user is delayed by seconds. The most important part is that the application itself doesn't block, when the user is using it. So storing data locally and sending it in the background is still a reasonable way to do this. This will currently make it more useful for users, while it is also providing realtime data to other users.

In my opinion this will change over time. We are at a moment were technology is changing fast, were websockets is becoming the next part of the web. It all depends how sites and applications are going to implement this and make design decisions. A game like Anatomy Pro-Am can use websockets intensively, but in other sites or applications it can be a less important part. I believe that over time the web is going to depend on websockets, were user experience is driven by these techniques and is becoming more realtime.

To sum up the most important conditions:
- Make sure you know if users care about 'realtime'
- Determine what data has to appear in realtime
- Using websockets is cost efficient, so make sure there is no overhead and latency, because users are not going to pay or will even not use your application.
- Make your application ready to use these techniques
Chapter 6

Bibliography


Appendix

7.1 I shot you first! Gameplay networking in Halo: Reach

On Thursday, David Aldrigde talked about how Bungie does networking in Halo: Reach. As a multiplayer first person shooter, networking is a really important aspect of the game. ‘Technology to help multiple players sustain the belief that they are playing a fun game’.

During this two hour talk he showed us Bungie’s approach to create this technology into their game. Halo: Reach doesn't use dedicated servers, so networking maybe different in some cases. But in this case it's mostly the same and is maybe even similar in what I’m trying to do with our upcoming Facebook-game.

Networking architecture

The key in a good architecture is to create a networking stack where a part is about prioritization and a part about replication (communication of state/events to a remote peer). There are several ways Bungie handles replication. The state is a guaranteed message of the current state of the user. Furthermore, there are events that are unreliable messages of occurrences. These are not guaranteed, because it are
transitions of gameplay, which only that specific user needs to now. Finally, there is control data, which are high-frequency transmissions of rapidly updated data.

These messages all get a priority. The priority is divided in three keys. The final priority is a value which is a summery of a few properties (e.g. is it a weapon, then it get’s a higher priority). The relevance of the message is the second key in prioritizing (e.g. is the weapon nearby the player). The last key is the desired update period (in ms).

**Design networking**

When you want to design your network, it’s best to create a sequence diagram. A sequence diagram will let you see where (at what time) happens. The most important thing of that is, that you could see where the gabs are, which are essentially lags. David Aldrigde showed us a few examples of early sequence diagrams, which were improved over time because of bugs.

There are 4 rules of gameplay networking:

1. Which parts of your gameplay need to be adjudicated by a single authority?
2. Always ask: where am I hiding the lag?
3. Don’t be afraid to change game mechanics to improve networking
4. Reserve time to iterate

**Optimize networking**

The problem with creating an invisible system with ever-growing complexity, like a networking system, is that it may appear to work, but has a lot of soft failures and inefficiencies.
The first rule of program optimization: don’t do it
- Michael A. Jackson

When your optimizing, it’s easy to find obvious architectural optimization points. With this you gain 1% efficiency and introduce lots of bug. Make sure you use the right tools to inspect where and what resources are being used.

This talk was in my opinion the best talk for me. It was the most relevant one and the most techy one. And I think it's very useful for the work I'm doing now.

7.2 Setting up a Node.js server with Socket.io

Some people find it hard to setup a Node server, so they could start playing around with realtime technologies. You don’t really need Node for that (you could also use services like PusherApp). But if you want to keep control over things, then Node is a good solution. Here’s how it works:

1) **Install Node with Homebrew**

First take a look at nodejs.org, there’s a lot of useful information on the site and it’s well documented. To setup Node you could just get it from the repository or download it directly from the site. But there are more lazy ways to get Node, which I more prefer. So I used Homebrew (use the terminal):

```bash
brew install node
```

2) **Install Node Package Manager**

Even better, there’s a package manager out there for Node. So you’re really going to need this:

```bash
brew install npm
```

[EDIT]: You don’t want to use sudo for this command (because of several reasons, which I’m not going to explain here). What you want to do is something similar like:

```bash
chown -R $USER /usr/local/
```

3) **Installing Socket.IO**

Socket.IO is probably the best solution out there to implement realtime technology. It
has a fallback (if your browser doesn’t support websockets) and it’s nicely integrated in Node and could also be used in your frontend. Here how you install SocketIO, the easy way:

```bash
npm install socket.io
```

4) Setting up the server
I use another module for setting a Node server. You don’t need it, but it makes things easier. So first we need to install `Connect`:

```bash
npm install connect
```

This was actually the hardest part to do, because many things can go wrong. If some did go wrong, please use Google it or just reply to this post. If you don’t know what all of those things are for, just play around with it, read about it and you’ll figure it out.

Open up your text editor, for example TextMate, and open a new file and add the following:

```javascript
// require the modules we need
var Connect = require('connect');
// setting up the server, we use port 3000 for our app
var server = Connect.createServer();
server.listen(3000);
// require another module (but we're pointing to a directory now) and
call a method in that module and pass the server with it as a parameter
require('./socket').bootSOCKET(server)

And save this file as `app.js`. 
5) Setting up the socket server
So we got a server, but it’s pointing to nothing now. So we need to set up the socket server to get things going:

// require the modules we need
var io = require('socket.io');
exports.bootSOCKET = function(app) {
    // let socket.io listen to our server
    var socket = io.listen(app);
    socket.on('connection', function(client) {
        client.on('message', function(message) {
            // sends a message to this client (the client that had sent this message)
            client.send(message);
            // sends a message to all connected users
            client.broadcast(message);
        });
        client.on('disconnect', function() {
            // do something when a user disconnects?
        });
    });
};

Save this file as socket.js in the same directory where app.js is.

In this example you can see it just passes all the messages. But in production mode, that’s probably not a good idea.

6) Final step to create your own realtime app
To start the server, just use the following in terminal (from the directory were app.js is):

node app.js
The server is now up and running. But you probably will need a frontend that communicates with your backend. It’s easy to do with Socket.IO (clientside). You could also check out our game we’re working on. It’s just a technical demo for now, which uses the same stuff: Anatomy Pro Contouring.

Original post: http://blog.davidvanleeuwen.nl/post/3278275938/setting-up-a-node-js-server-with-socket-io
7.3 DNode: RPC / event driven networking [updated]

In my last tutorial I explained to set up some basic things to begin writing your own realtime app. I also mentioned the blogpost from &yet about syncing states within backbonejs. But the most important thing is that I got inspired by David Aldrigde’s talk at GDC about networking in Halo: Reach and another talk about Eve Online’s networking system.

The problem in our previous example is, that you’re always will end up trying to stringify and parse your objects, which doesn’t make sense if you are trying to sync or arranging communication between other players. Furthermore, you’ll probably be using something like a switch or if statement and reading your object to pass it on to the right method, which could cause a lot of confusion if you have to handle a lot of different types of data.

Instead of creating an instance of the Socket.IO library and using it’s send method to send a stringified object, like so:

```javascript
socket.send(JSON.stringify({moveTo: {x: 123, y: 123}}));
```

It’s much easier to create a reference (which is handled by DNode) to the server and do a remote procedure call:

```javascript
remote.moveTo(123, 123);
```
So let's just do a simple tutorial (this is similar to the example code provided by DNode) where you can send a message, do something with that message on the server, and publish that message to all it's clients.

**Tutorial**

1. Dependencies

If you don't have NodeJS and NPM (node package manager), please first check out my [previous tutorial](#). The first step is to install some modules that are needed or helpful:

```
npm install dnode
npm install express
npm install hashish
```

Express is a famous library which let's you setup a simple server (but also let's you do other helpful things). Dnode is the module what it is all about ([read more](#) about DNode).

2. Server (app.js)

```javascript
// dependencies

var express = require('express'),
    DNode = require('dnode'),
    events = require('events').EventEmitter,
    Hash = require('hashish');

// globals

var subs = {};

function publish () {
```
var args = arguments;
Hash(subs).forEach(function (emit) {
    emit.apply(emit, args);
});

// server instance

var app = express.createServer();
app.use(express.static('/public'));
app.get('/', function(req, res){
    res.render();
});
app.listen(3000);

// DNode RPC client

var client = DNode(function (client, conn) {
    this.subscribe = function (emit) {
        subs[conn.id] = emit;
        conn.on('end', function () {

            publish('leave', conn.id);
            delete subs[conn.id];
        });
    }

    this.chat = function(msg) {
        publish('chat', conn.id, msg);
    }
});
It's a lot of code, but talk you through it. First are the dependencies, which are the modules that we're using. The second part are some globals. Subs (= subscribers), is an array which will store all subscribed users (to create a Pub/Sub mechanism). The third part is actually the server, which will serve some static files, but we'll take a look at that later. Then comes the most important part: the DNode RPC client. So basically what you do is define certain methods, which are binded to the client (not the variable, but the DNode instance) and that listens to your app (which is the Socket.IO websocket client that is in DNode library).

I know, it’s pretty exciting, but that's not all you're getting with DNode. The clientside part is even more exciting...!

3. Client (/public)
Create a directory ‘public’ where your app.js is located. Create a index.html file in that directory. I am not going to explain the specifics here, because it's not really hard to implement by yourself (and you can choose whatever library you want). What I'm going to explain is what you need to set up the chat. So create a html page and add the following line in the head:

```html
<script type="text/javascript" src="/dnode.js">
```

You don't need to place the dnode.js file in the directory, because it will be inserted by the server (from your package). The next thing you want to do is connect DNode.

```javascript
var remote;
DNode().connect(function(server) { remote = server; });
```
The next thing you can do is create a input field and when some submits something, you can do a remote procedure call:

```javascript
remote.chat(message);
```

Now your message it sent to the server and it calls the global publish method. In the publish method there is something that’s using the NodeJS EventEmitter. The really cool thing is, that DNode provides a clientside implementation for the EventEmitter (bindAsEventListener is a PrototypeJS thing which is equivalent to JQuery.proxy()). So let’s setup a client side event listener:

```javascript
var em = new EventEmitter;
em.on('chat', function(id, msg) {
    element.insert('<p><strong>' + id + ' says: </strong>' + msg + '</p>'),
}).bindAsEventListener(this));
```

That’s it!

4. Improvements
There are several things we could do to improve the code. For example, maybe you want to put something in where you can define a nickname (= another RPC). A design-issue, which is happening now, is that everything you do clientside is first sent to the server and is sent back to the client. Maybe you want to add it clientside and send it to the server and/or create a callback if it succeeds.

For more information, you could check out the DNode site or see it on github.

[UPDATE] (3/24)
DNode was actually updated last night and a few things changed. The clientside EventEmitter is not included anymore by default (because it's a part of NodeJS itself). To include the EventEmitter you need to do the following:

```javascript
npm install browserify

Then insert this into app.js (at the server instance part):

```javascript
app.use(require('browserify')({
    mount: '/browserify.js',
    main: require('events')
}));
```

In the clientside javascript the var em needs to be updated to:

```javascript
var em = require('events').EventEmitter.prototype;
```

And the head should include:

```html
<script type="text/javascript" src="/browserify.js"></script>
```

Original post: http://blog.davidvanleeuwen.nl/post/4046274825/dnode-rpc-event-driven-networking-updated
7.4 Clinic visit report

We have visited a clinic and watch those people how they did their work. We have met somebody that teaches anatomy to medical students. And now we had our first lesson about treatment planning, thanks to Jennifer B. Smilowitz.

With treatment planning are a few roles that define the treatment that is best for the patient. The team exists out of oncologists, therapists, physicists and dosimetrists. Everyone is responsible for a certain part in the treatment process, but they work together and use the same software to view the cancer. We worked on a case today with Philips Pinnacle, which is a medical imaging tool that allows you to contour and create a beam structure to treat the cancer. Other relevant tools are Varian Eclipse and Osirix (imaging software).

Medical students work with these kind of software tools too. It’s pretty difficult to grasp in just one lesson. Students have to learn to use this software and understand what is displayed on that image, from the anatomy (which is really hard) to creating the treatment (searching for abnormalities).
During our lesson we worked on a case with a patient that had prostate cancer (anatomized patient). We did some contouring (coloring in certain regions), defined some CT numbers (to determine the dose), the isocentric setup (which is used to determine the position of the patient) and setting up the beams. Although prostate cancer is the easy to treat (comparing to e.g. head and neck cancers), it is very interesting to see. With prostate cancer, it is acceptable that other parts in that area will be radiated, that’s why your whole prostate would always be radiated (also to prevent spreading).

Original post: http://blog.davidvanleeuwen.nl/post/3892436746/we-have-visited-a-clinic-and-watch-those-people