Building a distributed, fault-tolerant, offline web

Arthur Marques Felix Grund Paul Cernek

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.

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 FTCS that provides fault tolerance to FlyWeb application developers without exposing the underlying technical details.

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

2 BACKGROUND

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.

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.

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) 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.[4] 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

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

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

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[5]. 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 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] 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.

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