Gossip on top of Web Sockets:
Adding a dash of Total Order

Project Distributed Systems - 2011-2012

Professor: Prof. dr. M. Aiello

Student: Samuel Esposito - s1597183

Introduction

This project is about highly scalable peer-to-peer data replication among web pages of a single site running in different browsers (clients) on different machines on the web. The data is generated by users visiting the site and has to be synchronized to all other users visiting that same site at a specific point in time. The number of clients the data has to be replicated to varies over time as users can open and close webpages any time, and also can temporarily loose the internet connection on their mobile devices. A use case for this problem statement could be a web based multiplayer game for which each player's state has to be synchronized to all other players. Another example is an online source control system in which developers can concurrently write and edit source code.

In order to solve the posed replication problem the gossip protocol was implemented on top of HTML5 web sockets. Because the gossip protocol doesn't guarantee consistency for concurrent writes (there is no total order), the protocol was overlaid with a scalable algorithm that enforces total order for individual objects only. Also rebasing was used to allow offline processes to efficiently synchronize their local state with the processes in the network when they come back online.

Problem Statement

The problem addressed in this project is about replicating data generated by visitors of a specific site to all visitors of this site at a specific point in time (or in the future) in a peer-to-peer manner. Popular web sites typically have huge amounts of visitors, and the amount of visitors can change very rapidly over time. Therefore the algorithm used for replication must be very scalable. Also the visitors of a website form an open group. Users can open and close web pages any time. This implies that the replication algorithm has to tolerate visitors appearing on and disappearing from the peer-to-peer network and continue its operation transparently. With the advent of the HTML5 features of local storage and offline web applications, visitors can even use web pages offline. Visitors that generated data offline should be able to synchronize their data to all other visitors when they come back online. Finally because any visitor can write data to the 'shared state' of the site, write conflicts can occur and have to be dealt with appropriately.

Solution Design

In this section we go over the techniques we used to build the peer-to-peer network between all visitors of a site and the algorithms applied for the replication of their data. The actual implementation of these techniques and algorithms is detailed in the next section.

The whole concept of having site visitors send data to each other in a peer-to-peer manner is based on web sockets, or more specifically multicast over web sockets. A web socket is an HTML5 feature allowing bi-directional full-duplex communication between the browser and the web server over one single TCP-socket. As such it is an easy to implement replacement for the former, hard to set up comet solutions.
Because web sockets only take care of the communication between client and server, we need to build something on top of that to let the clients communicate with each other. For this we use multicast. Clients can subscribe and unsubscribe to a channel on the server and every message sent to this channel will eventually be delivered to all subscribed clients.
It is possible to provide a full membership service by having the server to keep a list of all subscribed clients and pass it to all clients that subscribe, by monitoring the clients for failures and by notifying clients about membership changes, so that every client knows at all times the members of the peer-to-peer network. But as we will see later on, this membership service is not necessary for the gossip protocol or any other used algorithm to work, as long as every client has a unique sortable id. This saves us overhead on the network and improves the performance of the replication system when the network changes rapidly.
Validity, integrity and agreement in multicast is attained as follows. Every process keeps a sequence number for the messages it sends and piggybacks it on the outgoing messages[1]. Processes receiving messages with sequence numbers they already received, discard those, so that each message is delivered only once. Processes receiving messages with sequence numbers higher that what they expect, push the messages in a holdback queue and request the missing messages using a negative acknowledgement (nack). This way every message sent will eventually be delivered and this will happen in the order the sender sent them. Every message contains a checksum, so that random errors, such as a bit flip in a message, can be detected.

With all this in place it is possible for the clients to reliably pass messages to each other in a peer-to-peer manner, either by using multicast through the server channel or by using point-to-point communication through a destination filter on top of the multicast channel. This is all we need to implement the gossip protocol.
The gossip protocol is a highly scalable solution for data replication in open groups. It uses vector timestamps (which are an extension of the sequence numbers mentioned before) to guarantee causal consistency of message delivery. Because all replica managers (RMs) in the system are equal (there is no master-slave architecture), it can easily tolerate failures of any RM in the network. This aspect makes it especially fit for networks of clients that can join or leave at any moment in time.

Thanks to the causal consistency enforced by the gossip protocol, we know that all messages will be delivered in causal order on all clients in the network. That is, when a message is sent after a specific set of messages are received on the sending client, this

---

[1] In fact a time vector is used, as we will see later on, but this has the same effect for validity as a sequence number (causal consistency implies first-in first-out for the messages sent by one process).

message will be delivered on every other client in the network after this set of messages was delivered. Unfortunately causal consistency doesn't guarantee that concurrent messages are delivered in the same order on all clients. This doesn't matter for some types of data like for instance messages in a chat channel or on a forum. But for other types of data like edits on a source file or the state of a multiplayer game, one can get conflicts when two concurrent writes on the same object are applied in different order on different clients. The only way to fix this is to add some kind of total ordering to the gossip protocol.
Traditional total ordering ruins the scalability and failure tolerance of the gossip protocol. Either all clients in the network have to agree on the order of the messages before they can be delivered or there has to be a master client that decides on the order of messages (cfr. Lamport bakery algorithm). Both options cause considerable overhead because they require a lot of extra messages to be sent (negotiation or heartbeat). When one of the nodes or the master node is temporarily offline, this can cause delays or require a new master to be elected, which causes overhead again.

Fortunately traditional total ordering is not necessary in our case, because concurrent writes on different objects can be tolerated. We only need total ordering for two writes on the same object. To enforce this total ordering for objects, we use revision numbers. Whenever a write message is compiled for sending on a client, the object's data is piggybacked with the latests local revision number for this data and a newly generated revision number. When this message is about to be delivered on another client in the network, this client checks the old revision number with the revision number is locally has for this object's data. When the revision number matches, there were no concurrent writes in the meantime and the message can be delivered. When the revisions do not match, a conflict has been detected: there have been two concurrent writes on the object which have been applied in different order on the sender and the receiver of the message.
In order to resolve this conflict in a scalable way, we have to make sure that all clients reach a consensus on which of the conflicting messages' sender is the winner and which is the loser without having them sending messages to each other. For this we can simply use the unique sortable id of each client in the network and define that the lowest id is always the winner[2]. The losing message is simply discarded and a message is send to the loser to inform it about the conflict, so it can try resending its message as a patch to the winning message (the losing message is rebased on the winning message) or a more recent message.

As we mentioned before a client can still generate data when it is offline. When it does so, it has no idea of the data generated by clients participating in the network, and as an effect its own data will most likely conflict with the network data. To prevent lot's of conflicts from occurring when an offline client rejoins the network, its locally generated data is rebased with the network data before it is sent out to the network. This happens in two steps. First the old network data the client contains from before it went offline (if any) is updated so that it reflects the current state of the network. After that for each data object the first locally generated message that edits this object is rebased on the object's network state. Because all next local messages are based on these initial messages, all local messages can now be sent to the network without having to fear an overload of conflicts.

---

[2] This might not be fair, but it definitely is scalable!

Overview of Realization

In this section we go into more detail about the actual implementation of our peer-to-peer network and the data replication that is going on in there.

The demo implementation that comes with this project consists of two components: a web page that runs in the client and a web socket server that operates as a router for all multicast messages. The web page is part of a Ruby on Rails application (rubyonrails.org), is written in Haml (http://haml-lang.com/) for the HTML part and Coffeescript (http://jashkenas.github.com/coffee-script/) and jQuery (http://jquery.com/) for the javascript and is styled using SASS (http://sass-lang.com/) for css. The web socket server was built in Ruby using EM-WebSocket (https://github.com/igrigorik/em-websocket), which is a web socket implementation running on EventMachine (https://github.com/eventmachine/eventmachine), a fast and simple event-processing library for Ruby.

Below we will discuss the implementation of the data replication middleware step by step, from web sockets and multicast over the gossip implementation to conflict resolution and rebasing local data.

Web Sockets and Multicast
The web socket API provides three self documenting callbacks on the client and server that lets one define the web socket behavior: onopen, onmessage and onclose. In addition it provides the send method for sending messages as strings. This is all we need to implement multicast.
First we discuss the web socket server implementation. Whenever a client connects to the web socket server, the onopen callback is triggered. In this callback the server subscribes the client to the multicast channel and defines that whenever a message is pushed into this channel, it should be sent to the client unless it is a point-to-point message and the client does not match its destination. It also saves the subscription id for unsubscribing the client later on should it disconnect. The implementation of the onopen callback is listed below (vanilla Ruby):

```
# a client connects to the web socket server
ws.onopen {
  # subscribe the client to the multicast channel (save the subscription for later)
  sid = @channel.subscribe do |msg|
    # check destination filter if point-to-point message
    unless msg["dest"] and msg["dest"] != @index[ws.signature]
      # send multicast message or point-to-point message to client
      ws.send JSON.dump(msg)
    end
  end
  ...
}
```

When a client closes the connection with the web socket server, the client is unsubscribed from the multicast channel. The onclose implementation is listed below (vanilla Ruby):

```
# a client disconnects from the web-socket server
ws.onclose {
  # unsubscribe it from the channel using the saved id
  @channel.unsubscribe sid
}
```

Finally when a message is sent to the web socket server this, message is inflated to a JSON object and pushed into the multicast channel. Besides that, the unique id of the sender is coupled to the web socket connection signature to use it later on in the destination filter for point-to-point communication. The onmessage callback is implemented as follows (vanilla Ruby):

```
# a message was sent to the web socket server
ws.onmessage {|msg|
  # parse the message to a JSON object
  object = (JSON.parse msg rescue nil)
  if object and object["id"]
    # save the client's unique id for point-to-point communication
    unless signature = @index.keys.select{|sign| @index[sign] == object["id"]}.first and
           signature == ws.signature
      @index[ws.signature] = object["id"]
      @index.delete signature if signature
    end
    # multicast the message to all clients in the network
    @channel.push object
  end
}
```

This is all it takes to implement multicast (and point-to-point communication) on the server side. Now lets take a look at the client's implementation. The client starts with opening the connection to the web socket server. That is, if it has support for web sockets, so lets check that first using Modernizr (http://www.modernizr.com/) (coffeescript):

```
$ -> # jQuery document.ready
  # check whether web sockets and local storage are available in this browser
  if Modernizr.websockets and Modernizr.localstorage
    # cross-browser compatibility
    Socket = if "MozWebSocket" of window then MozWebSocket else WebSocket
    # open a web socket connection
    web_socket = new Socket("ws://your.domain:8080/")
```

The clients behavior on opening a connection can be defined in the onopen callback. In this callback the client will send a message to the server containing it's unique id, so that the server knows how to handle point-to-point messages destined at this client. In fact it piggybacks this message with a time-vector and multicasts it to all members of the group.

This time vector is used for the gossip protocol discussed later on. The onopen callback implementation is listed below (coffeescript):

```
# upon opening a connection with the web socket server
web_socket.onopen = ->
  # multicast a message with the clients time-vector
  window.web_socket.send JSON.stringify
    id: localStorage._id
    vector: JSON.parse localStorage._vector
    subscribe: true
```

When the connection is closed by any other cause than the client deliberately going offline, it tries to reestablish the connection with increasing wait intervals. The implementation of the onclose callback is shown below (coffeescript):

```
# when the web-socket connection is closed
web_socket.onclose = ->
  unless window.offline?
    debug "reconnecting in \#{window.reconnect_on} seconds...", "Gainsboro"
    setTimeout ->
      document.init_socket()
    , window.reconnect_on * 1000
    # increase the timeout fibonacciwise in order to decrease the network load
    tmp = window.reconnect_on
    window.reconnect_on = window.reconnect_on + window.prev_reconnect_on
    window.prev_reconnect_on = tmp
```

On the reception of a message from the web socket server, it is inflated to a JSON object first. Depending on the message's type, its content is consumed by one of the message handlers defined (coffeescript):

```
# upon recieving a message from the web socket server
web_socket.onmessage = (evt) ->
  data = JSON.parse evt.data
  ...
  else if data.nack?                              # handle multicast nack
    document.send_retry data
  else if data.retry?                             # consume multicast retry
    document.consume_retry data
  else if data.conflict?                          # consume a conflict message
    document.resolve_conflict data
  else if data.id != id                           # consume data message
    document.handle_data data
```

This is all we need for multicast over web sockets. Below we discuss how we implement the Gossip protocol.

The Gossip Protocol

Before we dive into gossip, an explanation of the data model we used in the demo application for this project is in order. All the data that is subject to replication is stored in the browser's local storage object[3]. Local storage is the HTML5 way to store your web application's data and persist it even when the computer is shut down. The localStorage API is very intuitive, since it behaves as any other javascript object:

```
# save a variable
localStorage["variable_name"] = some_value # or
localStorage.variable_name = some_value
# read a variable
my_value = localStorage["variable_name"] # or
my_value = localStorage.variable_name
```

The only difference is that only strings can be stored. In order to store objects, they can be serialized using the JSON.stringify() and JSON.parse() functions (coffeescript):

```
# save an object
localStorage.my_object = JSON.stringify
  key1: "value1"
  key2: "value2"
  key3: [1..5]
# read an object
my_object = JSON.parse localStorage.my_object
```

Every data element in our model contains a revision number and every data update contains the revision number of the data it overwrites and the revision number of the data it writes. In the demo application the revision numbers take the form of 6 character hex strings, so that they can easily be visualized by the corresponding color and possible conflicts 'jump out' in the demo application's interface. For simplicity only the revision numbers (old and new) are communicated over the network while replicating data.

A consequence of using the local storage for data persistence is that in our demo gossip application there will be no difference between the client requesting or writing some data, the front end handling all data request from the client and the replica managers actually storing the data in the database. They all run in the same web page on the same machine (see figure 1). In our project we focus only on the replica manager part of this web page.

---

[3] The same goes for the state of the gossip protocol.

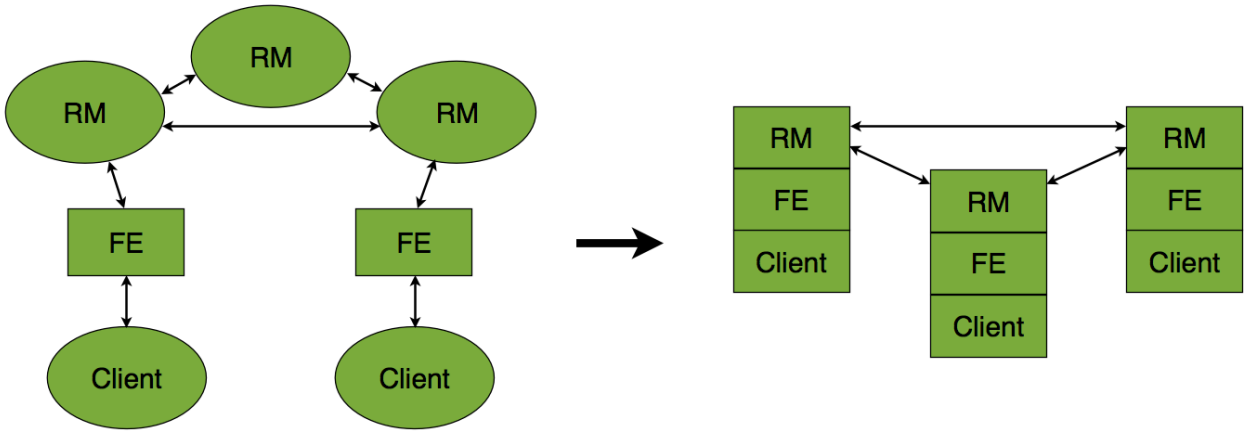


Figure 1: Traditional gossip vs. gossip for web application local storage.

Now over to the gossip protocol itself. The protocol enforces causal consistency by piggybacking all sent messages with a vector timestamp. This vector contains the timestamps of all latest messages that were delivered for each individual client in the network the moment the message was generated. Whenever a client receives a message, the client can use the piggybacked vector to check for messages already delivered by the sender at the time of sending the message, but not by the receiving client. This is implemented in the function below (coffeescript):

```
# Detect missing messages
@missing_id_for = (data) ->
  vector = JSON.parse localStorage._vector       # local time vector
  high_ends = JSON.parse localStorage._high_ends # highest missing message timestamps
  # missing messages of the sender itself
  missing_id = data.id if vector[data.id] + 1 < data.vector[data.id] and
                          (high_ends[data.id] >= data.vector[data.id] - 1 or
                           (high_ends[data.id] = data.vector[data.id] - 1)?)
  # missing messages of other clients
  missing_ids = (id for id of data.vector when id != data.id and
                                               vector[id] < data.vector[id] and
                                               (high_ends[id] >= data.vector[id] or
                                                (high_ends[id] = data.vector[id])?))
  missing_id ?= missing_ids[0]
  localStorage._high_ends = JSON.stringify high_ends
  missing_id
```

When a message or a set of messages is missing, the received message has to be put in a holdback queue for later delivery. This holdback queue is implemented as a priority queue for efficiency: the message with the least missing messages (if any) is always in front of the queue. This priority is obtained by the sorting function below (coffeescript):

```
# sort data by time vector
@by_vector = (first, last) ->
```

```
  do (first, last) -> # nesting to keep variables within the scope
    # initialize values for all client ids, should they not be present
    first_keys = (count for count of first.vector).sort()
    last_keys = (count for count of last.vector).sort()
    first.vector[key] ?= -1 for key in last_keys
    last.vector[key] ?= -1 for key in first_keys
    bigger = []
    smaller = []
    # count the number of positive and negative differences
    for key in first_keys
      bigger.push key if first.vector[key] > last.vector[key]
      smaller.push key if first.vector[key] < last.vector[key]
    # return the order
    return 1 if bigger.length > 0 and smaller.empty()
    return -1 if bigger.empty() and smaller.length > 0
    return 0
```

Enqueueing a message is implemented as follows (coffeescript):

```
# put a message in the holdback queue until it can be delivered
@enqueue = (data) ->
  holdback_queue = JSON.parse localStorage._holdback_queue
  holdback_queue.push data
  holdback_queue.sort @by_vector
  localStorage._holdback_queue = JSON.stringify holdback_queue
```

After enqueueing the message, all missing messages have to be requested using a negative acknowledgement (nack). The sending of the nack is implemented in the code below[4] (coffeescript):

```
# send nack for missing messages
@handle_missing = (id, start, end, data_id) ->
  window.web_socket.send JSON.stringify
    id: localStorage._id  # unique sender id
    dest: id              # message destination
    start: start          # timestamp of first missing message
    end: end              # timestamp of last missing message
    nack: true            # message type
```

When a client receives a nack for messages it sent before, it will look up these messages in a sending queue containing all messages it sent in the past (or the x most recent

[4] In the actual implementation the code checks whether a nack has just been sent for the missing messages and waits for them if so. This reduces the overhead on the network.

messages it sent) and compile a message for resending all data at once[5]. This is implemented in the code below (coffeescript):

```
# resend messages requested by nack
@send_retry = (data) ->
  # load all sent messages
  send_queue = JSON.parse localStorage._send_queue
  {start, end} = data
  # resend the requested messages
  window.web_socket.send JSON.stringify
    id: localStorage._id            # unique sender id
    dest: data.id                   # message destination
    retry: send_queue[start..end]   # missing messages
```

When a message with the requested missing messages arrives at the client, all messages are enqueued in the holdback queue. Then all messages in the queue are delivered. It can happen that a message in the holdback queue is still missing other messages. In that case the nack process restarts from the beginning. The handling of resent messages is implemented below:

```
# consume resent messages
@consume_retry = (data) ->
  # put messages on the holdback queue, if any
  for data_part in data.retry
    @enqueue data_part if data_part
  # process queued values
  process_result = true
  while process_result
    # pop message from the holdback queue
    holdback_queue = JSON.parse localStorage._holdback_queue
    break if holdback_queue.empty()
    data_part = holdback_queue.shift()
    localStorage._holdback_queue = JSON.stringify holdback_queue
    # try to process the data
    process_result = @handle_data data_part
```

Thanks to the piggybacked time vectors, all clients in the network will eventually find out that they miss some messages, as long as at least one client is sending out messages[6]. In addition, when a new client joins the network, it multicasts its own time vector. All other clients compare this time vector to their own and resend messages the new client is missing.

---

[5] By compiling all missing messages into one message, some sending overhead is saved. If the total amount of data becomes very big however, chunking it into separate messages of the right size might be an option.

[6] If no client sends out messages, this means there is no data to replicate.

Total Order for Individual Objects

The gossip protocol enforces causal order with respect to message delivery. It can happen though that two messages are generated concurrently and get applied in different order on different clients. This causes no problems when the messages affect different data objects. When they do however affect the same object, the consistency of a client's data can get compromised[7].

The first thing we have to do in order to mitigate this problem is detect these concurrent writes on a single data object. From now on we call these concurrent writes conflicts. As mentioned before we piggyback the data writes with two revision numbers: one for the data version the write has to be applied to and one for the version of the data after the write has been applied. This is illustrated by the listing below, where value contains the new revision number and old_value the old revision number[8] (coffeescript):

```
data =
  id: localStorage._id             # sender id
  key: key                         # data object id
  old_value = old_value            # old revision number
  value: value                     # new revision number
  crc: crc32("\#{key}|\#{value}")  # checksum
window.web_socket.send JSON.stringify data
```

This gives us enough information to detect conflicts. That is, conflicts occur when a message is generated for a specific revision of a data object on de sending client, but are to be applied on a data object with a different revision number on the receiving client. Because causal order is applied, we know this different revision number must be the result of a concurrent message[9]. The implementation of detecting conflicts is lined out below (coffeescript):

```
{key, value, old_value} = data      # extract variables
local_old_value = localStorage[key] # get local data revision number
if old_value != local_old_value     # compare to revision number message has to be applied to
  @handle_conflict data             # handle the conflict if any
```

Once a conflict has been detected, it can be resolved. A first step in this resolution is defining the winner and the loser of the two conflicting messages. This is done on each client by comparing the unique ids of the senders of two conflicting messages. The lowest

---

[7] Imagine for instance messages applying patches to source files. A patch meant for a specific revision of a source file possibly cannot be applied any more when the file changes in the meantime.

[8] Note that for simplicity only the revision numbers are passed through in our implementation, and no actual data is sent.

[9] Would the message causing the different revision number be contained in the other message's time vector, its data write would have been based on this revision number. And would the message causing the different revision number contain the other message in its time vector, it would have been pushed into the holdback queue until the other message was delivered. Therefore we can conclude that neither of the messages were contained in each others time vector and we detected two concurrent messages causing a write on the same object.

id is always the winner. Although this might not always be fair, the advantage of this approach is that all clients can reach consensus about the winner and loser without the use of message passing for negotiation, which saves us precious network load. The losing message is reverted if it was already delivered or simply discarded in the other case. Then a conflict message containing the conflict details is compiled and sent to the losing client. This is implemented by the code below (coffeescript):

```
conflict =
  id: localStorage._id
  dest: loser                                  # losing client
  loser_time: loser_vector[loser]              # message count of losing client
  winner: winner                               # winning client
  winner_time: winner_vector[winner]           # message count of winning client
  object: data.key                             # object subject to the conflict
  conflict: true
window.web_socket.send JSON.stringify conflict
```

Once a client receives a conflict message from another client[10], it will try to rebase the conflicting message on the new state of an object. That is, it will try to apply the changes intended by the original message onto the object's new state. This state can be the state of the winning message or even a more recent state[11]. Once the message has been rebased, it can be resent to the broadcast channel. Resolving a conflict is implemented below (coffeescript):

```
# fetch conflicting message's data and mark it as resolved
send_queue = JSON.parse localStorage._send_queue
conflicting_message = send_queue[index_of_conflict]
send_queue[index_of_conflict].resolved = true
localStorage._send_queue = JSON.stringify send_queue
# check whether current object state is up to date
vector = JSON.parse localStorage._vector
if vector[data.winner] <= data.winner_time
  # resend conflicting data update as a patch to the current storage state
  @construe_data(conflicting_message.key, conflicting_message.value, true)
```

The newly construed message can now be delivered by each client as any other message in the network.

---

[10] Note that clients can also detect conflicts caused by their own messages and resolve those before other clients detect them. This variant is considerably more efficient, because it does not make use of message passing.

[11] Should the winning message not yet be applied in the losing client when rebasing, the conflict message has to be queued and the winning message can be required using a nack.

Rebasing Offline Data
When a client compiles messages while no network connection is available, these messages are by definition concurrent to all messages compiled by other clients in the network. The chance that these messages will result in conflicts when they are broadcasted as the offline client comes back online is present. Because in such a situation all offline messages are broadcasted all at once, the overhead of resolving the resulting conflicts can cause problems to the network throughput and the processing power of the individual clients. Fortunately there is an easy fix to this problem. When an offline client comes back online, it keeps the offline generated data in a queue and first updates the state of its data with the peer-to-peer network by broadcasting its time vector (see the implementation of the web socket onopen callback). When a client in the network receives such a vector, it resends all missing messages to the client coming back online. This client can deliver these message as it did with other missing messages (see the @consume_retry implementation). Once the clients state is once more in sync with the network, the offline messages can be dealt with. To prevent conflicts with the data in the network, they are preventively rebased on the current storage state. This is implemented in the routine below (coffeescript):

```
@consume_offline_queue = ->
  offline_queue = JSON.parse localStorage._offline_queue  # fetch offline data
  localStorage._offline_queue = JSON.stringify []         # clear the queue
  for data_part in offline_queue
    @construe_data(data_part.key, data_part.value)        # rebase offline data and broadcast
```

Deployment

Here we given a brief overview of all steps to take in order to get the demo source code delivered with this project up and running.

To kick off with you have to fetch the source code from its online repository on github. For this you need Git. Download it from http://git-scm.com/download and install it following the instructions there or install it using your favorite package manager. Once Git is set up, the source code can be cloned using this command (terminal, console or command prompt if you will):

```
git clone git://github.com/krikis/replica.git replica
```

Since the project is written in Ruby and Ruby-based languages, to run the code you must first get Ruby up and running on your machine. Ruby can be downloaded from http://www.ruby-lang.org/en/downloads/ and installed following the instructions there. You can also install it by using your favorite package manager. When Ruby is installed the next thing to do is install RubyGems, the package manager for Ruby programs and libraries. Download it from http://rubygems.org/pages/download and install it following the instructions there.

Now our environment is all set up. The only thing we still have to do is install all ruby libraries that our demo application depends on. Because we use the bundler gem for managing these dependencies, we start with installing bundler (terminal):

```
gem install bundler
```

With our dependency manager in place we can have it install our dependencies for us[12] (terminal):

```
[sudo] bundle install
```

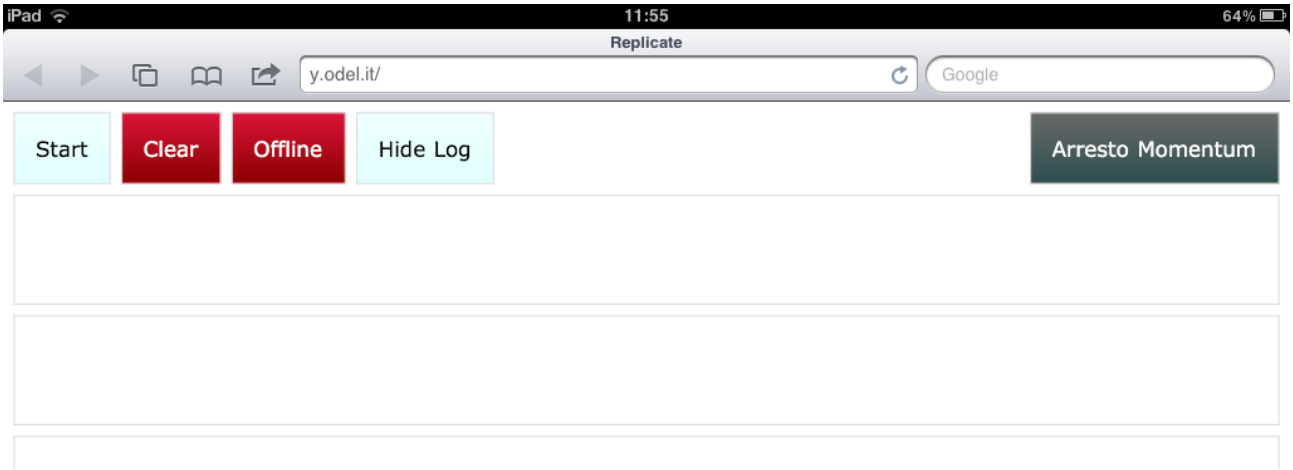
We're all set, so lets fire up our web server! (terminal)

```
cd /path/to/your/repository/
rails s                  # start up the web server
```

Note that we now started a Webrick development server that runs on port 3000 by default. When running on OSX Pow (http://pow.cx/) is an interesting alternative. In production apache (http://httpd.apache.org/download.cgi) can be used in combination with the passenger module (http://www.modrails.com/install.html), among numerous other options of course.
And lets not forget about our web socket server (open another terminal session):

```
cd /path/to/your/repository/
script/multicast    # fire up the web socket server
```

If you now open the application on http://localhost:3000/ or your own DNS, you should see this page pop up:



The controls in the interface will be explained in the 'Results' section. In the console where the web socket server is running, you should see something like this:

```
Server started
```

[12] For convenience this command also installs the dependencies for our web socket server.

```
+1
{"id"=>"a7ec773b815d7896b3edb47f9353df17", "vector"=>{}, "subscribe"=>true}
```

Yay, our web page is subscribed to the multicast channel. We're all ready for gossiping now!

Evaluation

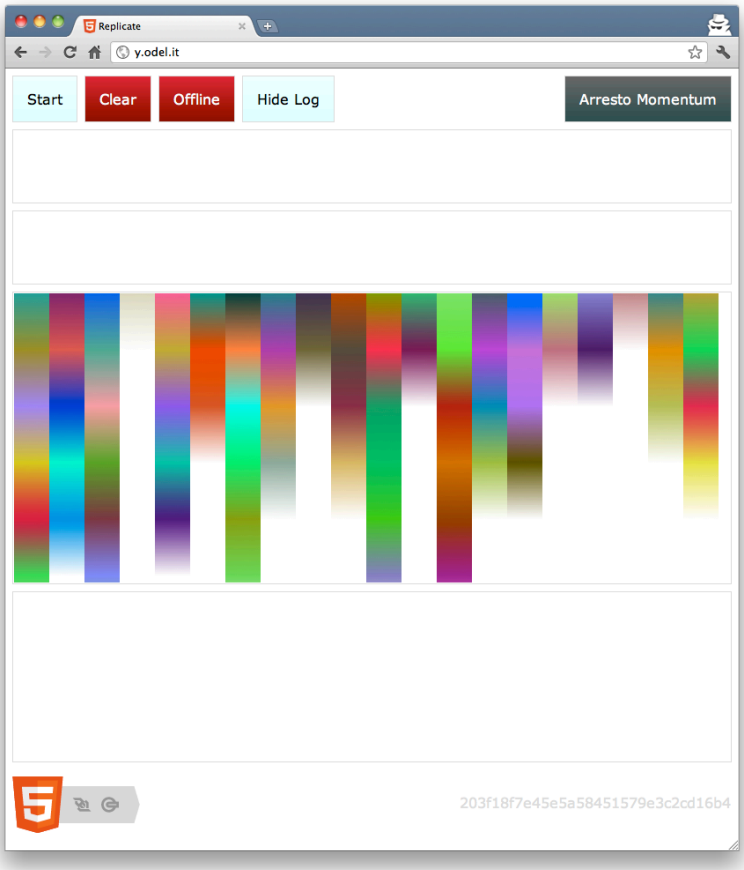
In this project we successfully implemented the gossip protocol on top of web sockets. By doing so we provide a scalable middleware for data replication that can be used by a variety of web applications, ranging from chat groups to multiplayer games.
Our multicast implementation however has one drawback: all messages go through one central server, the web socket server, which can become a bottleneck at high load. Because of that the network might not be as scalable as a 'real' peer-to-peer network, in which the clients talk directly to each other.
Fortunately there is a solution for this problem in the works named PeerConnection (http://www.whatwg.org/). With this API it will be possible to directly connect from one browser window to another. It was already built for chromium by the Ericsson Labs and is expected to be released soon. Support in other browsers is bound to follow.

Future Work

There is one interesting improvement that could be made to our gossip implementation. The web socket server could take a state snapshot of the peer-to-peer network on a regular basis. This snapshot could then be used to quickly initialize new clients with data when they join the network or for updating offline clients with the latest network data when they come back online. This would take considerable load off the network in the case of such events. In addition it would decrease the odds of clients waiting for messages from another client that went offline. Finally it would assure that most of the generated data is persisted, even when all clients leave the network.

Results

In this section we discuss some results obtained from experimenting with the demo application. But first we explain some of the controls in its interface.
At the top there are initially five buttons. The ‘Start’ button can be used to generate random data messages and multicast it over the network gossipwise. Once this button is clicked, an extra control appears allowing to switch between three different speeds, 1 message per second, two messages per second and 20 messages per second. By clicking ‘Stop’ the generation of data messages is stopped. When one clicks the ‘Clear’ button, all previously sent data and all queues, indexes and caches are cleared on all of the clients currently connected to the peer-to-peer network. When ‘Offline’ is clicked the connection of the client with the web socket is closed and the clients does not make part of the peer-to-peer network any more. By clicking ‘Reconnect’ the client joins the network once more. There is a button to hide the log and finally there is a big button on the right that allows for stopping the generation of data messages on all clients in the network at once.



Below this row of buttons, there are four empty boxes. The first one is the offline queue, where messages are stashed away while the page is offline. The one below that is the holdback queue, used for enforcing causal order of messages. The third and largest box contains a visualization of all subsequent messages sent for each data object. There are twenty data objects defined in the demo application and their messages are displayed from left to right. Each message is represented by a gradient of which the bottom color codes for the revision number the message is applied to and the top color stands for the resulting revision number[13]. The lowest box contains a log of all significant events,

[13] This way of visualizing data messages was chosen because it highlights conflicts in the data. When two messages conflict, their respective revision numbers will not match and therefore their start and end colors will not match, which is easy to see in the interface as there will be a ‘jump’ in color (see third experiment).

including conflict detection and resolution, synchronizing with the network state and handling offline data.

The first thing we will experiment with is a client synchronizing its state with the network when it joins the network. To simulate this, we take one client offline, let the other clients all generate some messages and then bring the client back online.

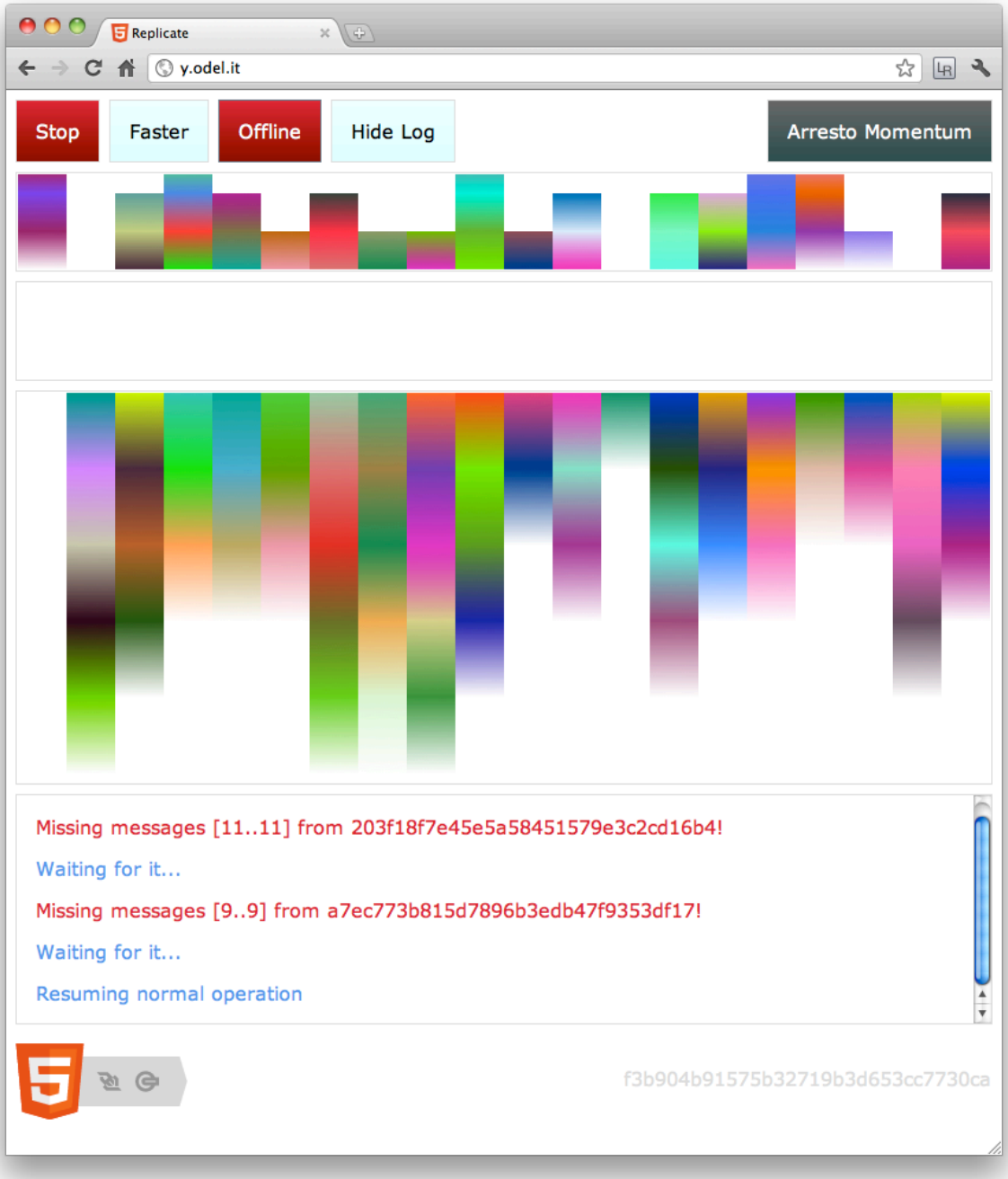
```
+10
{"id"=>"f3b904b91575b32719b3d653cc7730ca", "vector"=>{}, "subscribe"=>true}
{"id"=>"f3b904b91575b32719b3d653cc7730ca", "vector"=>{}, "subscribe"=>true}
{"id"=>"f3b904b91575b32719b3d653cc7730ca", "vector"=>{}, "subscribe"=>true}
{"id"=>"f3b904b91575b32719b3d653cc7730ca", "vector"=>{}, "subscribe"=>true}
{"id"=>"a6b41a34a11715148e18b3ab51d957d1", "dest"=>"f3b904b91575b32719b3d653cc7730ca",
"retry"=>[{"id"=>"a6b41a34a11715148e18b3ab51d957d1", ..., "vector"=>
{"203f18f7e45e5a58451579e3c2cd16b4"=>0, "a6b41a34a11715148e18b3ab51d957d1"=>0}},
{"id"=>"a6b41a34a11715148e18b3ab51d957d1", ..., "vector"=>
{"203f18f7e45e5a58451579e3c2cd16b4"=>1, "a6b41a34a11715148e18b3ab51d957d1"=>1,
"a7ec773b815d7896b3edb47f9353df17"=>0}}]}
{"id"=>"203f18f7e45e5a58451579e3c2cd16b4", "dest"=>"f3b904b91575b32719b3d653cc7730ca",
"retry"=>[{"id"=>"203f18f7e45e5a58451579e3c2cd16b4", ..., "vector"=>
{"203f18f7e45e5a58451579e3c2cd16b4"=>0}}, {"id"=>"203f18f7e45e5a58451579e3c2cd16b4", ...,
"vector"=>{"203f18f7e45e5a58451579e3c2cd16b4"=>1, "a6b41a34a11715148e18b3ab51d957d1"=>0,
"a7ec773b815d7896b3edb47f9353df17"=>0}}]}
{"id"=>"a7ec773b815d7896b3edb47f9353df17", "dest"=>"f3b904b91575b32719b3d653cc7730ca",
"retry"=>[{"id"=>"a7ec773b815d7896b3edb47f9353df17", ..., "vector"=>
{"203f18f7e45e5a58451579e3c2cd16b4"=>0, "a6b41a34a11715148e18b3ab51d957d1"=>0,
"a7ec773b815d7896b3edb47f9353df17"=>0}}]}
```

As one can see in the web socket server output listed above, the client coming back online first multicasts it's time vector. When clients in the network receive this vector, they resent all messages the new client is missing. In the client output listed below we can see that upon receipt of the first resent messages the new client detects that it is missing some messages from other clients that have to be applied first[14]. As it knows that these will arrive soon, since the client just multicasted its time vector, it decides to wait for these messages. When it receives another set of resent messages, it again detects it is missing messages and waits for it. Finally, upon receipt of the third set of resent messages, it can apply all messages in their correct order en resume its normal operation. The client now is in sync with the network.

```
Missing messages [0..0] from 203f18f7e45e5a58451579e3c2cd16b4!
Waiting for it...
Missing messages [0..0] from a7ec773b815d7896b3edb47f9353df17!
Waiting for it...
Resuming normal operation
```

[14] That this happens is no surprise if one closely examines the time vectors of the resent messages in the output of the web socket server. The resent messages of the different clients are clearly interleaved.

Now we will repeat this experiment but we will allow the client to generate data messages while being offline. As one can see in the image below the data is stashed in the offline queue.



And from the client output listed below one can derive that the client first synchronizes its state with the network and then sends out the messages in the offline queue.

```
Missing messages [11..11] from 203f18f7e45e5a58451579e3c2cd16b4!
Waiting for it...
Missing messages [9..9] from a7ec773b815d7896b3edb47f9353df17!
Waiting for it...
Resuming normal operation
Consuming offline queue...
```

Finally we will demonstrate the detection and resolution of conflicting messages for the same object. We can see in the log of the client that it resolves one conflict it detected itself and reports some conflicts to another client in the network. The conflicting messages are indicated with red borders in the visualization and the resolutions are indicated with a blue border. As one can see, the colors of the conflicting messages do not match the colors of their predecessors, while the resolutions of these conflicts do.