Zeroties: a Publish/Subscribe Service for Applications in Local Ad-Hoc Networks

Chris Satterfield Felix Grund
University of British Columbia
Vancouver, BC

ABSTRACT

1 INTRODUCTION

Throughout the last decade we have seen the Internet become the most commonly used infrastructure for communication: regardless of physical location, people and their devices communicate via messengers and VoIP, and collaborate via live editing tools like, for example, Google Docs, Sheets, and, Slides. At the same time, we have seen some applications communicating over local area networks become increasingly common for certain scenarios like home entertainment (e.g. Google Chromecast, Apple Bonjour, Spotify Connect) or Wi-Fi printers. With the advancements of "smart" devices and the "Internet of Things" (IoT) it is likely that this trend will grow beyond these currently still narrowly scoped application domains.

The adoption of standards that eliminate the burden of manual configuration of network devices further contributes to this movement. One such standard that has received widespread usage is Zero-configuration networking [?] and its protocols mDNS [?] for service advertisement and DNS-SD for service discovery [?]. Using the *Zeroconf* protocols, devices can publish named services in the local network and discover such services automatically in an ad-hoc fashion. While most applications for Zeroconf networks are shipped with specific hardware (e.g. Google Chromecast dongle) there have recently been attempts to provide software environments for developers to enable them write their own applications on already existing hardware infrastructure. One such application was Mozilla FlyWeb[1], an addon for the Firefox browser that made it possible to advertise and discover services from within Web applications through a JavaScript API. In a previous project, we created *Successorships* [2], a JavaScript library exposing an easy-to-use API to build fault-tolerant Zeroconf web applications. Successorships was built on top of Mozilla Flyweb as one main part of its architecture. This decision confronted us with significant problems:

- FlyWeb had been declared abandoned by Mozilla even before we finished our work on our library.
- The implementation of the Zeroconf protocols in FlyWeb was slow to a degree that made our library fairly unusable in practice.
- FlyWeb contained a bug that made our library usable on MacOS only.

To overcome our troubles in Successorships, we introduce Zeroties, a platform-independent asynchronous publish/subscribe service for Zeroconf advertisement and discovery. We carefully reviewed different publish/subscribe designs [?] and implemented a communication scheme based on *asynchronous notifications*. The operations exposed by Zeroties are as follows:

- **Publish**: publish a Zeroties service and advertise it in the local network
- **Subscribe**: listen for updates on the list of available Zeroties services

Zeroties ships with two components: (1) a standalone OS-level application, and (2) addons for Chrome and Firefox that connect to this application. With our addon implementations for Chrome and Firefox we aim to show that our approach translates well between different browsers and does not share the restrictions of FlyWeb. Our browser addons expose an API to web applications that comprises the full service/discovery functionality of Zeroconf. As a result, we have successfully eliminated the ties that prevented Successorships from usage in practice.

To evaluate Zeroties, we first created the webapp-based presentation for this project using Successorships in combination with the Zeroties daemon and the addon for Google Chrome. Running the presentation on Chrome proved that we successfully broke ties with FlyWeb and Mozilla Firefox. Furthermore, we could see in this example application that recovery from server failures was significantly faster than with the previous version based on FlyWeb. To evaluate these findings empirically, we repeated the performance measurements from the Successorships project, both with the previous version based on FlyWeb and the new version based on Zeroties. We simulated XXX server failures and subsequent recoveries in a small mobile network and found that the system recovered in average about three times as fast with Zeroties than with FlyWeb.

In summary, we make the following contributions:

- Zeroties, a asynchronous standalone publish/subscribe service for Zeroconf applications.
- Addons for Chrome and Firefox that make this service available to web applications.
- An empirical evaluation based on a Zeroties sample application indicating significant performance improvements.

The remainder of this paper is organized as follows: Section 2 provides some background and motivation on why the idea for Zeroties came to be. Section 3 presents the system model and design goals of Zeroties, before Section 4 describes its implementation. We evaluate Zeroties in Section 5 and suggest limitations and future work in Section 6. Section 7 situates our work in the context of related research and Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

We have been compelled by the idea to make the Zeroconf protocols available to web applications. For example, consider the scenario of a group of people, each with their own device, collaborating on a Google Doc document. Every keystroke in this document will need

[1]https://wiki.mozilla.org/FlyWeb
[2]https://github.com/ataraxie/successorships

to be sent to a Google web server and then be “pushed” too all other group members. Depending on geological location, this pattern of communication can impose roundtrips around the globe and lead to significant delays, despite the physical proximity of the devices. The fact that not everybody around the globe is equipped with the newest network infrastructure makes this even more proplematic.

Zeroconf web applications provide a solution to this problem. In the described Google Doc scenario, one person in the group would access the the Google Doc on the Internet and subsequently become a server in the local network. Her device would then serve the resources it fetched from the actual web server to local clients who maintain a channel to this local server. Effectively, only one Internet connection is required for the whole group, rather than one connection for every device[3], because all clients except the locally serving client access the original application (in this case Google Docs). Figure 1 illustrates this shift in architecture.

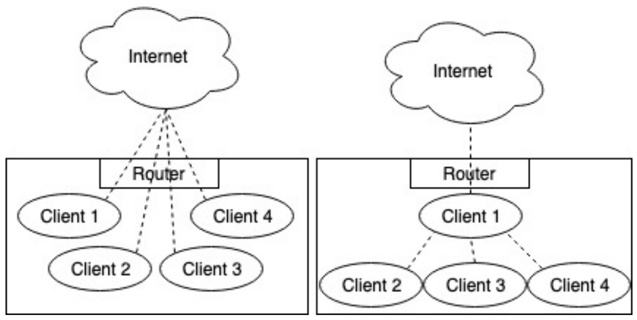

Figure 1: Architecture shift: on the left side is the traditional web application architecture where all devices are connected to a web server through the Internet. On the right is the Zeroconf architecture where only one client requires such a connection and becomes a server to all other clients in the network.

Evidently, this pattern of communication has one major problem: it has a single point of failure, namely the client acting as a local server. Solving this problem was our motivation behind Successorships; whenever the currently serving client failed, a new client in the network was elected as the new server and connectivity in the network was re-established automatically. Successorship thus provided a framework to build fault-tolerant Zeroconf web applications with an easy-to-use JavaScript API. However, our decision to built it on Mozilla FlyWeb proved to be a major limitation. Not only did we limit usage of our library to one specific browser vendor; we found out during the course of the project that Mozilla had already deprecated FlyWeb in favor of other priorities. To make things worse, FlyWeb had a dependency to a Firefox core component (the module responsible for launching a web server within the browser) that was only shipped in a range of versions of the developer edition of Firefox. In addition to this platform dependency that would, in essence, prevent our framework to be used in practice, the implementation of the Zeroconf protols in FlyWeb was incredibly slow. According to our evaluation, recovery from failure of the currently serving client took more than 30 seconds in many cases. With a continuously growing ’Limitations and Future Work’ section, we were finally struck by the discovery of an unresolved issue in FlyWeb that made our framework only work on MacOS. Despite our thoughtful and eager motivations, we had built a tool that nobody would ever use in practice. Hence the call to get rid of this dependency entirely and provide our own layer below Successorships.

3 APPROACH

3.1 Environment and System Architecture

Figure 2 illustrates the environment, placement of Zeroties, and the communication between components. We describe the parts in the figure refering to the underscored numbers as follows.

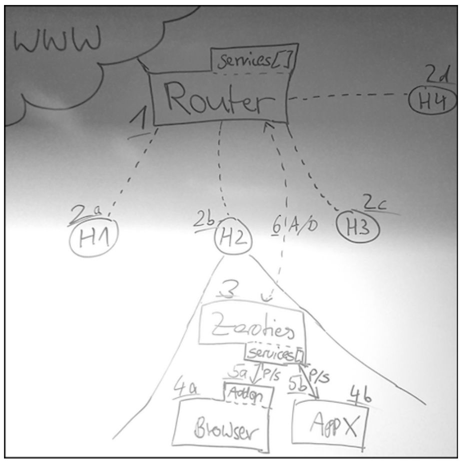

Figure 2: System architecture and environment in Zeroties.

(1) **Router** [4]: we use the protocols provided by Zeroconf ([**? ?**]) for Zeroties. In essence, this means that an authoritative list of currently available Zeroties services (as a subset of all Zeroconf services) can be obtained from the router at any given time.
(2) **Hosts**: there can be an arbitrary number of hosts in our local network. A host can be any device in the network with an instance of Zeroties running (i.e. any device with an OS capable of Zeroconf/DNS-SD protocols).
(3) **Zeroties Daemon**: the instance of Zeroties running on this device. This is an OS-level application maintaining a list of services.
This list is a mirror of the service list obtained from the router by means of the Zeroconf protocols. The list of services

[3]Obviously, this also has the advantage that only one device needs to have Internet connection in the first place. This is can be very interesting especially in scenarios where only dedicated participants in a network have such a connection, for example due to security restrictions.

[4]Note that we use the term router somewhat casually in this context. Our notion of router is basically any part in the local network that serves the list of available Zeroconf services, as it can be obtained with DNS-SD. This may involve DHCP, IPv6 Router Advertisement Options or other mechanisms as specified on page 27f of [**?**]. In other words, we simply call the component that stores the service list obtained from a DNS-SD-capable router.

constitutes the state of the distributed system [5] We originally intended to integrate this application into browser addons, but were restricted by browser policies. In particular, there is no possibility to start a web server or publish a Zeroconf service from within a browser addon. Hence our alternative solution of an OS-level app that communicates with our browser addons through stable bi-directional message channels.

(4) **Browser Addons**: we require browser addons that mediate between web applications and the Zeroties daemon. These addons expose a JavaScript API to apps built on top of Zeroties. We have implemented addons for Google Chrome and Mozilla Firefox that constitute such a mediator and we have run Successorships apps with these addons (see Section 5). These addons use Zeroties as a publish/subscribe service. Note that the Zeroties daemon itself is not restricted to these addons; they are only required for web applications running in browsers. Different applications can utilize the services provided by the Zeroties daemon without these components.

(5) **Publish/Subscribe operations**: applications built on top of Zeroties interact with the Zeroties daemon using the common publish/subscribe operations. *Publish* is the publishing of a Zeroties service and *subscribe* makes the application receive changes to the list of available Zeroties services.

(6) **Advertisements and service discovery**: the Zeroties daemon maintains a list of Zeroties services that is obtained from the router. In a sense, Zeroties acts as a middleware between the router and applications built on top of Zeroties (applications using Successorships as framework being one example). Whenever a Zeroties application publishes a service using the Zeroties publish/subscribe API, this will result in the publishing of a service in the network (service advertisement). The second part of this communication channel is service discovery. We describe both service advertisement and service discovery in more detail in Section **??**

3.2 Design Decisions and Goals

Zeroties is based on system models as in the example described in Section 2: local ad-hoc network applications. We claim that there is a dedicated set of such applications, for example

- A project presentation at a meetup where the audience can connect to the presentation and interact with it.
- An application for printer control in an office
- An application for the heating system of a hotel
- Multiplayer mode for browser-based games

Such applications have in common a limited number of nodes (generally < 100) and comparably lax requirements for time-to-recovery from system failures: certain downtimes can be tolerated as long as a consistent state is reached eventually. For example, a running application converging to a consistent state within a time frame of multiple seconds after failures is acceptable. Our particular goals behind Zeroties are described in the following paragraphs.

Publish/subscribe strategy. We reviewed the different publish/subscribe strategies described in [**?**] and decided to focus on an asynchronous invocation/callback style communication pattern as described in [**?**]§3.3. Asynchrony is essential for the communication between the Zeroties daemon and its applications. For example, applications built on Zeroties should not be blocked while waiting for the successful advertisement of a service, but rather be perform this action in the background while the application remains responsive in the foreground.

Consistency guarantees. The shared state in Zeroties systems is defined by (1) the list of currently available Zeroties services, and (2) the application-defined state that is shared between hosts in the network. If an instance is not up-to-date with this state, this can have different consequences. First, as long as applications are not notified that a new service has been created, that service remains unavailable. Second, a service that has failed but remains in the list can result in an error upon connecting to that service. Third, if notifications about updates of the application-defined state are delayed between hosts, this can have significant consequences. However, given our described system model of local ad-hoc network applications, we postulate that it is sufficient for most use cases if hosts converge to a consistent state *eventually* and therefore decided for *optimistic replication* [**?**]. The paper defines *eventual consistency* as "a weak guarantee [that] is enough for many optimistic replication applications, but some systems provide stronger guarantees, e.g., that a replica's state is never more than one hour old". Due to our requirements for an asynchronous publish/subscribe pattern and our awarenes that, according to the CAP theorem [**?**] we cannot achieve both high consistency and availability, our work can be considered as trading consistency in favor of high availability. Nevertheless, we consider the time frames for reaching consensus in our previous project (> 30 seconds in many cases) as insufficient and we want our new system to do so within a time frame of *10 seconds*.

Fault tolerance: In the case of Zeroties, fault tolerance is directly connected to consistency guarantees. Take Successorships as an example for a Zeroties application and assume the scenario of a failing server [6]. The service that was offered by the failed server will be removed from the list of available Zeroties services and the change will be propagated to Zeroties applications that poll information from the router (see Section 3.1). The application can the decide how to deal with this change. In the Successorships example, this will involve the selection of a new server (and therefore a new Zeroties service) and other clients connecting to it. In that sense, the system will have recovered from the failure of the server (and therefore have converged to a consistent state), as soon as (1) the failed service was removed from the list, (2) the newly elected service was added to the list, and (3) all changes propagated to all clients in the system. Consequently, Zeroties enables its applications to recover from failures in the same time frame as the system reaches consensus.

Performance: As described earlier, the performance of our previous implementation of Zeroconf based on FlyWeb showed significant bottlenecks and proved insufficient for practical usage. With Zeroties we had a few concrete goals in mind regarding time frames. First, the *publishing of services* and *notifications about*

[5]Note that there is a clear line between Successorships and Zeroties: Zeroties does not know about the arbitrarily complex state of Successorships or any other app that builds on Zeroties.

[6]In the Successorships example, by *server* we refer to the *the currently serving client*. Remember that in our world any client of the web application can become a server.

new services from the perspective of a Zeroties application should take *less than 10 seconds. Communication between Zeroties and its applications* should be in the range of *milliseconds.*

Reliability: Our system will remain highly reliable as long as our DNS-SD communication scheme between Zeroties and the router is robust and the communication channels between Zeroties and its applications is stable.

Polling strategy: We decided for a polling strategy for obtaining the list of services from the network router using DNS-SD. The list of available services is updated based on changes detected between the last version of the services list in the Zeroties daemon and the most recently polled services list. Clearly, this polling strategy will impose traffic on the local network, with all Zeroties hosts polling the router for the services list. However, we argue that frequent changes in the available services list and the assumed low number of nodes in our system model justifies this approach. A different strategy which could allow for a larger number of nodes without the potential problem of overloading the router with requests would be to implement this communication with a dedicated Zeroties leader and followers, for example using Paxos [**?**] or Raft [**?**]. However, this would mostly be a transformation of communication between Zeroties hosts and the router towards communication between Zeroties hosts. More importantly, we would waive the advantage of the direct use of the authoritative list of services maintained by the network. Nevertheless, we are aware that a design with the leader-and-followers approach, despite adding significant complexity, would have advantages as soon as Zeroties networks reach a certain size.

4 IMPLEMENTATION

4.1 Languages and Platforms

All components are written in JavaScript. The Zeroties daemon is written using Node.js[7], the browser addons are written with the vendor-specific addon frameworks. We decided for JavaScript for all components due to several reasons:

- We wanted the OS-level Zeroties daemon be as platform-independent as possible. Node.js is a stable environment on all major operating systems by now.
- We wanted to lean on a library implementing the low-level Zeroconf protocols. We found a capable, mature, and well-maintained library, *dnssd*[8] available for this purpose on Node.js.
- Browser addons must be implemented in JavaScript. Despite the fact that we could not integrate the core Zeroties functionality into our browser addons (see Section 3) we still wanted to avoid introducing a language barrier that would prevent us to do so even without browser-specific restrictions.
- Starting a HTTP server and a WebSocket server in Node.js and connecting to these servers from an addon is inherently straight-forward. This functionality was required for our purpose.

[7]https://nodejs.org/en/, accessed 2019-04-17

[8]https://www.npmjs.com/package/dnssd, accessed 2019-04-17

4.2 Zeroties Daemon

At the core of Zeroties is the Zeroties daemon. This is the piece of software that any application wishing to make use of the Zeroties services must connect to. The daemon runs continuously in the background on the users OS, operating a server to handle interactions with Zeroties applications. Zeroties applications are provided the following two pieces of core functionality by interacting with the Zeroties daemon:

- The ability to publish services on the local network, via mDNS.
- Notifications when the list of services available on the local network changes, as discovered via DNSSD.

The Zeroties daemon is also responsible for managing any client HTTP requests or WebSocket connections to the server published by a Zeroties application. Instead of handling these events directly, the daemon simply acts as a proxy, forwarding the events to the Zeroties app as approriate. This allows developers of Zeroties applications to easily define custom routing for HTTP requests recieved, and define custom actions to handle different WebSocket events.

All communication between Zeroties applications and the Zeroties daemon is made via a WebSocket connection, following the protocol outlined below. On connecting to the daemon, applications are automatically subscribed to recieve notifications about changes to the list of Zeroties services available. The daemon repeatedly polls the network to retrieve the current list of Zeroties services using the DNSSD protocol. If the list of services retrieved from the network differs from the one currently held by the daemon, an event is triggered. This event is fired over the WebSocket connection to a listening Zeroties application. To publish a server, a Zeroties application sends a message to the daemon containing the name of the service to publish. The Zeroties daemon then spins up an HTTP server and a WebSocket server listening on the same port, and advertises the newly available service over the local network via mDNS.

4.2.1 *Zeroties Communication Protocol.*

This section will outline the protocol that must be followed for communcation with the Zeroties daemon.

Messages to the Zeroties Daemon. At the moment, there is only one supported message that may be sent to the Zeroties Daemon unprompted: the "publish" message. The message should send a stringified JSON object of the following format:

```
{method: "publish", payload: {name: appName}}
```

This is the message that informs the Zeroties daemon that you wish to publish a server.

Messages from the Zeroties Daemon. Any app connected to the Zeroties daemon should expect to recieve and handle a message in the following format:

```
{method: "servicesChanged", services: <array of services>}
```

This is the message sent by the Zeroties daemon whenever a change in the list of Zeroties services is detected.

In addition, Table 1 outlines the messages any Zeroties app which publishes a server should expect to see, as well as the expected responses to these messages.

Method	Response	Response to response	Purpose
"request"	"response"		Dispatches an incoming HTTP request to the Zeroties App for handling. Waits for a response containin
"wsForward"	"wsForwardResponse"	"wsProxyHandshake"	Notifies the Zeroties app of an incoming WebSocket connection and establishes a proxy connection
"wsForward"	"init"		To complete the proxy connection, the Zeroties app must send an init message over the proxy connectio

Table 1: The messages which are sent to Zeroties apps by the Zeroties daemon, and the responses which are expected back. After recieving a wsForwardResponse, the daemon will respond again with a "wsProxyHandshake" message.

4.3 Browser Addons

The Zeroties browser add-on is an intermediary layer that easily allows webapps to function as Zeroties apps. It exposes a simple API through the browser that abstracts away the underlying complexities of proxy communication with the Zeroties server. On the backend, the addon handles the establishment of proxy connections between a webapp and the Zeroties daemon, facilitating bidirectional communication between server webapps and their clients. The addon works by injecting the API script into all accessed webpages using a content script.

4.3.1 API. The Zeroties addon API consists of two functions:

- `navigator.publishServer(appName)`
  sends a message to the Zeroties daemon, which then attempts to publish a server with the specified name. It returns a Promise, which resolves to a server object. From this server object, behaviour for handling HTTP requests and WebSockets can be defined.
- `window.addEventListener("zerotiesServicesChanged", callback)`
  listens for the firing of the "zerotiesServicesChanged" event. This event is triggered by the Zeroties daemon whenever the list of Zeroties services available has changed. The second parameter is a callback function which recieves the updated services list as its argument.

Together, these APIs provide the full functionality of Zeroties to apps in the browser. This API roughly mimicks the one that is provided by FlyWeb, so any webapp supported by FlyWeb will also work with Zeroties with some minimal changes, and vice versa.

4.4 Communication and Control Flow in Zeroties

```
navigator.publishServer(Shippy.internal.appName()).
    then(function(server) {
    // ...
    server.onfetch = onFetch;
    server.onwebsocket = onWebsocket;
    server.onclose = onClose;
    // ...
}).catch(function (err) {
    // ...
});
```

Figure 3

In this section we will discuss the flow of control and communication between Zeroties apps and the Zeroties daemon, using our Successorships app and the Zeroties addon as an example. Figure 3 shows an example of the use of the publishServer call. onFetch, onWebsocket, and onClose are handler functions that are called when their respective events are triggered. When this function is called, in the background a WebSocket connection is established with the Zeroties daemon. This WebSocket serves as the main channel of communication between the daemon and the addon, which interacts directly with the server webapp. Whenever a client triggers an event on the published server, for example upon recieving an HTTP request, it is forwarded to the addon via a WebSocket message. In the case of an HTTP request, the onfetch property of the server is then called, which generates a response to the request. The response is then serialized and sent back over the WebSocket as the payload of another message. Finally, the response is deserialized by the Zeroties daemon and dispatched back to the appropriate client. To make sure responses are sent back to the correct client, all messages in this series of events are accompanied by a unique identifier that is generated when the HTTP request is first recieved.

WebSocket connection events follow a similar control pattern. When the published server recives a WebSocket connection, the Zeroties daemon notifies the addon of the new connection by a WebSocket message. The addon then creates a new WebSocket connection with the daemon. This means that for each client connected to the server, there is a proxy WebSocket between the daemon and the addon. This facilitates pseudo-direct communication between the server webapp and client. WebSocket messages are sent from the client to the published server, and then forwarded by the daemon over the proxy connection to the server webapp. The reverse is true of messages sent from the server webapp to the client.

```
window.addEventListener('zerotiesServicesChanged',
    function (event) {
     // ...
 });
```

Figure 4

Figure 4 shows an example of subscribing to changes in the Zeroties services list. This is a much more straightforward process on the backend than publishing. The Zeroties daemon simply polls the network for changes in the services list using DNSSD. When a change is detected, the daemon sends a message over the WebSocket connection to the addon. This message contains the updated service list. The addon then causes the

`window`

to emit a 'zerotiesServicesChanged' event, with the updated services list as its payload.

5 EVALUATION

5.1 Sample Successorships Application

In our previous project, we created a web application using the presentation framework *Reveal.js* [9] to present Successorships to

[9]https://github.com/hakimel/reveal.js/, accessed 2019-04-17

graduate students of the computer science department of the University of British Columbia. The application used the Successorships API that made use of the Zeroconf protocols based on FlyWeb. Obviously, the presenters had to have a Mac computer and a dedicated old version of Firefox Developer Edition with the FlyWeb addon installed to be able to present. Moreover, the previously mentioned performance problems were clearly visible, with recovery of the system from intentionally injected faults taking over 20 seconds at times. In a similar fashion, we presented Zeroties to an audience of graduate students at the same university. The presentation technicalities remained the same, except that the API usage within Successorships was changed from FlyWeb to Zeroties. To demonstrate this new independence of Firefox, the presentation was performed on Google Chrome. A significant decrease in time-to-recovery could also clearly be observed.

5.2 Empirical Evaluation

6 LIMITATIONS AND FUTURE WORK

7 RELATED WORK

Publish/Subscribe is a widespread messaging pattern that offers a rendezvous style channel for senders (publishers) and receivers (subscribers). Using this pattern, participants do not need to be aware of what other specific participants are in the system and no direct communication channels between sender and receiver are required. This provides better scalability and more flexible network topology. The publish/subscribe pattern has been elaborated largely in both the distributed systems [**? ? ?**] and software engineering [**? ? ?**] communities.

Zeroconf has been investigated by several researchers both in terms of how the protocols can be improved [**? ? ?**] and what problems they impose on networks [**? ?**]. The most widely available implementation of Zeroconf is Apple's *Bonjour* [10] software that is also available for Windows and Linux. There are also Linux- [11] and similar Windows-specific [12] implementations. These implementations are widely used by different applications to communicate in ad-hoc networks, including home entertainment (e.g. Google Chromecast [13], Amazon Fire TV [14]), desktop applications (e.g. Adobe Photoshop [15], iTunes [16]).

Zeroties. In contrast to these platform-specific implementations and applications of Zeroconf, we are interested in providing a framework for web application developers to use the compelling ad-hoc networking characteristics of Zeroconf. Our approach provides this framework on top of the publish/subscribe pattern makes the compelling Zeroconf properties available to a wide variety of web applications in an efficient manner. To the best of our knowledge, Mozilla FlyWeb [17] has been the only attempt that shared our motivation of Zeroconf in-browser web servers. Unfortunately it had major problems (see Section 2) and has been abandoned by Mozilla. Zeroties not only extracts the functionality from Mozilla FlyWeb but also abstracts the implementation away from platform-specific details and fixes a range of issues in Mozilla's now-abandoned addon. In particular, it extends the compatibility from Firefox-only to Firefox and Chrome [18], and improves performance of Zeroconf communication at least by a factor of two.

8 CONCLUSION

[10]https://support.apple.com/en-ca/bonjour, accessed 2019-04-17
[11]https://www.avahi.org/, accessed 2019-04-17
[12]http://techgenix.com/overview-link-local-multicast-name-resolution/, accessed 2019-04-17
[13]https://store.google.com/product/chromecast, accessed 2019-04-17
[14]https://amazonfiretv.blog, accessed 2019-04-17
[15]https://www.adobe.com/products/photoshop.html, accessed 2019-04-17
[16]https://www.apple.com/ca/itunes/, accessed 2019-04-17
[17]https://wiki.mozilla.org/FlyWeb, accessed 2019-04-17

[18]Zeroties is also easily adaptable for other browsers' addon ecosystems.