Extending the web to support personal network services

John Lyle
Department of Computer Science, University of Oxford
Oxford, UK
john.lyle@cs.ox.ac.uk

Claes Nilsson and Anders Isberg
Sony Mobile
Lund, Sweden
Claes1.Nilsson@sonymobile.com
Anders.Isberg@sonymobile.com

Shamal Faily
Department of Computer Science, University of Oxford
Oxford, UK
shamal.faily@cs.ox.ac.uk

ABSTRACT

Web browsers are able to access resources hosted anywhere in the world, yet content and features on personal devices remain largely inaccessible. Because of routing, addressing and security issues, web applications are unable to use local sensors, cameras and nearby network devices without resorting to proprietary extensions. Several projects have attempted to overcome these limitations yet none provide a full solution which embraces existing web concepts and scales across multiple devices. This paper describes an improved approach based on a combination of Web Intents for discovery, a custom local naming system and routing provided by the *webinos* framework. We show that it can be applied to existing services and that improves upon the state of the art in privacy, consistency and flexibility.

Categories and Subject Descriptors

D2.11 [**Software Engineering**]: Software Architectures

General Terms

Architecture, Standards, Design, Security

Keywords

Browser, Web, Intents, Personal Network, APIs

1. INTRODUCTION

Mobile web applications will only compete with their native counterparts when they have access to the same capabilities. While native applications can access sensors, cameras and NFC readers, the slow introduction of equivalent browser APIs means that web applications remain second-class citizens. Despite this, the web offers interoperability and compatibility advantages that many argue make browsers the application platform of the future [12]. As a result, several initiatives exist to create new browser APIs or develop new web application environments [21, 17, 13, 1].

SAC'13 March 18-22, 2013, Coimbra, Portugal.

At the same time, the increasing number of personal devices creates a need for web applications to access private network resources *as well* as device features [3]. Unfortunately, browsers have generally failed to penetrate home networks. This is partly due to difficulties routing behind firewalls but also because there is no standard way of *discovering* or using local network resources. This is even harder for devices on mobile networks with frequently varying addresses. However, this situation is changing: there are many use-cases involving multi-device interaction which could be satisfied by web applications. For example, games which use both the large display on a television as well as the motion sensors on a smartphone [9]. This is one of the motivations behind the *webinos* project [22].

While attempts have been made to provide access to local and network features from web browsers, all have limitations. Many introduce new browser APIs [17, 21], which are convenient but do not extend beyond the local device. JavaScript APIs also require every browser to implement for compatibility. Other approaches successfully provide access to resources within a local area network but fail to route between different networks and do not provide a way for web applications to discover services [9].

In this paper we present an approach which combines the best aspects of existing systems without breaking the abstractions expected by web browsers and applications. Our solution places a server on each device, introduces a new set of local domain names and uses the *webinos* system to create a virtual personal network accessible from the web browser. This network can then be accessed using either HTTP or WebSockets in order to use device features. Each server can be secured using a combination of existing techniques and simple rules based on web *origins*. The emerging Web Intents standard is proposed to allow services to be anonymously connected to web applications.

The paper is structured as follows: in section 2 we provide background material on web applications. In section 3 we describe existing systems and, based on their limitations, in section 4 we define requirements for a successful system. In section 5 and 6 we outline our proposal and implementation before evaluating against our requirements in section 7. In section 8 we conclude.

2. BACKGROUND

2.1 Web applications and browser security

Web applications are 'a web page or collection of web pages delivered over HTTP which use server or client-side

processing to produce an application-like experience' [14].

Web browsers isolate content by *origin* – the DNS name, port and scheme they are served with – such that JavaScript belonging to one origin cannot access any other, with some exceptions [25]. This is known as the *same-origin policy*. When cross-origin communication is desired, several mechanisms exist to support it. As well as new APIs discussed in the next section, web servers can use the Cross Origin Resource Sharing (CORS) protocol to indicate which other origins may access the resource being served [18].

Browsers *sandbox* web applications to limit their access to the underlying device. However, several new browser APIs have been proposed to let web application access local resources, such as orientation sensors, address books and geolocation [19].

2.2 Cross-origin browser communication

The *WebSocket* API and protocol allows web applications to establish two-way communication with a remote host [20]. WebSockets overcome limitations with HTTP that make it difficult for client web pages to receive messages from the server without frequent polling.

Channel messaging enables direct communication between web applications from different origins running in different browser contexts (such as different frames) [24]. Connections are created using a *MessageChannel* object and messages are received and posted using *MessagePorts*.

The *Web Intents* specifications let a web application request an abstract action be performed — e.g., *share*, or *view* a resource — and let services register their *intention* to handle such actions [23]. The browser acts as a broker, anonymously connecting the client web application to a compatible service, with a user consent and selection step. Web Intent services register to handle an intent based on the action name and a 'type' field which allows the browser to differentiate the request to *view*, say, a Word document from an image or video.

2.3 Personal and home networks

Home and *personal* networks provide interoperability between multiple personal computers. Perhaps the most significant personal network technology is UPnP [16], a set of protocols which allows home network devices to discover each other and share resources. UPnP is widely supported but offers no standard API for applications and is not sufficient for accessing resources which are behind firewalls or on mobile networks. Another relevant technology is Multicast DNS (mDNS) [8], which allows individual devices on a local network to register a link-local host name of the form `devicename.local.` without a central name server.

3. EXISTING APPROACHES

There are several proposals for how web applications may gain access to both device resources and local networks. We focus on two common approaches: the use of a privileged web server on each device and the extension of browsers through new JavaScript APIs.

3.1 Device servers

Lin et al. [9] propose *Gibraltar*, as a system where each device has a web server running at *localhost* capable of accessing native resources. Other web pages can access the server via standard HTTP, CORS and AJAX. The authors demonstrate that this is adequate for several use cases. They also propose additional security controls with the aim of reducing the trust in the browser.

Similarly, the now-defunct Opera Unite [15] bundled a web server with the browser, enabling personal devices to connect to one-another via an online proxy. Devices were exposed through a URI pattern `http://device.user.operaunite.com` and Opera Unite applications were made available on pages hosted at this domain. APIs were provided to access file systems and other resources.

However, Gibraltar does not provide a solution to routing or discovery. It is not clear how a web application would discover nearby devices or access those on different networks. This is a reasonable omission, as the problem is orthogonal, but presents an opportunity for improvement. In Opera Unite, users were required to have a MyOpera account and use the Opera browser, limiting adoption. Furthermore, it was based on a DNS system managed by Opera and was difficult for users to create their own domains. Most use-cases involved personal file servers and widgets, and it is difficult to tell whether it was suitable for accessing or discovering other services.

3.2 Browser APIs

Web applications could gain access to local device resources through new JavaScript APIs in browsers. As well as the widely supported Geolocation API [19], various initiatives [17, 21, 13] have proposed APIs for calendars, media capture, battery status and more. These APIs have the advantage of integration with the browser, allowing custom security and privacy controls to be applied. Browser APIs also arguably provide an easier abstraction for developers compared with HTTP and AJAX.

There are several drawbacks. It is not clear how APIs extend to multiple devices. We argue that this is because device APIs are not aligned with the rest of the web: concepts such as hosts, domains and request/response protocols already exist and it would be conceptually cleaner to reuse them. Another issue is privacy: many browsers APIs and extensions enable user fingerprinting [5]. The combination of browser names, versions and capabilities make it possible for websites to re-identify an individual without requiring an explicit user log-in process. The more APIs implemented by browsers, the worse this may become.

Moreover, implementations may vary by browser vendor and version, particularly in how access control is managed. Browser APIs require a completely new access control framework for user consent, with which web applications must be able to interact. If existing protocols and concepts were re-used to provide access to device features, it seems more likely that a standard, interoperable solution would exist.

3.3 webinos

The *webinos* project is a hybrid: it provides new JavaScript APIs which contact a device server to access local resources. Its implementation is similar to Gibraltar but uses WebSockets. Like Opera Unite, the server can contact other devices via an online hub. However, it can also use other bearers and local networks. The *webinos* system supports several device APIs as well as a service discovery API, allowing devices to be dynamically queried for available services. *webinos* uses an XACML-based policy system for access control [10].

The main disadvantage of *webinos* is the introduction of

a new, proprietary discovery protocol and device addressing scheme. This abandons existing web abstractions. It also makes fingerprinting worse, as the set of discovered services may allow re-identification of users. The access control system also diverges from web concepts of origins and sandboxes, although it does provide more potential for interoperability than current browser access control.

3.4 Other approaches

Ford et al. [6] propose the 'Unmanaged Internet Architecture' (UIA) which implements a domain name system for personal devices with no central authorities. Names are independent of location and automated routing between devices in the same namespace is possible. UIA solves many routing and addressing issues but does not attempt to solve problems surrounding discovery of services, or how to integrate with web applications. UIA is complementary our work and provides a generalisation of the *webinos* routing approach.

IPv6 may help with addressing entities on different networks but does not help with discovery, nor does it help connect devices that exist behind firewalls.

Finally, a proposed addendum to Web Intents [3] allows web applications to access local network services exposed via mDNS, UPnP and potentially other methods. This work provides inspiration for our proposals but requires additional infrastructure to route across different networks.

3.5 Summary

Existing systems provide almost all the required functionality but each has limitations. In particular, we argue that browser APIs are attractive but are not appropriate for accessing *remote* resources because they are inconsistent with web concepts. Personal servers are a browser-agnostic way to expose resources to other devices on the *local* network, but need additional routing capabilities and a discovery system. Consistent access control should be provided throughout.

4. REQUIREMENTS AND CHALLENGES

Based on the analysis of previous work we highlight the following requirements for a system designed to give browsers access to local and networked resources.

Routing and addressing. Web browsers must be able to access services from any personal device, including those on mobile networks, using address translation, or available over bearers such as Bluetooth. Addresses should remain consistent regardless of how the device is reached, and local network services should still be available when there is no internet connection.

Discovery. Web applications must be able to discover available devices and services. Discovery must be dynamic: internet connections can be unreliable and nearby computers may be switched on or off. However, to make this usable by web applications it must be possible to discover services using a consistent vocabulary and mechanism.

Browser lock-in. The ability to access local resources should, where possible, not be coupled with a specific browser. Web browsers are in fierce competition and a solution supported by only one could fail due to the browser's other features. Integrating with browsers has advantages, but remaining loosely-coupled would allow for faster uptake and interoperability.

APIs. Features should be accessed in a standard way by all personal devices. APIs should allow for synchronous and asynchronous functions, and support server-side events.

Evolutionary. While a radical refactoring of the web [4] would have long-term benefits, arguably success is more likely if web browsers are *extended* using the same abstractions that they use today. For example, addressing personal devices with URLs rather than arbitrary identities returned by JavaScript APIs. Using existing protocols and concepts would let web applications remain agnostic as to whether they are accessing services on the device, over a local network or via the public internet. This means that web applications need little modification and ought to make future upgrades (such as IPv6) easier.

Privacy-friendly. Giving web applications access to more resources should not have an adverse impact on user privacy either due to *fingerprinting* or a lack of access control.

Security. Introducing new browser capabilities increases the risk to users of exposing security-sensitive data or functions to malicious web applications. It should therefore be possible to integrate new proposals for generic mitigations [2, 9] as well as service-specific access controls. At the same time, the existing security model of browsers should be respected for compatibility and to avoid introducing new, subtle errors. Where suitable, origin-based access control methods ought to be re-used.

5. SOLUTION: THE BEST OF BOTH

We propose the following solution based on the combination of prior art in browser APIs, personal device servers and Intent-based service discovery. From Gibraltar [9] and *webinos* we require each device to run an HTTP and WebSocket server. Based on mDNS [8] and Ford et al. [6] we propose that a local DNS server resolves all domains with suffix '.zone' to localhost, with subdomains for user, device and service. Connecting to a '.zone' address will connect to the device server, which will use the *webinos* overlay network to route to the user, device and service requested. Because *webinos* already implements service discovery and routing across multiple users and networks we do not need to re-implement this. Discovery of services is primarily implemented via Web Intents. Access control is managed either implicitly through Web Intents, via CORS for HTTP requests or via policies based on origins for WebSockets.

5.1 Services

Our solution allows for local services to be accessed over HTTP or via WebSockets and channel messaging. HTTP allows local services to be accessed in the same way as web services, whereas WebSockets and channel messaging support server-side events for APIs which need to provide frequent updates. For service implementations we used the *webinos* project which defines APIs for features including messaging, sensors, actuators and more.

5.1.1 Access over HTTP

Each service is hosted on a domain with the pattern `service.device.user.zone`, resolving to the local device server. Service methods are invoked using GET or POST requests to a path `/methodName` and with parameters in the usual format (`argument1=value1`). For example, the Geolocation API's 'getCurrentPosition' method on Alice's phone would be at `http://location.phone.alice.zone/getCurrentPosition`, with the returned HTTP body containing a JSON-encoded

success or error response. It should be noted that this does not match the asynchronous existing API and does not use callbacks. However, Lin et al. [9] show that this approach is adequate for accessing most device resources.

5.1.2 Bidirectional communication

For bidirectional and server-side communication, web applications can open WebSocket connections to the service domain. This corresponds with suggestions by Lin et al. [9] and is similar to the *webinos* implementation. The protocol for communicating with each service is dependent on the API being implemented, but can use JSON-RPC for requests and responses.

WebSocket communication can be wrapped using JavaScript APIs. These may be easier for developers to use and provides a way to mimic browser-defined APIs while not requiring browser support. It also allows for the rapid introduction of new APIs for new services.

5.2 Discovery and selection

We suggest most applications use Web Intents to discover services. The local device server hosts a page at `http://zone/registry` containing mark-up for each supported *type* as a new 'discover' intent. E.g.:

```
<intent
 action="http://intents.w3.org/discover"
 type="geolocation"
 href="http://zone/registry/geolocation"/>
```

A web application would then declare that it wanted access to a service of the type 'geolocation' as per Figure 1. The intent service is hosted at `http://zone/registry/geolocation` and, upon being loaded by the browser, uses the *webinos* framework to find the currently available services of this type. If several services are available, the user may choose between them. Having selected an option, the page connects to the local server via a WebSocket. This connection is wrapped by a MessagePort object and returned to the requesting page which can then access the service as defined in section 5.1.2. For services which may be accessed via HTTP, the intent can return the URL of the service rather than just a MessagePort. This is less desirable, however, as it reveals the address of the service to the web application.

5.3 Security

5.3.1 Browser access control

For HTTP requests the use of separate subdomains for services gives each a unique origin and means that no web application has access to any service by default. In order to permit access to certain web applications, the web server at localhost will return CORS headers dictating permitted origins. CORS is primarily used for interoperability and consistency – The server will also need to implement access control, as CORS does not prevent the initial HTTP request being made, only the response.

Access via WebSockets can be mediated by inspecting the origin of the calling page. An access control policy system such as the one implemented in *webinos* [10] can be queried to make this decision. How to populate the access control policies is beyond the scope of this paper. If the origin's domain is `zone`, then the WebSocket request has been invoked via Web Intents. As such, permission can be implicitly granted *unless* a policy explicitly denies it. The WebSocket communication from `zone` must therefore include the calling page's origin.

We do not define a user authentication method. This can be provided at a lower layer, as in *webinos* [11], or through standard web approaches. For example, if Alice were accessing Bob's services over HTTP, Bob's service could indicate that Alice must first visit `http://service.pc.bob.com/authenticate` and present a credential, granting a session cookie. Because this approach is the same as normal web services, the full range of techniques are available.

Threats such as cross-site scripting (XSS) and cross-site request forgery (CSRF) remain. Solutions are orthogonal to our proposals, Lin et al. [9] give suggestions although these may be less successful in cross-device scenarios. Google Chrome introduce several mitigations [2] which constrain packaged applications and extensions. These may be successfully adapted to our solution as we do not introduce new metaphors or abstractions.

5.3.2 Outside the browser

The disadvantage of placing a web server on each device is that applications other than browsers may attempt to misuse them. Because access control is primarily based around the browser reporting the origin of the requesting application, malware can impersonate any origin and gain access to any resource. We suggest that the local server could request a certificate from clients connecting to it, and trusted browsers could have a suitable key and certificate installed. However, this is only as secure as the key storage mechanism, and it seems likely that malware would be able to bypass this mechanism. On operating systems offering a secure IPC primitive, such as Android, the WebSocket protocol could be replaced with an alternative, providing mutual authentication of the browser and the local server. This would require changes to the browser and would not be platform agnostic. We note that any system designed to provide inter-device connectivity will suffer from this problem.

5.3.3 Accessing external resources

Web applications can request access to domains referring to users, e.g. `location.pc.bob.zone` and the underlying *webinos* system will route requests to the correct endpoint. A translation step is required to convert these local domain names into full user identities but this is a straightforward mapping assuming the user is known to the system.

So that access control is still based on *origin*, in inter-user communication the origin of the application is replaced with an origin referring to the requesting user (such as `alice.zone`). Further restrictions based on the requesting application or device are also possible, but users may be unwise to trust this information.

5.4 Example

We imagine Alice is using a web application at `http://example.com/` on her PC which wants to access her location. Alice is with Bob, who has his mobile phone with a GPS. Example application code is shown in figure 1.

1. `http://example.com` uses Web Intents to request a geolocation service.
2. A browser prompt asks Alice to select a service. She clicks on the 'webinos' service, which loads a page hosted at `http://zone/registry/geolocation/`.

```
/* define an intent to 'discover' a
    geolocation service */
var intent = new Intent(
  "http://intents.w3.org/discover",
  "geolocation");
window.navigator.startActivity(intent,
    on_success);
function on_success(data, ports) {
  // Wrap the returned MessagePort
  geo = new GeolocationHelper(ports[0]);
  geo.getCurrentPosition(found);
}
function found(position){/* Render map */}
```

Figure 1: A web application accessing geolocation

3. This page finds available services by creating a WebSocket connection to `discovery.pc.alice.zone` (using the *webinos* service discovery API) and displays the results. Because Alice and Bob have communicated before, Bob's mobile phone is available.
4. Alice selects this service and the page initiates a WebSocket connection to `geolocation.phone.bob.zone`.
5. Behind the scenes, Alice's PC's device server proxies the connection via the *webinos* system, which establishes a bluetooth connection to Bob's device.
6. Depending on his policy, Bob will be asked whether `alice.zone` may access his location. Alice's server did not prompt her because the Web Intents process is used as implied consent.
7. The WebSocket connection is wrapped using a HTML5 MessagePort and returned to `http://example.com`.
8. `http://example.com` can send messages to this service directly or access it through a helper library, converting the message interface to a JavaScript object.

6. IMPLEMENTATION

The proposed architecture has been implemented (with minor exceptions) as a proof-of-concept extension to *webinos* using Ubuntu 12.04 and Google Chrome.

6.1 Device server

The domain name scheme was prototyped using marlon-tools DNS proxy[1]. We developed the device server using nodejs[2] and used it to interface with the *webinos* personal zone proxy[3] over a WebSocket connection. We have used *webinos* successfully for multi- device and user interaction. Because our extension only wrapped this capability, we are confident that the approach is sound.

For discovery we prototyped the Web Intents approach, demonstrating that the solution proposed in section 5.2 was feasible. Due to the limited support for channel messaging and Web Intents in current browsers we had to use a JavaScript Shim[4] to implement the service registration and selection. It was also necessary to simulate the passing of MessagePorts through use of the `postMessage` API. A disadvantage of our approach is the need for two steps of user input: when an intent was requested by an application, users must select our handler and then select the service that they want. As an alternative we implemented a single dynamic service registration page which populated the intent registry directly with the user's available services. This saves a step but requires this page to be regularly navigated to in order to find new services.

Access control for HTTP requests was prototyped successfully using CORS, although the translation from *webinos* XACML policies was not implemented; we expect this to be straight-forward. Similarly, access control for WebSocket connections is trivial to implement as *webinos* already contains a policy decision point which may be queried about web application permissions.

6.2 Services

We experimented with turning the APIs provided by *webinos* into services based on our scheme. This included Geolocation, the DeviceStatus API and the W3C File APIs. For APIs with more complex inputs and methods we encoded all JSON objects using JSON-RPC 2.0 encoding proposals[5].

7. EVALUATION AND DISCUSSION

We now evaluate our proposal and implementation based on the requirements identified in section 4. In this paper we do not analyse the performance of our implementation. There are no new processing steps compared with related work, and Lin et al. [9] and Gutwin et al. [7] have demonstrated that HTTP/AJAX and WebSockets, respectively, have reasonable performance when accessing local network and device services.

Routing and addressing. Our solution employs the same routing mechanism as *webinos* but could also be adapted to use UIA [6] or mDNS [8]. Devices are accessible on any network, thanks to the persistent connection between each device and the central hub. In addition, local service access is possible as a fall-back when there is no internet access. Multi-device and multi-user scenarios are supported.

Discovery. Service selection via Web Intents and *webinos*-based discovery was effective. While the implementation needed to adapt due to limited browser support, this approach successfully provided dynamic service discovery using a soon-to-be standardised mechanism. However, dynamism is limited to the point of discovery: after a service is selected, changes to the local network will not affect the application.

Browser lock-in. Our architecture uses standard protocols and APIs (or those expected to be standardised shortly) and is therefore theoretically independent of browser implementation. While WebSockets, channel messaging and Web Intents have varying support in different browsers, this should improve in the near future.

APIs. Our system allows for synchronous, asynchronous and server-side events. Both HTTP services and channel messaging (wrapped by JavaScript APIs) are supported, meaning that the most appropriate can be used at any time.

Evolutionary. The approach described in this paper uses existing concepts such as domains and origins and introduces no new changes to browsers beyond those currently in progress elsewhere. Furthermore, via the RESTful interface, this architecture exposes local network services in the

[1] `http://code.google.com/p/marlon-tools/`
[2] `http://nodejs.org/`
[3] `https://github.com/webinos/Webinos-Platform`
[4] `http://webintents.org`
[5] `http://www.simple-is-better.org/json-rpc/jsonrpc20-over-http.html#encoded-parameters`

same way that normal web services might be. However, the naming system is only consistent for the current user as usernames are not unique. This constraint is due to limitations on the format of domain names. For the most part, simple translation solves the problem.

Privacy. Web Intents maintain the anonymity of the service being used, which ought to limit the fingerprinting capability of web applications. Intents could also be used to protect user privacy by giving web applications access to fake services than do not return real data. This would allow users to trial applications without revealing private information. However, in practice, the presence of a device server may be detectable by web applications through timing attacks.

Security. For Web Intents, security depends on users giving web applications consent at runtime to access services. Security for other access methods depends on the existence of a policy dictating whether a particular origin should be allowed access to a resource. This approach should be compatible with other origin-based mechanisms such as CORS. However, the need for some new infrastructure to create policies is unavoidable. A disadvantage of our approach is that it does not allow for API-specific security controls. For this reason we expect it would be improved with further integration into the browser, despite all the related standardisation and compatibility challenges.

Security for users and service providers is still dependent on web applications being trustworthy and resistant to attack. A history of vulnerabilities make this an unreasonable assumption. The problem is independent of our work but implies that adoption of our approach for more security-sensitive resources must coincide with general improvements to browser and web application security.

8. CONCLUSION AND FUTURE WORK

We have presented a practical approach to accessing local device features and network resources from web browsers. Our system combines existing proposals to provide services in a consistent way with no significant modifications to browsers or applications. It is based on a custom DNS scheme, service access via HTTP and WebSockets, and Web Intent discovery. Our approach solves problems in addressing, discovery, browser lock-in, security and privacy and considers issues around standardisation and adoption. Having developed an implementation and prototyped several services we have shown this approach to be feasible.

For future work we intend to address some of the integration challenges that prevent our proposals from being incorporated into *webinos*, including the lack of browser support for some features. We will also investigate how this approach can be extended to support custom authentication and authorisation schemes for services.

9. ACKNOWLEDGEMENTS

The research described in this paper was funded by EU FP7 *webinos* Project (FP7-ICT-2009-5 Objective 1.2).

10. REFERENCES

[1] Adobe. PhoneGap. `http://phonegap.com/`, 2012.

[2] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the USENIX Security Symposium 2012*, August 2012.

[3] Web Intents Addendum - Local Services (W3C Editor's Draft). `http://dvcs.w3.org/hg/dap/raw-file/tip/wi-addendum-local-services/Overview.html`, September 2012.

[4] J. R. Douceur, J. Howell, B. Parno, M. Walfish, and X. Xiong. The web interface should be radically refactored. In *Proceedings of HotNets '10*. ACM, 2011.

[5] Electronic Frontier Foundation. Web Browsers Leave 'Fingerprints' Behind as You Surf the Net. `https://www.eff.org/press/archives/2010/05/13`, May 2010.

[6] B. Ford, J. Strauss, C. Lesniewski-Laas, S. Rhea, F. Kaashoek, and R. Morris. Persistent Personal Names for Globally Connected Mobile Devices. In *Proceedings of OSDI '06*. USENIX, 2006.

[7] C. A. Gutwin, M. Lippold, and T. C. N. Graham. Real-time groupware in the browser: testing the performance of web-based networking. In *Proceedings of CSCW'11*, pages 167–176. ACM, 2011.

[8] IETF. Multicast DNS. `http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt`, December 2011.

[9] K. Lin, D. Chu, J. Mickens, L. Zhuang, F. Zhao, and J. Qiu. Gibraltar: exposing hardware devices to web pages using ajax. In *Proceedings of WebApps'12*. USENIX, 2012.

[10] Lyle et al. Cross-platform access control for mobile web applications. In *Policy 2012*. IEEE, July 2012.

[11] Lyle et al. Personal PKI for the Smart Device Era. In *Proceedings of EuroPKI'12*, LNCS. Springer, 2012.

[12] T. Mikkonen and A. Taivalsaari. Reports of the web's death are greatly exaggerated. *Computer*, 44(5):30 –36, may 2011.

[13] Mozilla. Boot to Gecko Project Website. `http://www.mozilla.org/en-US/b2g/`, 2012.

[14] The W3C. Mobile Web Application Best Practices. `http://www.w3.org/TR/2010/REC-mwabp-20101214/`, December 2010.

[15] H. S. Tømmerholt. Opera Unite developer's primer. Available on `http://dev.opera.com/`, October 2009.

[16] UPnP Forum. `http://upnp.org/`, 2012.

[17] W3C. The Device APIs Working Group Website. `http://www.w3.org/2009/dap/`, 2012.

[18] Cross-Origin Resource Sharing (W3C Working Draft). `http://www.w3.org/TR/cors/`, April 2012.

[19] Geolocation API (W3C Proposed Recommendation). `http://www.w3.org/TR/geolocation-API/`, 2012.

[20] The WebSocket API (W3C Editor's Draft). `http://dev.w3.org/html5/websockets/`, Sep. 2012.

[21] The Wholesale Applications Community (WAC) Website. `http://www.wacapps.net/`, 2012.

[22] The *webinos* project. `http://webinos.org/`, 2012.

[23] Web Intents (W3C Working Draft). `http://www.w3.org/TR/web-intents/`, June 2012.

[24] HTML Living Standard: 10.4 Cross-document messaging. `http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html`, September 2012.

[25] M. Zalewski. *The Tangled Web: A Guide to Securing Model Web Applications*. No Starch Press, 2011.