Successorships - Fault-Tolerant Local WebApps

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

ABSTRACT

As networking capabilities become more ubiquitous across different types of devices, applications that communicate over local area networks are becoming increasingly common. This trend has been accelerated by the adoption of zero-configuration (zeroconf) networking standards that eliminate the burden of setup procedures. Applications operating in zeroconf settings often face the challenge of maintaining reliability and consistency in the face of wireless links and mobile devices, resulting in potentially intermittent connectivity between hosts. Fault-tolerance is thus a desirable property for such applications, but it can be difficult to achieve in practice. To this end, we introduce Successorships, a JavaScript library that provides the appealing qualities of zeroconf enriched with fault-tolerance, to applications running in the browser. Our library enables graceful recovery after server failure by handing over the server role to one of the clients currently in the network. We evaluate our approach with a sample application, focusing on usage scenarios that involve interactions between users in a small mobile network. We find that our library recovers from failures gracefully with an average of 15 seconds and that application state is maintained with eventual consistency.

1 INTRODUCTION

In recent years, applications that communicate over local area networks have become increasingly prevalent; prominent examples include Apple Bonjour, Spotify Connect, Google Chromecast and ad-hoc Wi-Fi printers. This trend has been accelerated not only by the proliferation of "smart" devices, but also by the adoption of standards that eliminate the burden of manually configuring devices in a network. Zero-configuration networking has become one of the most widely adopted such standards, and its suite of protocols allows devices to automatically discover other devices or peripherals in a network as well as their offered services [2, 3].

While most applications are shipped with specific hardware or software to publish and discover services in a network, recent innovations enable this to be done programmatically, making it easy to devise new applications that communicate within a local area network. As an example, browser vendors are integrating this technology and delivering a new set of Web applications built on top of Zeroconf. Indeed, the characteristics of Web applications are appealing for Zeroconf applications: since they obviate traditional installation models, browser-capable devices may become the only requirement for both server and client roles, as is the case in the example of Mozilla FlyWeb.

While we see strong potential for Zeroconf Web applications, we argue that they still face the common challenges imposed by the type of network in which they operate. Specifically, maintaining reliability and consistency in the face of wireless links and mobile devices may result in intermittent connectivity between hosts and jarring user experience as a consequence. Therefore, we consider fault-tolerance and graceful recovery from failures to be highly desirable properties for such applications. To the best of our knowledge, no current APIs offer built-in fault-tolerance to local-area in-browser applications.

To this end, we introduce Successorships, a JavaScript library intended for Zeroconf browser applications, that provides fault-tolerance functionality to developers, while completely abstracting away the details of its implementation. Our library enables graceful recovery after server failure by handing over the server role to one of the clients currently in the network. By exposing an easy-to-use API, we abstract away the complex details of Zeroconf communication, state replication and consistency and participants' roles of clients and servers. We evaluate our approach with a Successorships application used in a small mobile network. Measurements considered 29 server failures and subsequent recoveries. Results indicate that usually we are able to recover from failures under 15 seconds. Additionally, we discuss characteristics of Successorships network traffic.

We make the following contributions:

- Successorships, a JavaScript framework with a simple API that seamlessly enriches Zeroconf Web applications with fault-tolerance through state replication
- A reproducable evaluation methodology for Zeroconf Web applications
- Example applications that demonstrate the benefits and usability of our framework

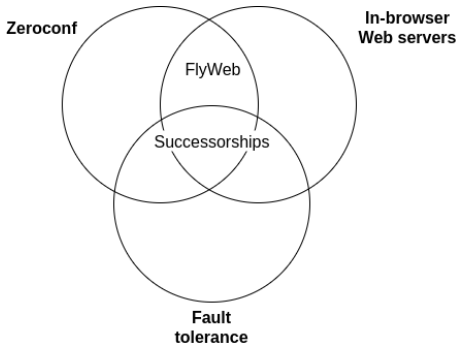
The remainder of this paper is organised as follows: Section 2 describes the promise of Zeroconf and in-browser Web servers, and motivates the case for extending fault tolerance to applications built on top of these technologies. Section 3 describes our approach, Successorships, which we then evaluate in Section 4. We suggest some limitations of our approach, as well as some ideas for future improvements, in Section 5. Section 6 situates our work in the context of related research, then Section 7 concludes the paper.

2 BACKGROUND & MOTIVATION

Our work lies at the confluence of three distinct areas of networking and distributed systems: Zero-Configuration Networks (Zeroconf), in-browser Web servers, and fault tolerance. Here we provide an overview of these topics.

2.1 Zeroconf

Hosts connected to a network rely on maintaining a common consensus on such basic matters as addressing and name resolution, if they are to have any hope of communicating over that network. This need is often fulfilled by specially-designated configuration servers, such as DHCP or DNS servers. However, such servers can



be absent in common networking situations, such as *ad hoc* networks. In such cases, it remains desirable for hosts to handle these matters in a seamless and cross-platform fashion, without the need for manual configuration. This is the problem addressed by the group of protocols referred to as Zeroconf, short for "Zero Configuration Networking".[1] Zeroconf protocols solve the problems of address allocation, address queries (mDNS), and network service discovery (DNS-SD), specifically when hosts share a direct link (be it logical or physical). Since the first of these (address allocation) is a *de facto* standard, as it has been integrated into consumer operating systems and printers since the early 2000s, we focus on the latter of these two here.

Address queries. A name server can be used on a local network to maintain a mapping of logical names to IP addresses. In the absence of such a server, a Zeroconf protocol called mDNS (short for "multicast DNS") can be used to formulate address queries on the local network. As per its name, mDNS broadcasts these queries onto the local network rather than directing them to a single server; it also includes a mechanism to resolve naming conflicts [3]. mDNS aims to handle any type of record lookup that could be handled by a DNS server, not just name-to-address lookups. mDNS has grown steadily in popularity amongst networked devices; implementations of mDNS include Apple Bonjour, Spotify Connect, Philips Hue, Google Chromecast, and Avahi (an open-source implementation for Linux).

Service discovery. Once connected to a network, a host may wish to learn about the services offered by the other devices connected to the network, such as printing or media streaming. In a centrally-administered network, this may be accomplished by a directory server, such as a DNS server. However, when such a server is lacking, the Zeroconf protocol called DNS-SD ("DNS service discovery") leverages the ability to make distributed DNS queries provided by mDNS, to register, enumerate, and resolve local network services [2].

2.2 In-Browser Web Servers

More recently, a number of technologies have emerged to enable Web servers to be deployed entirely from within Web browsers. Such technologies have the appeal that they simplify the process of serving Web content, thus potentially expanding the breadth of users capable of doing so. For example, Opera Unite[2], Web Server Chrome[3], PeerServer[4], and FlyWeb[5] all enable browser applications to publish fully functioning Web servers. We discuss FlyWeb below, and defer details on a few other approaches to Section 6.

2.3 The power of Zeroconf in the browser

HTTP servers running on hosts that usually access the Web as clients face a few challenges not normally encountered by those running on dedicated server machines. For example, they must be properly configured to circumvent firewalls, and they must ensure that other clients have a way of finding out their IP address.

The FlyWeb project, developed by the Mozilla Firefox community, addresses the problem of advertising and discovering in-browser Web services in the particular environment of local networks, by leveraging Zeroconf service advertisement and discovery. To this end, FlyWeb provides two key pieces of functionality: *(i)* an implementation of mDNS, allowing those services to advertise their name and address to peers on the local network, and *(ii)* a FlyWeb service discovery menu, which uses a built-in implementation of DNS-SD to enumerate locally-discovered services. The goal is for devices on a local network to be able to stream applications and content to one another using widely available Web technology.[6]

An important part of the appeal of this approach is its sheer versatility: it empowers any Web application with the ability to connect heterogeneous devices over an *ad hoc* network, without each user needing to download a native app for their particular platform. Examples of successful demonstrations of this idea include a collaborative photo sharing app, a printer interface, a temperature monitoring interface, and even a quadcopter controller.[7]

2.4 The need for availability

A different challenge facing in-browser Web servers is that their availability is limited by that of the host machine. Dedicated server machines usually have static network addresses and are often streamlined for serving Web content; however, the class of devices that can run a Web browser is much wider, including mobile devices, hence posing a novel challenge to service availability. To the best of our knowledge, at the time of writing, none of the currently-existing technologies address the issue of recovering from disruptions of server availability for in-browser Web services.

We argue that application availability in the face of server failure is a requirement for many common use cases of client-server Web applications running on local networks. This is especially true when one considers the current prevlance of networked mobile devices, where the server node might leave the local network, or fail due to other reasons such as low battery.

To motivate this concretely, consider a simple queuing Web application called `QueueApp`, which might be useful in the following scenario: a TA[8] holds office hours in a small classroom, and students arrive at random intervals seeking individual assistance. Students

[1] Although the term is sometimes used more broadly to designate any suite of technologies that seeks to address this problem, we observe the more specific meaning established by the former IETF Working Group by the same name. See e.g. http://www.zeroconf.org/.

[2] http://help.opera.com/Windows/12.10/en/unite.html
[3] https://github.com/kzahel/web-server-chrome
[4] http://www.peer-server.com/
[5] http://flyweb.github.io/
[6] https://hacks.mozilla.org/2016/09/flyweb-pure-web-cross-device-interaction/
[7] https://github.com/flyweb/examples
[8] Teaching Assistant

may arrive at a rate that exceeds that at which the TA is able to address their concerns, and so the TA wishes to keep track of the order in which they arrive. The TA takes out her smartphone and loads QueueApp, which immediately starts a Web server on her phone. As students arrive, she directs them to the URL of the local QueueApp service; when they connect, they are faced with an interface that enables them to either join the queue if they are not already in it, or to leave it if they are. Now, say the TA needs to temporarily leave the room, for example to take an important phone call. Unless special functionality is implemented by the application, the clients (students) will get disconnected from the server (TA), and the application state will need to be reset.

We argue that in such a case, it would be of great practical value for QueueApp to continue functioning seamlessly, even in the absence of the initial server. What if 5 students enter the room while the TA is out? In this paper, we make the case that those students should be able to enqueue despite the absence of the TA.

In the following section, we explain how we achieve this in the Successorships library.

3 SUCCESSORSHIPS

Our goal is to provide a framework to build Zeroconf Web applications and expose an easy-to-use API to application developers. We aim to provide fault-tolerance seamlessly without the developer having to deal with the underlying details of state replication and consistency challenges. We implemented our approach as a JavaScript library and describe it in the following sections. We first cover a few necessary assumptions to make this implementation tractable and then explain our exposed API. We then continue with a more detailed conceptual description and our replication strategy and consistency guarantees. Finally, we elaborate on what failure scenarios are handled by our approach.

3.1 Assumptions

We make some strongly simplifying *assumptions* in the first version of Successorships to make initial design and implementation tractable. In particular, we assume the following, being aware of necessary improvements for a productive version of our implementation.

Assumption 1: The nodes in the network trust one another. An implication of our model is that any node in the network (with a reliable connection) may at some point become the application server. This status comes with all the responsibilities of the server, including running the server logic, and hence maintaining the server-side application state. Without any further substantial design considerations, this model would be extremely susceptible to a malicious client acquiring server status, and exploiting this status for their own ends, either by serving downright malicious material, or by more subtly forging application state.

Assumption 2: Updates to server-side state are small. A core requirement of our model is that the server be able to broadcast its changes in state to all clients in the local network with relatively low latency, to diminish the possibility of clients' copies of server state being out of sync. Therefore, we assume that communication in the network is not traffic intensive, i.e. nodes neither produce bursts of requests in a small period of time, nor do they have payloads higher than some threshold τ that might generate bottlenecks in the network.

Assumption 3: Applications are restricted to a client-server model with one state object. Successorships apps follow a client-server model and are built using one shared object. While this may sound very limiting at first, we can see this becoming a norm in modern Web applications, especially with the popularity of frontend frameworks like React[9] and Redux[10]. In these frameworks, any change in the application's frontend is triggered by an operation on a single state object. We posit that Successorships aligns very well with these frameworks and consider this design choice well-justified.

3.2 API Overview

Successorships – or shortly *Shippy* – provides a framework for fault-tolerant local area Web apps. Its API is designed to hide underlying details of the mDNS and DNS-SD protocols, state replication, and the distribution of client and server roles. The goal of this design is to spare the app developer the complexity of network behavior and let her focus on the implementation details of the app itself. The library is shipped as a JavaScript file `shippy.js` to be included in HTML files of the Web app. When loaded, all functionality is exposed on a JavaScript object `Shippy` that resides as a property on the browser's global `window` object. This object is the only place of interference with the browser's global namespace to avoid naming collisions with the app's environment.

The API consists of three methods exposed on the `Shippy` object as shown in Listing 1. Using these three methods, Shippy apps:

(1) Describe their operations on the app's replicated state (`Shippy.register`);
(2) Trigger these operations when required (`Shippy.call`);
(3) Listen to events dispatched by the framework (`Shippy.on`).

It is important to highlight that the aforementioned methods will be called from all nodes without knowledge of underlying client or server roles. These characteristics will remain within the library and only changes to the current state and other information will be dispatched to the app with events.

```
Shippy.register(serviceName, {
    init: function
    operations: <object:string=>function> });
Shippy.call(operationName:string, operationData:object);
Shippy.on(eventName:string, callback:function);
```

Listing 1: Successorships API

Usage of the API is best described with the QueueApp example from section 2.4. The app declares a function `init` that describes how the initial state (initially an empty object) should be changed. In the case of our QueueApp the state will contain a simple array for the queue. Listing 2 shows how this queue variable is added to the state object.

```
let init = function(state) { state.queue = []; };
```

Listing 2: QueueApp init function

The operations of the QueueApp are addition and removal of names to and from the queue. These functions will be called by

[9] https://reactjs.org/
[10] https://redux.js.org/docs/introduction/

the current server node whenever the respective operations are invoked by clients (using `Shippy.call`). By definition, the operation functions will be called with two arguments: the current `state` object and the `params` given to `Shippy.call`. Listing 3 shows code snippets for the operations of the QueueApp.

```
let operations = {
    add: function(state, params) {
    /*add params.name to state.queue if not existing*/},
    remove: function(state, params) {
    /*remove params.name from state.queue if existing*/}
};
```

Listing 3: QueueApp operations

The declared `init` function and `operations` object are then registered at the Shippy framework by calling `Shippy.register`. Listing 4 shows the QueueApp register function. The first argument will be the name of the service to be advertised in the local network for the app, *QueueApp* in our case, while the second argument contains the app init function and its set of operations (note that we refer to the previously declared variables as property values).

```
Shippy.register("QueueApp", {
    init: init, operations: operations });
```

Listing 4: Shippy.register

Once the app is registered and a node is connected as client, operations can be invoked with `Shippy.call`. Listing 4 shows how the client node for Bob would add its name to the queue.

```
Shippy.call("add", { name: "Bob" });
```

Listing 5: Shippy.call

Underneath, this invocation of `Shippy.call` will result in a message sent on the WebSocket channel to the current server node. It will contain the operation to be executed, *add*, and the payload `{name: "Bob"}`. On the current server node, the operation name will be matched against the set of operations provided earlier with `Shippy.register`. If an operation is found, the respective function is called with the current state object and the received payload (containing the name "Bob" in this case) as arguments. After the change was applied, an event *stateupdate* is triggered and broadcasted to all clients such that they can apply the operation to their state in the same way. Eventually, the app on each client will be notified with the *stateupdate* event providing the newly computed state. Listing 6 shows the listener for this event in the QueueApp. As an example, apps can update their UI if they detect changes in the state.

```
Shippy.on("stateupdate", function(state) {
/* update app based on new op. (e.g. UI changes) */});
```

Listing 6: stateupdate listener

In addition to the *stateupdate* event, Shippy currently dispatches *connect* and *disconnect* events such that apps can update their interface in accordance with the current network state (e.g. changing background color). By design, this mechanism of loose coupling is extensible with many more events without breaking existing Shippy apps.

3.3 Conceptual Description

Architecture. Figure 1 shows the Successorships stack. On the bottom layer are the *Application layer protocols*. The Zeroconf protocols *mDNS* and *DNS-SD* are used for the advertisement and discovery of Shippy services in the local network. In terms of application-level network communication we use the common protocols of *WebSocket* and *HTTP*. The initial request to a Shippy app is in form of a *HTTP GET* request and associated requests for static resources. Clients then establish a stable *WebSocket* channel that is used for further communication between nodes (e.g. for state replication). Above the application layer protocols is the *FlyWeb* layer. We use FlyWeb as a library for facilitating interaction with the Zeroconf protocols. Our implementation was designed with as few connection points with FlyWeb as possible such that this dependency can be easily replaced with a different implementation in the future. Above the FlyWeb layer is the *Successorships* framework that exposes the API described in section 3.2 to its applications that are located in the topmost layer.

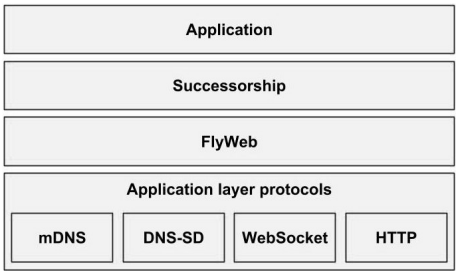


Figure 1: Successorships Stack

Roles and shared state. When a Shippy app is loaded in the browser the new node becomes either a client or server node. If it becomes a server node, it becomes a client node to "itself" shortly after. The application's global state is replicated and shared among all client nodes (see section 3.4 for our replication strategy). The global state can contain arbitrary application data and global metadata accessible only by the Shippy library. One such required metadata field is a `successors` list containing a list of current clients, except the client node that is currently also the server. This list is used to determine which node should become the next server node upon failure of the current one. Figure 2 depicts the distribution of roles. Node *A* is currently the server with nodes *B...n* being clients. Both server and clients have their own replica of the state, accessible through the global state.

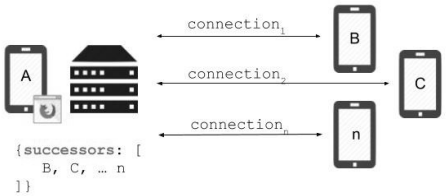


Figure 2: Successorships Roles

Service discovery. The Shippy library comes with a compulsory Firefox add-on responsible for notifying apps with the current set of available Shippy services. In the add-on, we register an event listener to the FlyWeb *DNS-SD* component, which dispatches a list of current local services in frequent intervals. We sample these events

to a maximum frequency of 100ms[11] and filter out all services that were not published in the FlyWeb context. We then dispatch our own events containing the current list of FlyWeb services with the service name, IP address and port on the Browser's global `window` object[12]. These events are then accessible by event listeners in Shippy apps as illustrated in listing 7.

```
window.addEventListener('flywebServicesChanged',
     function(event) {
    let services = event.detail.services;
    // e.g. [{ serviceName: "QueueApp",
    // serviceUrl: "http://206.12.69.249:51629" }]
});
```

Listing 7: Event listener for service discovery

Service publication. A Shippy node will publish a service and hereby become a server node in certain cases depending on service discovery events and the current list of successors in the shared state. The decision of whether to become a server is first triggered when (1) *the app is loaded* in the browser and there is currently *no service discovered with the app's name*. This decision is then repeated (2) any time a *discovery event* occurs and there is *no such service*. In both cases, the node will become a server if *either* of the following conditions is true:

- The `successors` list is empty *and* this node loaded the application initially.
- The `successors` list is not empty *and* the first item in the list refers to this client.

If one of the above conditions is met in a decision phase, a service with the app's name will be published and that node will become a server. This will then result in a service advertisement in the local network using the *mDNS* protocol.

Client connection. A Shippy node will become a client when it discovers a service and connects to it. To do so, *there must be a Shippy service running with the app's name* and *the node must be currently disconnected*. When a node becomes a client, it establishes a *WebSocket* connection to the current server as the first part of the initial handshake. On the server side, a unique ID is created for the new client and appended to the `successors` list. The updated `successors` list is then broadcasted to all currently connected clients. The created ID is sent to the new client as the second part of the handshake. The client node then saves its ID such that it can be matched against the IDs in the `successors` list in failure scenarios.

Client succession. Whenever the current server fails, a successor must be appointed. To determine the successor, clients match their ID against the elements in the `successors` list. If their ID matches the first element, this client will become a server node. Otherwise, it will wait for the new server and connect to it as client. Referring to the scenario in Figure 2, suppose that server `A` fails. The connected clients `B` to `n` detect that failure and match their ID against the ones in the `successors` list, as presented in Figure 3. Node `B` will become a server node as its ID matches the first element in the list while other nodes will wait and connect to `B` as clients nodes.

[11] We observed many duplicate and overly frequent event triggers from the `FlyWebDiscoveryManager` making this sampling necessary.

[12] There are a few limitations in this approach that we address in section 5.

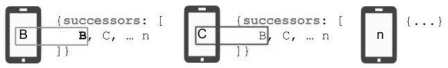


Figure 3: Successorships client succession

Server implementation. We built on the concept of in-browser Web servers based on the `navigator.publishServer` implementation in FlyWeb[13]. This implementation provides an API for received *HTTP GET* requests and *WebSocket* messages. Since any Web application relies on static resources (e.g. js, css, images) our FlyWeb server implementation must be able to load and serve these resources when requested by clients. The documentation of FlyWeb[14] describes that this can be achieved using the browser's *Fetch API*[15]. However, this is essentially just a redirect to the Web app that served the FlyWeb app at the first place. Since, after a server failure, the original Web app and its static resources will not be available to clients that took over the server role from another node, Shippy must store static resources at every node. We solved this problem by storing all resources required for the Shippy app in the browser's `sessionStorage`[16] that is persistent for the duration of the current browser session. In the current implementation, all resources obtained upon load of the app are scanned for referenced resources. These are then fetched and stored in `sessionStorage`. Subsequently, when Shippy request static resources, these can be simply obtained from `sessionStorage` and served without any access to the original Web app.

3.4 Replication Strategy and Consistency Guarantees

State replication. All Shippy nodes share a versioned global state during the lifetime of the app. The app is "alive" as long as there is one node that advertises the app's service in the local network. Each time the state is changed on the server node, the state's version is incremented. The current state is obtained initially by clients triggered by the first handshake with the current server: whenever a new client connects, a WebSocket `open` event is triggered on the server and a new unique client ID is added to the `successors` list in the global state object. The changed state is then broadcasted to all clients, including the new one. This ensures that any new client is up-to-date shortly after connecting and that all other clients will have the new client in their `successors` list. After a client is connected, it can invoke state changes on the server by calling `Shippy.call` which will result in a WebSocket message to the server where the submitted operation is applied. After an operation was applied on the server, this operation is broadcasted to all clients in the order as they appear in the `successors` list in the shared state.

Since the state is always changed on the current server first and changes are then broadcasted to clients, clients can simply apply the changes locally in ordinary cases. However, there can be exceptions in certain failure scenarios due to race conditions and

[13] https://github.com/flyweb/spec

[14] https://flyweb.github.io/posts/2016/11/01/introducing-flyweb.html, accessed 2017-12-09

[15] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API, accessed 2017-12-09

[16] https://developer.mozilla.org/de/docs/Web/API/Window/sessionStorage, accessed 2017-12-09.

the asynchronous nature of WebSocket broadcasting. We cope with such failures by versioning the state: when clients receive state changes (by listening to `stateupdate` events), they will check the version of the server's current state and compare it with the version of their local state. In case their local state is newer than the server state, they will send a `_aheadofserver`[17] message to the server with their current local state as payload. The server can then accept this newer state and again broadcast a state update. Note that even if several clients send `_aheadofserver` messages, the server will only accept these if their state's version is higher than its own, which indicates that consensus will be reached eventually. In the current Shippy implementation, we consider clients being "ahead" of the server a hypothetical scenario that we could not produce in any of our evaluated use-cases.

Reaching consensus. We leverage the concepts of *eventual consistency* and *lazy replication* in Shippy [7, 8]. Briefly, this means that nodes holding replicas of the state periodically exchange information, tolerating out-of-sync periods; thus, state may diverge across nodes, but is guaranteed to converge when the system quiesces for some period of time.

We argue that our domain of local ad-hoc offline Web applications will mostly tolerate minor time frames of inconsistent states and we therefore focus on availability and performance rather than accepting the overhead induced by stronger consistency principles. With the replication strategy previously described, the state manipulation can only happen on the current server and clients are only notified of changes applied on the server. The most obvious source for inconsistencies would be lost messages carrying operations from server to client and vice versa. However, this potential problem is mostly mitigated by the design of the WebSocket protocol: each client is connected to the server with a stable bi-directional channel. Whenever this channel is interrupted, an `error` or `close` event will be triggered on both the client and the server side of the channel. In such cases, the server will simply remove the client both from its maintained collection of WebSocket clients and the `successors` list on the shared state (resulting in a `stateupdate` broadcast). The client on the other hand, will try to create a new connection to the server, which will then result in obtaining a fresh copy of the server state. While we are certain that all nodes will eventually converge to a consistent state with our approach in most scenarios, we cannot eliminate the risk of inconsistencies in certain edge cases (see section 5).

3.5 Handled Failure Scenarios

With our replication and consistency strategies, we handle different scenarios of failures gracefully.

Client disconnection. While disconnecting clients are mostly a trivial circumstance in traditional Web apps, they are more complex in Shippy apps because the system relies on clients being represented in the shared `successors` list. A disconnecting client will trigger a `close` event on the current server for the associated WebSocket channel and the disconnected client's ID is obtained. The ID is then removed from the `successors` list on the shared state and the update is broadcasted. If the disconnected client was the first element in the `successors` list, the list will be shifted and the second element will be the new successor.

Server disconnection. A disconnected server will result in a WebSocket `close` event triggered at each connected client. After some time, the service discovery mechanism described in Section 3.3 will reveal that there is no more service currently registered with the app's service name. This is the time when one of the previously connected client nodes will become the next server[18]: each client compares the first item in the shared `successors` list with its ID (obtained with the `welcome` message of the initial handshake). If it matches, the client will publish a server as described in Section 3.3. Otherwise, it will connect as soon as a service with the app's service name is available in a subsequent service discovery event.

Simultaneous disconnection. In this scenario, the server and one or multiple clients fail within a short time frame. For example, both the server and the first successor could disconnect and other clients would not receive the update that the first successor disconnected. If we did not target this case specifically, no client would become the new server because they all expect a disconnected client to do so. We solve this problem with a *repeated timer* algorithm: clients that are not the first item in the `successor` list set a timer that will prune the first item from the list after some time. This interval is set sufficiently to give the current first successor enough time to publish the service. The interval is varied with a random number to make it highly unlikely that two clients prune successors at about the same time which could result in race conditions publishing a service. At some point in this process, one other client will be the first in the `successors` list and will become a server. Other clients will be notified eventually by a service discovery event and connect to the new server.

4 EVALUATION

In this section we evaluate the efficiency of Successorships in detecting server failures and in subsequent recovery (Section 4.2). We then evaluate the time it takes for a new client to fully join a running Successorships app, i.e. to establish a WebSocket connection to the running server and receive the server state with the `successors` list (Section 4.3). Finally, we measure package traffic at the devices connected with a Successorships app (Section 4.4).

4.1 Methodology

In order to evaluate Successorships, we created a small mobile network using an Android Pixel 2 smartphone mobile Wi-Fi hotspot. We aimed to simulate intended uses cases where Successorships could be used in practice. For instance, a group of executives could use their smartphones and share a business presentation with their peers, or teaching assistants could use their phones and start a queue application, as previously explained in our motivating example (Section 2.4).

[17] We precede messages not associated with app-defined operations with an underscore.

[18] We considered starting the next server immediately subsequent to a WebSocket close event but restrained from this approach, considering it too error-prone. For example, a simple page reload on the current server would trigger a WebSocket `close` event on all clients and one of them immediately becoming the next server. This would introduce another race condition between the first successor and the original server that registers itself again immediately after page reload. Our existing approach deals with this problem much more gracefully: the original server will simply remain the server because the set of current services is updated much less frequently. The downside of our approach is a significantly increased time of recovery.

Table 1: Experiment Summary

# Server Recoveries	# Client Connections	# Operation calls
29	257	28

We connected to our ad-hoc network with three MacOS computers, among which one acted as the starting server launching an instrumented version of a queue application built using the Successorships API. All devices used Firefox Aurora Nightly build `50.0a2` and the three researchers acted as users of the queue application.

Throughout our evaluation we followed a small script describing tasks to be executed at each of the devices. After the server was started, each device *(1)* discovered the running server in the network, *(2)* connected to the server and *(3)* executed either an enqueue or dequeue operation. Afterwards, *(4)* the running server disconnected and *(5)* the successor node started a new server. Finally, *(6)* all existing clients (re-)connected to the new server.

At step *(3)* we added an extra field to the payload of the called operations that contained a random string with variable size drawn from a long-tailed power-of-two distribution. By adding this field, we aimed to investigate the effects of different payloads/state sizes to Shippy apps. Our scripted execution was performed in a section that lasted about 40 minutes and some devices had one or more connections to the server node through multiple tabs.

Table 1 summarizes the events observed in our evaluation. We had a minimum of 1 to a maximum of 8 devices connected at a given time, with 3 devices connected on average (sd ±1 devices). There were 29 server disconnections and sequential server recovery events as well as a total of 28 executed operations. The total number of client connections represents the number of times a device either connected to a server or reconnected to a new server after a failure.

For each of the events presented in the Table 1, we are interested in measuring the time between the occurrence of the event and the expected function calls in response to this event. Each logged event is a tuple that contains at least an event name, device unique identifier, and a timestamp representing the milliseconds elapsed since the `UNIX` epoch.

For example, whenever a running server disconnects, our instrumented app logs a `disconnecting` event which is followed by a successor device responding with `become_server_begin` event and either a `become_server_end` or `become_server_error` event after the function executed. These events are chronologically sorted, as depicted in Listing 8, and we compute the elapsed time between them.

```
(timestamp, event, deviceID, server)
(1512766249862, "disconnecting", 15127210, true),
(1512766265059, "become_server_begin", 15223107, false),
(1512766265669, "become_server_end", 15223107, true),
...
```

Listing 8: Tuples with logged events

Each device stored its own log using a local server and, after execution, all these files were grouped and processed to compute elapsed times. We chronologically sorted all collected events and filtered events of interest to compute our measurements. Results are discussed in the following subsections and we explicitly detail the events used to compute measurements for each subsection whenever necessary.

4.2 Server Recovery

We define server recovery as the amount of time it takes for Successorships to publish a new server after a server failure. In order to compute recovery time, we measured events related to disconnecting devices and server publication, as shown in Listing 8.

Figure 4 illustrates the cumulative empirical function for server recovery time. In most cases, server recovery is achieved within 15 seconds but there are edge cases where server recovery took almost 1 minute. It is worth noting that this recovery time takes into account the DNS-SD service discovery time, i.e. the time it takes for the service discovery protocol to detect that the last running service with a given name is not available anymore until a new server is published with the same name.

Despite recovering within 15 seconds in most cases, we acknowledge that our current recovery time should be improved for a productive version of Successorships. One problematic aspect of a slow recovery is that operations cannot be applied on the shared state when clients are disconnected. While there are options to mitigate this problem (e.g. a client-side local queue for operations and batched submission upon re-connecting), we acknowledge that faster recovery is necessary for truly fault-tolerant local Web apps. The performance of server recovery is mainly dictated by the implementation of service discovery in FlyWeb and we aim to research how this can be improved in the future. For example, we could use more a aggressive protocol for service discovery [5], or, the first successor could immediately start a background "failover server". We elaborate more on potential improvements in Section 5.

4.3 Client Connection

We define client connection as the amount of time it takes for a device to connect to a published server as a client. This takes into account not only the time it takes for a client to connect to the server, but also the time it takes for a client to own its copy of the server state. Events related to client connection were used for this measurement, namely `welcome` messages sent from the server in the handshake and the first `stateupdate` event received at the client.

Figure 5 presents the cumulative empirical function for client connection. On average, clients are fully connected to a published server in 11 seconds (± 16 seconds). The majority of the clients (80%) were able to establish a connection under 15 seconds, but we noticed some cases were clients took longer to connect. Inspecting these edge cases, we identified that these were mostly related with scenarios were the state was larger than 1MB. In other cases were this affirmation did not hold, we hypothesize that the mobile network connection was not stable, but we lack data to support this hypothesis.

In order to further investigate client connection properties, we analyzed whether the number of connected devices or the server state size may influence connection time. Figure 6 presents a boxplot chart for the connection time grouped by the number of successors. Despite the fact that all connected clients received a state update with the most recent `successors` list, we could not see how the

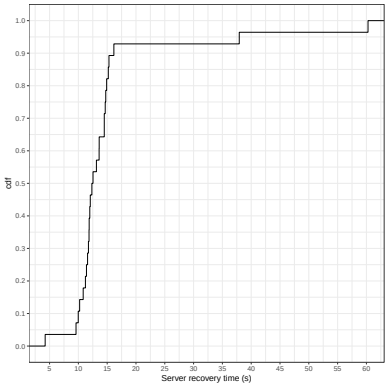


Figure 4: Recovery time after a server failure

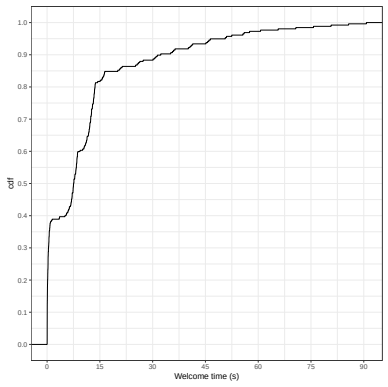


Figure 5: Client connection time until receiving a copy of the state

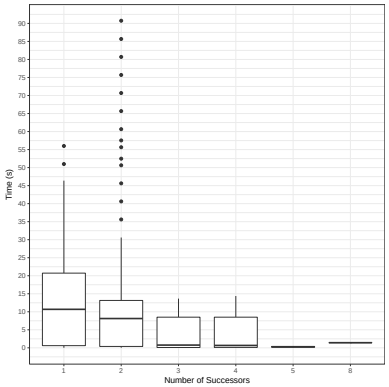


Figure 6: Client time to connect per number of successors

number of successors would influence a new client connection and all boxplot charts overlap.

On the other hand, Figure 7 models time as a linear function of the server state size. Intuitively, as the server state increases in size, clients take longer to connect. This give us insights about benefits and limitations of our current implementation. Based on client connection measurements, we posit that Shippy could scale with more connect devices, though the state variable is a bottleneck and applications that require a larger state are compromised.

4.4 Traffic

Finally, we evaluate network traffic in Shippy applications. We are interested in the round trip time (RTT) of messages as well as in how long it takes for client nodes to converge to a consistent state. In order to compute RTT, we measured events related to devices executing the `Shippy.call` operation until they received a broadcast with a state update from the server node. As each state update carries the most recent state version, we measure consistency as the time between the execution of `Shippy.call` and the last node updating to the most recent version. For instance, if a server `A` currently in version `V1` has 3 successors `[B, C, D]`, and `B` calls the `add` operation, RTT is the time it takes for `B` to receive a state update. On the other hand, consistency is longest time frame from the `add` call until `B`, `C`, or `D` updates to `V2`.

Figure 8 plots the cumulative empirical function for RTT while Figure 9 plots state convergence time. In most cases, RTT is considerably fast and clients receive a state update in the order of milliseconds. However, state convergence takes longer and in 90% of the cases client nodes converge upon a new state roughly under 5 seconds (average of 1.3 seconds, sd ±2.7 seconds). We also noticed that results from the empirical cumulative function are affected by 4 outliers that are greater than 1.6 seconds. We hypothesize once again that the mobile network was not stable during convergence for these cases.

Since state convergence takes longer than message RTT, we investigate how it is influenced by the number of connected nodes and payload sizes. We followed a similar strategy as for client connection, evaluating boxplots per number of successors and convergence time as a linear function of the operation's payload. For the sake of brevity we refrain from plotting charts for these scenarios. It suffices to say that we could not see bottlenecks for either the number of nodes or the payload size. For example, using a payload with a size equal to the one that slew down client connection (1MB), convergence time was $\approx 1.6s$ while client connection time would be $\approx 20s$. Therefore, we believe that Shippy applications should not suffer considerable traffic bottlenecks and provide good user experience.

5 LIMITATIONS AND FUTURE WORK

While we consider Successorships a good functional proof of concept we acknowledge a number of limitations that we would like to improve in the future.

Platform dependency and time-to-recovery. We used components of Mozilla FlyWeb for service publishing and discovery, acting as a mediator between Shippy and the Zeroconf protocols. FlyWeb is currently not maintained by Mozilla[19] and we relied on an archived version of the Firefox Developer Edition for our implementation. Furthermore, our implementation is currently only running on MacOS due to a known bug in the *mDNS* implementation in FlyWeb. While the respective ticket[20] claims the bug is fixed we were not able to obtain a version with the fix. Furthermore, the service discovery implementation in FlyWeb is considerably slow and both the removal of a disconnected service and the addition of a new service take several seconds. This is the root problem of a fairly disappointing time-to-recovery in section 4 and our highest priority for future improvements. Due to the aforementioned issues we have considered discarding FlyWeb and working on our own implementation of the Zeroconf protocols instead.

Security. We clearly accepted a few security concerns for the Successorships proof of concept but we are aware of their importance in a potential productive version of our framework. Firstly,

[19] We contacted the authors of FlyWeb and asked about the current status of the project. FlyWeb started as a side-project of two Mozilla developers and was then advertised intensely in 2016. The add-on was eventually integrated into the Firefox core. However, maintenance was discarded after a "shift of priorities" at Mozilla and the module was removed in a subsequent version of Firefox. We approached the authors with our idea of adding a fault-tolerance layer to FlyWeb and received a very encouraging response: the idea had been around among the authors and could have been one next step if the project had been continued.

[20] https://bugzilla.mozilla.org/show_bug.cgi?id=1294772, accessed 2017-12-10.

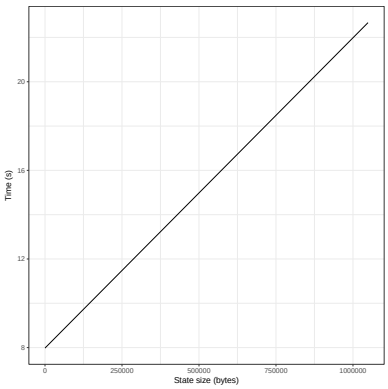


Figure 7: Client time to connect as a function of the server state size

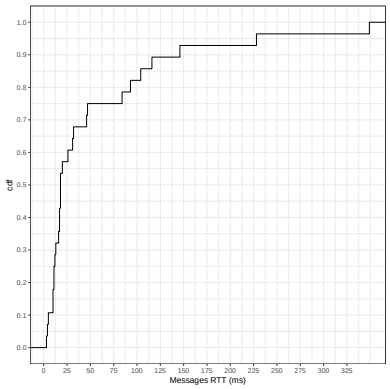


Figure 8: Operation Round Trip Time

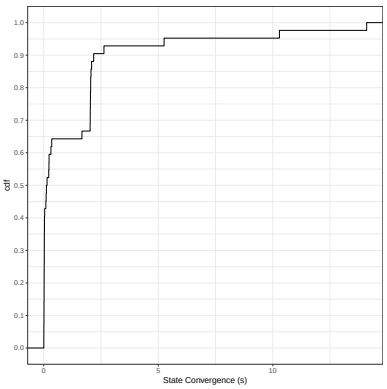


Figure 9: State convergence - time it takes for all clients to update their state

in certain failure cases where a client state is newer than a server state, clients can essentially *overwrite* the server state. This opens up the risk of malicious clients imposing a manipulated state onto our network. This could be tackled by a validation mechanism on the server and other means of establishing trust between participants. Secondly, our add-on component described in section 3.3 exposes *all* local FlyWeb services to the browser's global `window` object. This essentially means that any Web application running in the browser can read information on the FlyWeb services available in the user's local network. This can be solved in the future by only exposing the service for the currently running app in discovery events.

Consistency guarantees. As described in section 3.4 we built Successorships on the concepts of eventual consistency and lazy replication. This approach will tolerate inconsistencies between participants to some degree with the assumption of reaching consensus when the system "quiesces" for a period of time. We acknowledge that this model may not be sufficient for some applications. We have considered to notify the first client in the `successors` list and only broadcast updates to other clients when an acknowledgement was received. We faced technical difficulties with this strategy and restrained from an implementation of the idea. Since it is much more important that the first successor is up-to-date compared to other clients, we can still imagine a future implementation.

Furthermore, we still broadcast the whole state in certain scenarios. This could result in significant traffic on the WebSocket channels that may overwhelm the network for considerable sizes of the state and number of clients. We consider moving to a fully operation-based approach in the future.

A stronger model of eventual consistency than the one used by Shippy is offered by *conflict-free replicated data types* (CRDTs) [9]. As long as they receive the same updates, no matter the order, any two replicas using CRDTs are guaranteed to converge to the same state. We have considered using CRDTs for our implementation, but we believe that the constraint of commutativity of operations, or associativity of merging conflicting states, is potentially too restricting for the range of applications we would want to allow to run on our framework.

Finally, the global state is currently computed any time an operation is received. We consider treating the state as an ordered set of atomic operations such that the state can be re-computed any time by executing all operations from the set. This would improve the consistency guarantees we can provide and establish a basis for at least two more features we have in mind: the merging of multiple servers running in parallel and local operation queues on currently disconnected clients that are sent in batches to the server upon reconnecting.

Usability. We currently use IP/port based URLs for Shippy services. Besides the lack of usability of such URLs, there is another major disadvantage: in cases where the server role migrates to a different node, clients will establish a WebSocket connection to the new server node using the advertised IP-port URL, but the browser's address bar will still point to the old address. A reload in the browser will then result in a *not found* page because the old server is not available anymore. This problem may be solved by using persistent hostnames and local DNS resolution for Shippy apps but this would require further research.

Also, we currently do not provide a browser UI listing available Shippy services The respective FlyWeb menu works only unreliably and comes with the previously described platform dependency problem. We aim to provide a separate menu UI for supported browsers with a menu listing only Shippy services.

Server implementation. As described in section 3.3 our current server implementation re-fetches all static resources obtained from the original Web server and saves these resources in the browser's `sessionStorage`. Since the resources are already fetched when they are discovered in the loaded HTML file (e.g. index.html) by the browser, this second fetch is technically not required. We aim to analyze how these resources can be accessed without a second manual fetch.

Moreover, we did not evaluate server performance. Since any device can become a server with our implementation it is important to analyze performance and resource consumption of our approach on different devices and scales.

Who can publish a service? We did not differentiate between clients in terms of device types, user groups or other criteria. In our implementation, all participants are treated equal and any client can publish a service and become a server. We acknowledge that different devices and user roles can be very useful for Shippy apps.

For example, we might only want more powerful devices or certain users become servers. We can imagine bringing this aspect to Successorships in the future.

6 RELATED WORK

Zeroconf. Several researchers have investigated Zeroconf [1, 4, 6]. For instance, [5] discusses that Zeroconf service discovery may cause overhead to the network while discovering new services. Thus, they propose an algorithm to accelerate service discovery based on network topology changes. Since a server failure would imply a change in the network topology, i.e. the server node being removed from the topology, client nodes in our implementation could use their approach to accelerate service discovery. Nonetheless, their implementation uses Linux Wireless extensions, which may not be accessible within a browser. Thus, we used the FlyWeb service discovery implementation.

In [10], Stolikj et al. argue that the number of published services in a network may also slow down service discovery. As a solution, they propose a context-based approach, where queries specify which services they are interested in. This approach is highly suitable for our work and we could use it to filter Shippy services. However, their approach changes service discovery queries and we question whether this could be easily integrated with existing devices.

Local Networking APIs. Published in 2008, Universal Plug and Play (UPnP) is another widely-deployed set of networking protocols that facilitate discovery and interaction between devices on the same network, with minimal configuration. Since it leverages common protocols (HTTP/XML/SOAP on UDP/IP) and is agnostic to the link medium, it is truly cross-platform, extending not only to phones and laptops but also to printers, WiFi routers, and audio-visual equipment, to name a few examples. UPnP has been characterized as consisting of protocols that are more specific to particular classes of devices and applications; this is in contradistinction to Zeroconf, which aims to provide a device-agnostic foundation on which any device class or application-level protocol can build. [21]

More recently in 2017, as part of its *Nearby* project, *Google* released its *Connections* API, which enables Android devices in close proximity to one another to communicate in a peer-to-peer fashion. [22] This is done over a seamless mix of Bluetooth and WiFi hotspots. Unfortunately, Google has only made this API available on the Android platform; we seek a solution that is truly cross-platform.

In-browser web servers. An early example of fully in-browser web server technology was *Opera Unite* (2009)[23]. This extension to the Opera browser enabled users to serve general-purpose Web applications directly from their browsers. Communication between clients and servers in this way could either be via Opera Unite's proxy servers, which also offered a name registering and directory service, or direct peer-to-peer, the latter requiring more advanced technical configuration. Importantly, Opera Unite applications were constrained to be written as Opera "Widgets"[24]. Though the service was popular with a sizeable subset of Opera users, especially for purposes such as file sharing and media streaming, Opera eventually retired Unite in 2012 to consolidate the multiple extension frameworks offered in its browser.[25]

7 CONCLUSION

As networking capabilities become more ubiquitous across different types of devices, applications that communicate over local area networks are becoming more common in recent years. Using a proper suite of networking protocols and technologies, these applications can easily discover other devices in the network and exchange data with them. One appealing way to build such apps is using Zeroconf networks and Web browsers. Nonetheless, applications operating in Zeroconf still face the common challenges imposed by the type of network that they operate on, e.g. unstable wireless links and host mobility. Also, Web developers have little to no browser API support for these technologies.

To address the aforementioned issues, we proposed Successorships, a JavaScript framework for Zeroconf Web applications that abstracts and decouples networking logic from application logic and seamlessly integrates fault-tolerance to applications that require state but face failures due to intermittent connectivity. We evaluated Successorships by measuring network traffic and state recovery in the face of failures using proof-of-concept applications. Successorships supports all its intended use cases and future work will improve bottlenecks and performance problems identified during our evaluation. The first presentation we held for Successorships was promising indeed: it was itself the first live example of a Shippy application.

REFERENCES

[1] H. Bohnenkamp, P. Van der Stok, H. Hermanns, and F. Vaandrager. Cost-Optimization of the IPv4 Zeroconf Protocol. *Proceedings of the International Conference on Dependable Systems and Networks*, 00(c):531–540, 2003.

[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] M. Günes and J. Reibel. An IP Address Configuration Algorithm for Zeroconf . Mobile. *Discovery*, 2002.

[5] S. G. Hong, S. Srinivasan, and H. Schulzrinne. Accelerating service discovery in ad-hoc zero configuration networking. In *Global Telecommunications Conference, 2007. GLOBECOM'07. IEEE*, pages 961–965. IEEE, 2007.

[6] A. J. Jara, P. Martinez-Julia, and A. Skarmeta. Light-weight multicast DNS and DNS-SD (lmDNS-SD): IPv6-based resource and service discovery for the web of things. *Proceedings - 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2012*, (January 2016):731–738, 2012.

[7] 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.

[8] 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.

[9] 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.

[10] M. Stolikj, P. J. Cuijpers, J. J. Lukkien, and N. Buchina. Context based service discovery in unmanaged networks using mdns/dns-sd. In *Consumer Electronics (ICCE), 2016 IEEE International Conference on*, pages 163–165. IEEE, 2016.

[21] http://www.zeroconf.org/zeroconfandupnp.html

[22] https://developers.google.com/nearby/connections/overview, accessed 2017-12-11

[23] http://help.opera.com/Windows/12.10/en/unite.html,accessed2017-12-11

[24] An Opera Widget is an application that runs on Opera's now-deprecated Widget Engine, which allows such apps to be run independently of the browser.

[25] http://www.instantfundas.com/2012/04/opera-to-discontinue-unite-widgets-and.html