Building a distributed, fault-tolerant, offline web

Arthur Marques Felix Grund Paul Cernek

October 7, 2017

Abstract

paper instructions: https://www.cs.ubc.ca/ bestchai/teaching/cs527_2017w1/proposals.html

1 Introduction

2 Background

2.1 Flyweb

FlyWeb[1] is a web API developed by the Mozilla Firefox community[2] which enables web applications to connect and communicate to each other over a local-area network. To do so, FlyWeb relies on a zero-configuration network and its mDNS/DNS-SD protocols [2, 1]. With FlyWeb, a web page may be seen as a device, which hosts a server. This device advertises itself to other client devices in the local network and, eventually, these other clients will discover and connect to the advertised server. Thus, connected clients/servers enable cross-device communication.

2.2 Zero-configuration Networks

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

Roughly, when a device enters the local network, it assigns and IP/name to himself and then multicast that to the local network, resolving any name conflicts that may occur in such process. IP assignment considers the link-local domain address, which draws addresses from the IPv4 169.254/16 prefix and, once an IP is selected, a host name with the suffix ".local" is mapped to that

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

IP [2]. As devices are mapped to IPs/host names, their available services are discovered using a a combination of DNS PTR, SRV, and TXT records [1], thus their services can be requested by other devices.

Despite smoothly assigning names and discovering 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.

2.3 Replication

(work in progress...)

Data replication is one of the major design approaches to achieve reliability and fault-tolerance in distributed systems: information is shared on redundant replicas such that any replica can become the new master if the current master replica fails. While enabling the system artifacts of fault-tolerance, reliability and availability, replication comes at a 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 which resulted in different levels of consistency as a design choice for the target system.

In order to support replication in the client-server paradigm present in network applications, the State Machine Replication model was proposed in the 1980s in [3] and later refined in [4]. It is based on the concept of distributed consensus in regard to reliably reaching a stable state of the system in the presence of failures. [3] introduced the strategy of *active* replication (also called *primary-backup* or *master-slave* scheme) where 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.

In the passive scheme (also called multi-primary or multi-master scheme), requests are first processed on the master replica and the resulting state is distributed to other replicas.

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. the ability for any client, but exactly one client, to automatically assume the responsibilities of the server if the server goes down

2. the ability for all clients in the network to automatically update their connections to the server in the event that the server migrates from one node to another

3. (Optional, if we have time) the ability for the initial server node to resume responsibilities of the server once it comes back online (and can be reasonably believed to be robustly online)

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

- Constant: A GUID for the initial host node (the first to serve the application)
- The current state of the server
- 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
- 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 spins up the application server.

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

Our current running name for the library is `ftcs`, short for "fault-tolerant client-server". We propose to implement the following interface in `ftcs`:

- Server side:
  - `server = ftcs.initServer(name)`
  - `server.onReceive((msg, src) => { })`
  - `server.commitState(state)`
  - `server.commitChange(change)`
  - `cur_state = server.getState()`
- Client side:
  - `connection = ftcs.connect(name)`
  - `connection.onReceive((msg) => { })`
  - `connection.send(msg)`

3.2 Limitations of client-server model

The client-server model describes a scenario in which one or many clients require a service or resources from a centralized server.[3] Tacit in this model is the assumption that the server has access to resources that are unavailable to the client, whether this be compute power, storage, sensitive data, etc. Therefore, it seems potentially misguided to seek to migrate server functionality to arbitrary clients; it seems like we might be shoe-horning the client-server application model into working as a peer-to-peer service. It follows that our project might gain some clarity from listing some concrete use cases that would benefit from this fault tolerant behaviour, all the while maintaining the appearance of client-server communication.

One example we have come up with so far is a queue application, e.g. for a TA to use during office hours: the TA spins up the application on her phone, and as students enter the room, they connect to the server on the TA's phone and request to be enqueued. Maybe the TA needs to leave the room momentarily, and therefore has to leave the local network. In this event, we wish for the entire application state, and even the ability of new students to enqueue as they arrive, to persist even as the initial server host leaves the network ("distributed failover"). When the TA returns, the application seamlessly returns to being hosted on her device ("failback"). In this example, however, we need to be explicit about the gains obtained from adhering to this client-server model, rather than e.g. implementing this application as a peer-to-peer application.

3.3 Limitations of FlyWeb

FlyWeb requires all application users to be connected to a common local area network (LAN) that enables multicasting.

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, this comparison will help us in identifying and discussing possible benefits and drawbacks of our approach. Second, we want to measure network traffic in face of failures. How much network traffic is required to achieve stability once a server device fails? 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[4]. As for our second evaluation, we will write scripts that simulate clients connecting to a server device through our `ftcs` API. Once a set of clients

[3] https://en.wikipedia.org/wiki/Client-server_model
[4] https://www.wireshark.org/

establish communication, our simulation will then remove the server device from the network such that we can evaluate how `ftcs` handles failures.

5 Timeline

References

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

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

[3] 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.

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