An Architecture Pattern for Network Resilient Web Applications

Julien Richard-Foy
IRISA
Université de Rennes, France
julien.richard-foy@irisa.fr

Olivier Barais
IRISA
Université de Rennes, France
olivier.barais@irisa.fr

Jean-Marc Jézéquel
INRIA
Université de Rennes, France
jezequel@inria.fr

ABSTRACT

Network resilience is a cross-cutting concern in the development of Web applications. This paper presents a synchronization algorithm and an architecture pattern isolating the network resilience concern.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

The Web is a convenient application platform: no installation or deployment step on clients, collaborative work, support of heterogeneous clients, *etc.* However, Web applications work only as long as the network connection and the server are up. If the network connection or the server is down, Web applications may suffer from inconsistency, freeze or latency, resulting in a bad user experience. With the advent of Internet connected mobile devices (smartphones and tablets) rises the need for a good support of an off-line mode for Web applications.

In this paper, we use the term *resilience* to refer to the ability of a Web application to deliver a good Quality of Experience (QoE) under bad network conditions. The resilience level can vary. For instance, when an application detects a network failure, it can either:

- Display an error message informing the user that its last action failed (*e.g.* Trello[1]);
- Fallback to a read-only mode (*e.g.* Google Drive[2]);
- Store the actions performed by the user and try to synchronize them to the server when the connection is back (*e.g.* Workflowy[3]).

Achieving resilience can be more or less hard according to the application's domain. In all cases, resilience is a cross-cutting concern: any action performed by the user may trigger an AJAX call that may fail. To keep delivering a good QoE when the server is unreachable, the application must resite the business logic on the client-side and then synchronize the changes to the server when the connection comes back.

In the particular case of collaborative applications, the synchronization step can be challenging: if two clients concurrently change the same resource, how can the server decide which one wins? The more the number of concurrent actions clients can perform without synchronizing to the server, the higher is the risk to be unable to order them.

Our paper presents an architecture pattern based on event sourcing to achieve resilience in Web applications. Our pattern is agnostic to the whole application architecture, it requires little adaptation for existing applications to use it, it only captures and isolates the client-server synchronization process and allows to deliver a good QoE when the network is down. To resolve synchronization conflicts, our architecture can be mixed with either operational transformation or conflict-free replicated data types approaches.

We implemented libraries for both the server and client sides, intended to be used along with the architecture pattern, and used it to develop several use case applications demonstrating the isolation of the resilience concern. We proved the eventual consistency of our synchronization algorithm.

2. RELATED WORKS

2.1 Offline Support

2.2 Collaborative Work

[1]http://www.trello.com
[2]http://drive.google.com
[3]http://www.workflowy.com

Operational Transformation

Conflict-free Replicated Data Types

2.3 Persistent Resources

3. CONTRIBUTION

3.1 Architecture

Our architecture relies on the idea of event sourcing: the state of the application is determined by the application of a succession of events. For instance, in the case of a text editor, such events could be text insertion and text deletion.

Figure 3.1 gives a high-level view of our architecture pattern. The application code is partitioned between the client and server sides, with a synchronization module in between.

In order to achieve resilience, when a user triggers an action, the corresponding command is executed on client-side, so it works even if the network is down. The command is also put in a queue, which is periodically synced with the server. When the server acknowledges the application of a command, the client removes it from its queue.

3.2 Synchronization

Figure 3.2 shows a detailed view of the synchronization module.

Basically, resilience is achieved by executing on client-side the commands logic: users do not have to wait for the server response to have feedback on their actions.

Nevertheless, in the case of a collaborative application, the commands also need to be sent to the server, which is responsible of holding the application state. Clients maintain a queue of commands in which commands are stored while the connection is down. When the connection is back, the queue is sent to the server.

Another parameter to take into account is concurrency: isolated clients can perform actions concurrently. When their commands are sent to the server, what is this one supposed to do if the commands conflict whith each other?

We propose an algorithm to achieve synchronization between clients and server. Web architectures are centralized distributed systems, making it simpler to achieve synchronization than in a completely decentralized system.

Our algorithm uses a logical clock on the server only: each command applied by the server acts as a tick. The clock is used to detect precedence between commands.

The goal of the synchronization algorithm is to provide eventual consistency between all clients.

Each time a client performs an action, the synchronization component assigns it a unique identifier (1), invokes the corresponding logic (2), defined by the application, and appends it into a queue (3). If the network connection is up, the synchronization also tries to send the command to the server along with the timestamp of the last known command applied by the server (4).

When the server receives a synchronization message from a client, consisting of a timestamp of the last known command applied and a command to apply it first checks if the client missed some events and sends them (5). Then it determines, using the timestamp, the commands that have been applied without being notified to the client, and transforms the commands given in the message according to these missed commands so the effect of their application is intention preserving (6). The server then applies the transformed commands by invoking the logic defined by the application (7, 8), appends them to an event log and associates them a timestamp (9). Finally, the produced timestamped events are broadcasted to all clients (10).

When a client receives a notification message (11), consisting of a timestamp and an event, it checks if the event corresponds to a command in its queue (12). If that is the case it just removes the corresponding command from the queue, appends the event to a log and sets the timestamp of the last known command applied by the server to the event timestamp. Otherwise (if the event corresponds to a command initiated by another client), the command logic is invoked on the client after having been transformed according to the commands that have been locally applied but not sent to the server (13, 14), the event is append to a log and the timestamp of the last known command applied by the server is updated.

4. VALIDATION

4.1 Reusable Library

4.2 Synchronization Algorithm Eventual Consistency

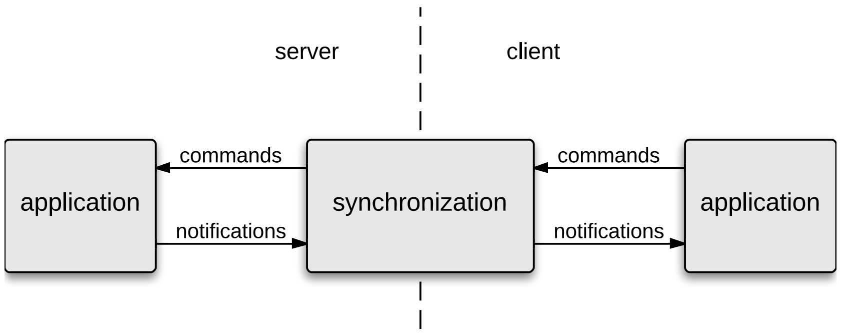
Our synchronization algorithm must be eventually consistent: the state of all the clients must converge to the same value.

We define the following variables and functions:

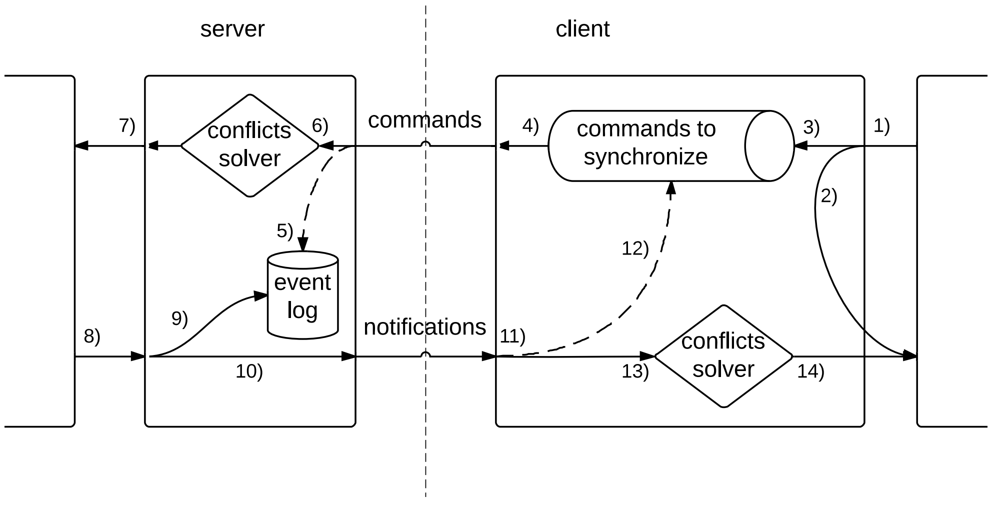
- S_n, the state of the application on the n^{th} client, and S_s, the state of the application on the server ;
- Q_n, the events queue on the n^{th} client ;
- t, a logical clock value stored on the server, starting at 0 ;
- l_i, the last known clock value by client n^{th} ;
- $run(S, e)$, a function computing the state resulting of the application of an event e on a given initial state S ;
- $converge(e1, e2)$, a function such that $run(run(s, e1), e2) = run(run(s, e2), converge(e1, e2))$. This function makes events application commutative.

The implementation of the *run* and *converge* functions depends on the application domain, hence they are provided by users.

```
function $handleAction(action)$:
  $e \leftarrow (action, l_i)$
  $S_i \leftarrow run(S_i, e)$
  $append(Q_i, e)$
```



server
client
application
commands
notifications
synchronization
commands
notifications
application



server
client
7)
conflicts solver
6)
commands
4)
commands to synchronize
3)
1)
2)
5)
event log
12)
8)
9)
notifications
11)
conflicts solver
10)
13)
14)

```
function $synchronize()$:
  $sendToServer(Q_i)$

function $onMessage(e, l)$:
  if $l < t$:
    for $e'$ in $eventsFrom(l)$:
      $sendToClient(e')$
      $e \leftarrow converge(e, e')$
  $S_s \leftarrow run(S_s)$
  $t \leftarrow t + 1$
  $append(log, (e, t))$
  broadcastToClients((e, t))

function $onMessage(e, t)$:
  if $e \in Q_i$:
    $remove(Q_i, e)$
  else:
    for $e'$ in $Q_i$:
      $e \leftarrow converge(e, e')$
    $S_i \leftarrow run(S_i, e)$
  -- update log
  if $t > l_i$:
    $l_i \leftarrow t$
```

The pseudo code handling commands emission on client i is given in listing **??**

Eventual consistency. Our algorithm is eventually consistent if and only if, the state of the application on all the clients and the server converge. All sites (clients and server) start with the same initial state S. When a client performs an action, its state is locally modified to S' according to the effect of the action on the current state. If the network connection is up, the action is sent to the server, which

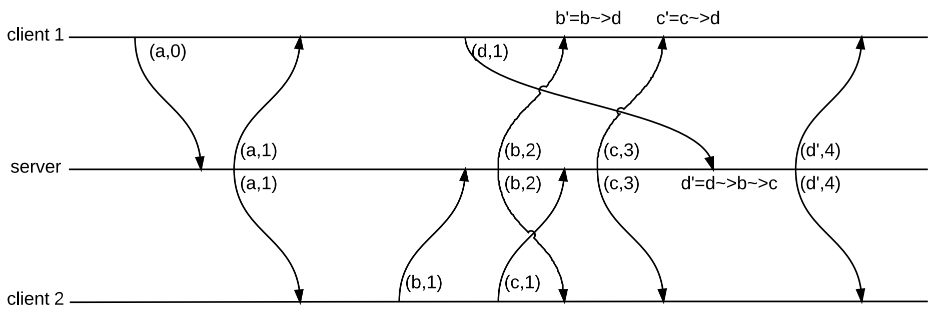
4.3 Limitations

4.3.1 Data Loading

4.3.2 Notifications Personalization

4.3.3 Snapshots

5. DISCUSSION


client 1
server
client 2
(a,0)
(a,1)
(a,1)
(d,1)
b'=b~>d
c'=c~>d
(b,2)
(b,2)
(c,3)
(c,3)
d'=d~>b~>c
(d',4)
(d',4)
(b,1)
(c,1)