Who is the next server? Enabling fault-tolerant local Webapps

Arthur Marques Felix Grund Paul Cernek
University of British Columbia
Vancouver, BC

1 INTRODUCTION

The Internet as a platform of communication has developed a one-and-only character in the previous decade: no matter where devices and users are located, they connect to the Internet and talk to each other through it. Even in scenarios where these devices are close to each other on the same network, local architecture is rarely leveraged and all traffic has to go through the global network, possibly traversing multiple autonomous systems throughout the globe. Driven in particular by the growing domain of the Internet of things (IoT) where different "smart" local devices require network communication, a movement towards making more use of local network infrastructure seems just as reasonable. Nonetheless, adding and configuring these devices in a local-area network may be error prone, time consuming, and non-inviting to the public that purchase them. On top of that, applications run by these devices may require unnecessary Internet connection in order to communicate with their peers.

Imagine a classroom with students following a presentation which is concluded with a vote from students' smart phones. If students are connected to the same WiFi, ideally one student's phone could detect their peers phones in the network and open a local app that would manage the voting system. However, such voting systems are usually performed using online voting platforms where all voters have to connect to the Internet for accessing the platform. Despite the obvious unnecessary traffic, this scenario has another significant problem: the voting cannot be performed upon failing Internet access. This seems almost abstruse knowing that all participants are connected to the same local network.

In response to the aforementioned issues, zero-configuration networking an its suite of protocols (mDNS/DNS-SD) [2, 3] intend to automatically discover devices and their services in a local area network, thus enabling new possibilities for device interaction in the application layer. As an example, the Mozilla Firefox[1] FlyWeb[2] extension leverages this suite of protocols to allow clients of Web applications to start their own local Web server from within the browser. Advertised in the local network through mDNS, other devices in the network can detect the new server and can connect to it via their own browser.

The FlyWeb extension caught our attention as it offers a range of new possibilities for Web applications and local vs. global network behavior. But, on its current implementation, it has severe limitations, one of them being the complete lack of fault tolerance: when the local server dies, the client-server network dies with it. Since this technology is inherently driven by the idea of any device being able to become a server, we assume that it is more likely that servers misbehave in comparison to the traditional scenario of "real" Web servers. We therefore regard fault tolerance as very important in such networks and we think the lack of mechanisms for graceful recovery is problematic.

We propose an approach to enrich FlyWeb with fault tolerance. We hypothesize that a technology like FlyWeb is a good basis for fault tolerance through replication, since any participant can become the server. In the situation of a failing server, we think it is intuitive that a client can become the "next" server and all other clients establish a connection to the new server. Obviously, replication comes at the cost of complexity. We intend to analyze the different strategies of replication and choose one that adheres best to our scenario. We aim to deliver our approach in a JavaScript library Successorships that provides fault tolerance to FlyWeb application developers without exposing the underlying technical details.

2 BACKGROUND

In this section, we detail the underlying concepts that are related to our project proposal.

2.1 Flyweb

FlyWeb is a Web API developed by the Mozilla Firefox community which enables clients of Web applications to publish a local server from within the browser. Building on the concept of zero-configuration networks and its mDNS/DNS-SD protocols [2, 3], the server advertises itself in the local network and can be discovered by other devices which become clients to the server by connecting via a HTTP or WebSocket connection. This essentially enables cross-device communication within a local-area network.

2.2 Zero-configuration Networks

Zero-configuration networking is a combination of protocols that aim to automatically discover computers or peripherals in a network without any central servers or human administration. Zero-configuration networks have two major components that provide *(i)* automatic assignment of IP addresses and host naming (mDNS), and *(ii)* service discovery (DNS-SD).

When a device enters the local network, it assigns an IP/name pair to itself and multicasts this pair to the local network, resolving any name conflicts that may occur in the process. IP assignment considers the link-local domain address which draws addresses from the IPv4 169.254/16 prefix and, once an IP address is selected, a host name with the suffix ".local" is mapped to that IP [3]. As devices are mapped to IPs/host names, their available services are discovered using a combination of DNS PTR, SRV, and TXT records [2]; their services can then be requested by other devices.

Despite smooth assignment of names and discovery of services, zero-configuration networks do not address client disconnection. As a consequence, when a device disconnects from the network, communication to that device ends abruptly.

[1] https://wiki.mozilla.org/FlyWeb
[2] https://flyweb.github.io/spec/

2.3 Replication

Fault tolerance and reliability in distributed systems with client-server architecture are generally achieved by data replication: information is shared on redundant server replicas such that any replica can become the new master if the current master fails. While improving system artifacts like fault-tolerance, reliability and availability, replication can come at the cost of performance: depending on the required operations in the system for replication, system performance can suffer significant bottlenecks. Different models of replication have been proposed to trade consistency for performance, resulting in different levels of consistency as a design choice for the target system. Traditionally, two strategies of replication are distinguished: *active replication* and *passive replication*. A third type of replication, *lazy replication*, was later introduced and is gaining more attention recently. The following paragraphs describe these three types of replication.

Active replication. The first strategy (also called *primary-backup* or *master-slave*), requests to the master replica are processed to all other replicas. Given the same initial state and request sequence, all replicas will produce the same response sequence and reach the same final state. Active replication has become most prominent with the introduction of the State Machine Replication model which was introduced in the 1980s [6] and later refined in [7]. It is based on the concept of distributed consensus with the goal of reliably reaching a stable state of the system in the presence of failures. While providing small recovery delay after failures due to an imposed total order of state updates, computation performance can suffer tremendous bottlenecks since updates must be sequentially propagated through all replicas.

Passive replication. The second strategy (also called *multi-primary* or *multi-master* scheme) relaxes sequential ordering: clients communicate with a master replica and updates are forwarded to backup replicas. Computation performance is improved with this pattern since all computation takes place on the master replica and only the results are propagated. The downside of the approach is that more network bandwidth is required if updates are large. Since the primary replica represents a single point of entry to clients with this approach, there must be some kind of distributed concurrency control in order to reliably restore state when the primary fails. This makes the implementation of this approach more complex and recovery potentially slower. One promising approach of passive replication is Chain Replication [9]. Whereas traditional topologies resemble stars with the master replica at the center, this approach forms a chain of replicas with the primary being the tail of the chain. This model aims at providing high availability without sacrificing strong consistency. The main advantage is that reads can address one end of the chain and writes the other. While recovery of failing tail or head servers is simple, recovery of failing middle-servers is complex.

Lazy replication. A third strategy of replication was proposed in 1990: *lazy replication* [4, 5] (also called *optimistic replication*) aims at providing highest possible performance and availability by sacrificing consistency significantly. With this approach, replicas periodically exchange information, tolerating out-of-sync periods but guarantee to catch up eventually. While the traditional approaches guarantee from the beginning that all replicas have the exact same state at any point in time, lazy replication allows states to diverge on replicas, but guarantees that the states converge when the system quiesces for some time period. In contrast to the strong consistency models used in the traditional approaches, lazy replication is based on eventual consistency which has gained more attention recently, in particular in online editing platforms, NoSQL cloud databases and big data [3]. Eventual consistency is the weakest consistency model, providing no guarantee for safety as long as replicas have not converged. Rather, it "push[es] the boundaries of highly available systems" [1]. The introduction of *conflict-free replicated data types* [8] aimed at a stronger model of eventual consistency: any two replicas that receive the same updates, no matter the order, will be in the same state. CFDTs are categorized in operation-based (only update operation is propagated) and state-based (full state is propagated). A number of CFDTs have been suggested, among them are sets, maps and graphs. It is important to mention that all eventual consistency models impact the application designer since she has to determine what level consistency is sufficient for the specific application.

3 PROPOSED APPROACH

Our goal is to build a framework to facilitate the development of offline client-server Web applications that robustly recover from server faults. We posit that the following features are prerequisites to achieving this:

(1) Nodes in the network trust each other;
(2) Any client, but exactly one client, has the ability to automatically assume the responsibilities of the server if the server goes down;
(3) All clients in the network have the capability to automatically update their connections to the new server in the event that the server migrates from one node to another;
(4) Communication in the network is not traffic intensive, i.e. nodes neither produce bursts of requests in a small period of time nor have payloads higher than some threshold τ that might generate bottlenecks in the network;

The model we propose for achieving this is one in which clients connecting to the server automatically acquire distributed state including the following elements:

(1) Constant: A UUID for the initial host node (the first to serve the application);
(2) The current state of the server;
(3) A “successorship” list: a list of (potentially not all of the) nodes in the local network, in order of “who is next” to assume server responsibilities, in the event that the server goes down;
(4) Constant: The actual server code to execute, in the event that one of the clients needs to begin acting as the server.

Note that the elements marked “(Constant)” are permanently fixed (for the lifetime of the application) when the initial server node first runs the application server.

We propose to develop a Javascript library that implements the functionality listed above, providing a clean interface to enable

[3] http://www.oracle.com/technetwork/consistency-explained-1659908.pdf, accessed 2017-10-08

developers to seamlessly integrate fault-tolerance into their offline client-server Web applications, without having to worry about the details of how such fault-tolerance is achieved.

3.1 API Overview

Our current running name for the library is `Successorships`[4], i.e. the next ship that will lead the flotilla after yet another sunk boat. We propose to implement the following interface in Successorships:

- Server side:
  - `initServer(name)`
  - `onReceive(msg, callback)`
  - `commitState(callback)`
- Client side:
  - `initClient()`
  - `connect(serverName)`
  - `send(msg, payload)`

We briefly discuss the major functions of our proposed API in the following subsections. Throughout the discussion, we use the TA queue example to illustrate our API usage.

Server initialization and service instantiation: the first functions to initiate a server are `initServer` and `onReceive`. The former starts the local server in the device's browser and, after initialization, assigns a host name for that server. The later register entry points for services offered by that server.

In our queue system, one would initialize a server and define two functions to handle requests to enqueue students and also to dequeue them once they are helped. Additionally, the server provides the queue service in order to provide the current state of the queue. If no recognizable service is requested, the TAQueue server responds with the queue service.

```
function getQueue(req, event) { ... }
function handleEnqueue(req, event) { ... }
function handleDequeue(req, event) { ... }

(function main(){
    server = sship.initServer("TAQueue");
    server.onReceive('queue', getQueue,
        default=true);
    server.onReceive('enqueue', handleEnqueue);
    server.onReceive('dequeue', handleDequeue);
})();
```

Establishing connections: as a server starts running, it broadcasts its name in the local-area network and clients in the same network can discover this server. A client device needs a single line of code to initialize itself. Upon initialization it will lookup for host servers in the network. Once a list of servers is retrieved and displayed, a client may select a server to connect to. In our TA queue example, we explicitly know the server name and skip the server list phase:

```
(function main(){
    client = sship.initClient().connect("TAQueue");
})();
```

Data exchange: clients can ask for services through the `send` function. The function explicitly takes a requested service as one of its parameters and a payload as its second one. In our queue system, two distinct clients may request to enqueue themselves.

```
(function main(){
    client1 = sship.initClient().connect("TAQueue");
    client1.send(enqueue,
        {student: "Arthur", csid: "cs4321"});

    client2 = sship.initClient().connect("TAQueue");
    client2.send(enqueue,
        {student: "Paul", csid: "cs9876"});
})();
```

Updating the server state: Finally, it is necessary to define which data structures or variables are important for a server, thus the `commitState` function receives a function which is executed every time that a service is successfully requested and executed in that server. Revisiting our `server = sship.initServer("TAQueue")` code snippet, we would add a final function to define how the server would be updated after queueing/dequeuing students.

```
var queue = [];
function currentQueue() { return queue; }

(function main(){
    server = sship.initServer("TAQueue");
    ...
    server.commitState(currentQueue);
})();
```

3.2 What's happening under the hood?

When a server is initialized and it starts running, we envision that our API will create a state for that server and that this state is updated after the execution of any received message callback.

As clients connect to a server, the clients themselves assign their own host names and the server keeps a list of connected clients. Upon connection, the clients start a heart beat routine to monitor the server state. Clients are updated in two scenarios, the first one happens through one acknowledgement in the heart beat cycle while the other happens whenever a client receives an acknowledgement from a service usage. In both cases, the acknowledgement carries the server's current state and clients maintain a replica of that state within their storage memory.

Upon failed messages, the clients will start their own servers with the same name as the one that they were connecting to, but followed by a randomly generated unique suffix. As this new server is started and broadcasted in the network, clients will detect other servers with the original server prefix and thus, they will have to reach a consensus among themselves on who will be the next leading server. Through this process, they will also exchange their own server states such that they can eventually identify which client has the most up-to-date state.

4 EVALUATION

Due to the nature of our project, i.e. an offline fault-tolerant client-server Web browser API, our evaluation will be twofold. First, we want to measure network traffic in this offline network and then compare it against a traditional client-server Web application. Despite having different network characteristics: local-area vs Internet, this comparison will help us identifying and discussing possible benefits and drawbacks of our approach. Second, we want to measure network traffic in the face of failures. How much network traffic is required to achieve stability once a server device fails?

[4] We still need a proper acronym for it, e.g. sship

How long does it take? Is our approach scalable? These are some of the questions that we want to answer with the second evaluation.

In order to measure network traffic, we will rely on a packet analyzer such as Wireshark[5]. Regarding traffic comparison, we will monitor traffic in an application running in *(i)* a traditional client-server architecture, *(ii)* a default FlyWeb implementation, and also *(iii)* in our fault-tolerant API. We aim to compare traffic in each one of these scenarios and discuss their differences. For the traditional client-server vs. FlyWeb comparison, our purpose is to evaluate differences in delay while for the within-FlyWeb comparison, we want to compare network overhead generated by our fault-tolerance strategy.

As for our second evaluation, we will write scripts that simulate client connecting to a server device through our `Successorships` API. Once a set of clients establish communication, our simulation will then remove the server device from the network such that we can evaluate how `Successorships` handles failures.

Regarding our simulations, we will consider a queue system as a baseline application. The queue will have i configurable consumers, and j configurable producers which will produce-consume queue entries at random given times. Such entries will also have a configurable payload size. Such system will give us a local-area network with $n = i+j$ devices and it will also allow us to experiment with different size configurations. For instance, we can compare scenarios with a small, medium, or large number of devices connected.

As a risk, we are not entirely sure if it is possible to run this evaluation solely with Wireshark and scripts simulating our consumer-producer queue. However, we will continue researching and exploring evaluation possibilities for our final proposal.

5 TIMELINE

We consider late November as the project final deadline. With that in mind, there are a few key activities that we define as milestones, such that we keep up with the project schedule.

- **Oct 25th:** Study and evaluate replication patterns. Build a sample FlyWeb queue application;
- **Nov 1st:** Implement the core functionalities of the Successorships API.
- **Nov 8th:** Continue API implementation. Start writing scripts for evaluation;
- **Nov 22nd:** Wrap-up API. Write scripts to analyze and plot data; Start drafting project report.
- **Nov 30th:** Supposed project deadline?

REFERENCES

[1] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, Mar. 2013.

[2] S. Cheshire and M. Krochmal. Dns-based service discovery. RFC 6763, RFC Editor, February 2013.

[3] S. Cheshire and M. Krochmal. Multicast dns. RFC 6762, RFC Editor, February 2013.

[4] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the 4th Workshop on ACM SIGOPS European Workshop*, EW 4, pages 1–6, New York, NY, USA, 1990. ACM.

[5] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[6] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6/2:254–280, April 1984.

[7] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[5] https://www.wireshark.org/