Web-based Framework for Accessing Native Opportunistic Networking Applications

Marcin Nagy*, Teemu Kärkkäinen†, Arseny Kurnikov*, Jörg Ott†
*Aalto University, Finland
firstname.lastname@aalto.fi
†Technical University of Munich, Germany
kaerkkae|ott@cs.tum.edu

***Abstract*—Opportunistic networking is one way to realize pervasive applications while placing little demand on network infrastructure, especially for operating in less well connected environments. In contrast to the ubiquitous network access model inherent to many cloud-based applications, for which the web browser forms the user front end, opportunistic applications require installing software on mobile devices. Even though app stores (when accessible) offer scalable distribution mechanisms for applications, a designer needs to support multiple OS platforms and only some of those are suitable for opportunistic operation to begin with. In this paper, we present a web-based framework that allows users to interact with opportunistic application content without installing the respective app and thus also includes users whose mobile OSes do not support opportunistic networking at all via minimal stand-alone infrastructure. We describe our system and protocol design, validate its operation using simulations and a testbed. We implement web versions of two existing native mobile opportunistic applications: PeopleFinder and Here & Now.**

I. Introduction

Research into opportunistic networking over the past decade has spawned a large number of different systems and architectures (e.g., *Haggle* [23], *MobiClique* [19], *SCAMPI* [8], *IBR-DTN* [4], *PodNet* [11], and *NetInf* [3]). These platforms come with a variety of applications that work in environments with limited or no Internet connectivity. Instead of detouring information exchange via a centralized "cloud", these systems disseminate the information directly between co-located devices over short range links.

Opportunistic networking architectures can be broadly divided into two categories: fully ad-hoc and infrastructure-assisted. Systems of the former are based only on direct encounters between nodes, while those belonging to the latter introduce distributed infrastructure components, sometimes called "throwboxes" [30], which serve as (possibly temporary) fixed points that offer services to nearby mobile nodes. One such opportunistic infrastructure architecture is the *Liberouter* system [7], which combines an inexpensive embedded hardware platform (Raspberry Pi or Intel Edison) with an opportunistic networking middleware (SCAMPI, IBR-DTN) to create a do-it-yourself opportunistic router. In this paper, we extend our *Liberouter* platform to support mobile devices running just web browsers. Our extension framework enables interactions with existing opportunistic networking applications from unmodified web browsers within the vicinity of a *Liberouter* device. [15]

Enabling access via web browsers to opportunistic networking applications has two main benefits:

1. Support for more platforms. Implementing a native opportunistic networking platform requires support from operating system that many mobile OSes do not provide (e.g., long running background processes capable of doing networking). However, every mobile OS platform has a modern web browser, which the users are accustomed to using as a gateway to rich online services.

2. No need to install native software. Even users of devices that can support a native opportunistic networking platform are required to install and run an application to access the network. While today's users are accustomed to installing hundreds of apps on their devices, each additional step the user must take will reduce the uptake the service gets. Furthermore, the users are used to installing apps from the OS vendor's app store, while in an opportunistic scenario the apps are likely to be distributed via other channels. For example, the *Liberouter* devices distribute the apps directly to the users via a local web portal, which may feel foreign to some users and cause the OS to throw up security warnings.

The main challenge that needs to be solved by the framework is the interworking between the peer-to-peer style opportunistic networking and the client-server style web access approaches. While native opportunistic networking platforms are designed to exploit short, random encounters between nearby nodes to pass around messages, web browsers are designed around the client-server model where application data and logic are fetched from always-accessible servers. Furthermore, the software development and distribution models for native mobile apps are well developed and understood, which also holds for many native opportunistic networking applications. However, the development and application distribution mechanisms for web browser-based adaptations of mobile opportunistic applications are unsolved problems.

Our solution is based on leveraging two key concepts: a *throwbox* based deployment and *self-contained and semantically meaningful messages.* While not every opportunistic networking system exhibits these characteristics, many practically

978-1-7281-0270-2/19/$31.00 2019 IEEE

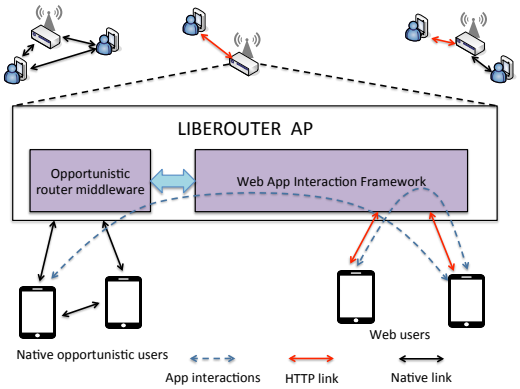


Fig. 1: Overview of the *Liberouter* based system model.

deployable systems do. In particular, throwboxes are needed for very low density deployments, and self-contained messages for store-carry-forward routing.

Throwboxes serve as natural points to deploy web servers, to which the web browsers can connect. Our framework runs on these throwboxes, serving opportunistic web apps accessible through unmodified web browsers.

Self-contained and semantically meaningful messages allow us to attach the necessary logic to interpret content of the message. This solves the problem of distribution of a web application logic in the mobile opportunistic environment and allows web apps to have simple semantics and design.

Next, in Section II, we describe our system model and architecture at a high level, and provide a justification for our message based approach from a distributed systems view point. We then provide the detailed design of our framework in Section III, and its proof-of-concept implementation, including integration with an opportunistic network platform and adaptation of two existing applications, the Google People Finder native application and Here & Now, to use our framework, in Sections IV and V. We end with a simulation and testbed based validation of the design in Section VI, a look at the related work in Section VII, and our conclusions in Section VIII.

II. System Model

There are two main aspects to the underlying system model: 1) the *network model* that defines the communicating entities, their roles and interactions, and the application messaging characteristics, and 2) the *application model* that defines the structure and the behavior of the web applications targeted by our framework. We describe these different aspects next.

A. Network Model

We assume an underlying opportunistic networking system composed of stationary lightweight infrastructure nodes and mobile data carrier nodes—our implementation extends the Liberouter [7] neighborhood networking system, but the design is generally applicable to other proposed systems such as the IBR-DTN [4]. As is usual for opportunistic networks, we only assume the existence of transient, direct, short-range, peer-wise communication contacts, which result in a space-time network without instantaneous end-to-end paths. This assumption naturally leads to a messaging model with two key characteristics. First, the messages are assumed to be large, semantically-meaningful and self-contained, which allow them to be forwarded and replicated between nodes without the need for end-to-end packet paths. Second, since the delays and delivery probabilities are highly variable, no globally consistent state is reached and instead the applications create *locally consistent* states from the random subset of messages that they have received at any given point in time.

The approach we take is to extend the lightweight infrastructure nodes—the Liberouters—with a new framework that allows them to provide nearby nodes access to web adaptations of native opportunistic networking applications using an unmodified web browser. Figure 1 shows the high level model of these nodes. The stationary nodes that act as Wi-Fi access points (Liberouter AP in the figure) to which nearby mobile clients can connect as Wi-Fi stations. We consider two types of users: 1) native users who have installed the opportunistic networking software and applications (bottom left), and 2) web users who are assumed to only run an unmodified smartphone with no special software installed (bottom right). This leads to native and HTTP links respectively. The additional application interactions that we aim to enable through our framework are between the web browser users and the native opportunistic network users (the native users can already interact among themselves). We extend the opportunistic routing functionality of the Liberouter devices by adding the *Web-based Framework* between the local opportunistic router instance and a local web portal. The interactions are bi-directional, allowing the web users to both consume and produce content for the existing opportunistic networking applications.

B. Application Model

Interactive applications with (graphical) user interfaces, such as these running on mobile devices, opportunistic or not, can conceptually be described by the the Model-View-Controller (MVC) paradigm [20], which may also serve as an implementation model for such systems. We are not tied to this model, but rather use it in the following for illustration purposes. As shown at the top of Figure 2, the MVC model comprises three elements: a data *model (M)* (schema plus actual data) represents the application state as structured data, a *view (V)* of this model is rendered to the user via a GUI, and a *controller (C)* receives input from the user via the GUI and acts upon the input by modifying the model accordingly (after validating the input). The code for such (opportunistic) applications usually resides in the user device (e.g., tablet, smartphone) and thus requires installation before being able to interact with the respective application and its contents. Instead of having the users install binaries, our framework allows the application developer to replicate parts of the view and controller code along with the model for message interpretation and generation inside the message itself in addition to the complete (self-contained) state as shown in top-right of Figure 2.

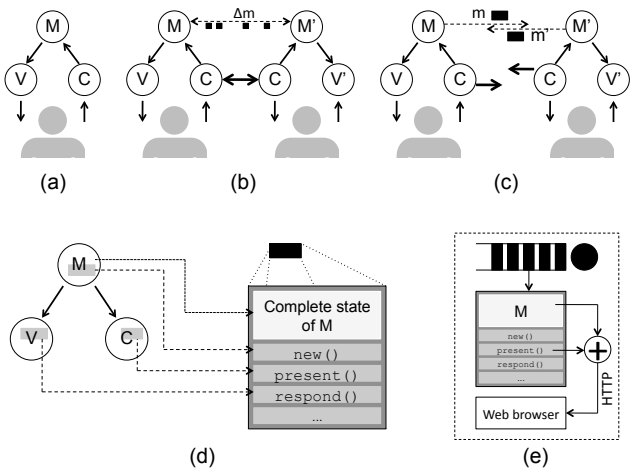


Fig. 2: Application model: (a) basic MVC model; (b) sample extension of MVC to a distributed system with a synchronous communication channel between two controllers exchange manipulations (deltas) to the model; (c) MVC variant with an asynchronous channel and controllers exchanging full state in self-contained messages; (d) including model interpretation and manipulation code in messages; (e) utilizing the interpretation code to render content in a web browser over HTTP.

To execute the code embedded in the framework-compliant messages, we design an application-independent interpreter that is integrated with an opportunistic networking router and has access to its buffered messages. The interpreter screens messages queued in the router and processes those containing framework-compliant code. It extracts and formats the contents to make it accessible via standard web technologies (HTTP as transport and HTML5 for presentation/interaction) so that any smart mobile device can interact with the contents, as depicted at the bottom Figure 2. This enables any node to dynamically interpret and render messages of any framework-compliant applications, allowing web users to access and interact with the contents, generate replies, and create entirely new messages.

We foresee that the router (e.g., SCAMPI or IBR-DTN) and the interpreter run on dedicated opportunistic (infrastructure) devices, e.g., *Liberouters*, which also serve as access points and message relays for mobile devices. However, they could also run on other mobile user devices.

III. Web App Interaction Framework

We now turn to the details of the framework. We begin with the description of the framework and its functionality at a conceptual level in Section III-A. Section III-B provides a detailed description of the framework including functionality and interactions between the components. Section III-C describes the bootstrapping of web applications. Finally, in Section III-D, we discuss security implication for the framework design.

A. Conceptual Model

Our goal is to enable users to interact with existing native opportunistic networking applications via a web browser. To achieve this, we need to design a generic web-framework that can replicate the native application models with the ability to both *present* application-specific content received from the network and to *generate* new content into the network in a form interoperable with the native applications. This is made possible by the nature of opportunistic networking (see Section II-A), where communications between applications do not happen via tight end-to-end control loops, but rather via a public, asynchronous, unreliable exchange of self-contained and semantically meaningful messages. This allows the web-framework to run a generic approximation of the native application model and expose it to web browsers.

Conceptually, the core idea of the framework is the ability create generic approximations of the native applications. We achieve this by mapping the popular MVC paradigm (see Section II-B) to *transformations* between applications state, views and messages as shown in Figure 3. In the figure (top left), the MVC model corresponds to the *app state*, the MVC view to the *app view and message views*, and the MVC controller logic to transformations between these. The evolution of the application model in response to received and deleted messages is shown at the top of the figure, which illustrates how the transformations are simply functions that advance the state (model) or create views in response to message events. Similarly, the bottom of the figure shows how a new message can be created from a user input via transformation functions.

While the generic MVC structure is the same for all apps, each different application has its own custom details for the business logic. This application-specific behavior is captured by the transformations. In essence, they are a discrete and concise representation of the application's business logic, which we can then attach to the messages generated by the application. The combination of the generic framework installed in a device, such as a *Liberouter*, and the application-specific transformations that are carried along with the messages, is everything that is needed to produce an approximation of the native application accessible via a web browser.

For the app view transformations, we define four types: *application presentation*, *message presentation*, *message creation*, and *message response*. The application presentation transformation maps the application state into an HTML view to be displayed via a web browser. For example, in a photo sharing app, it could correspond to a list of photos view. The message presentation transformation generates a concise view of a particular message. This transformation could correspond to a detailed photo view together with its comments, in a photo sharing app. Additionally, we also allow the app to define custom transformations that use the application state to return arbitrary data. The remaining two transformations are used for generation of new messages via the web browser. The message creation and response transformations produce views that can be used for submission of the new message content. Although they output views, we consider them to be a part of the controller, as data submitted through them leads to creation of a new message. The only difference between them is that the message creation transformation does not take any input as it

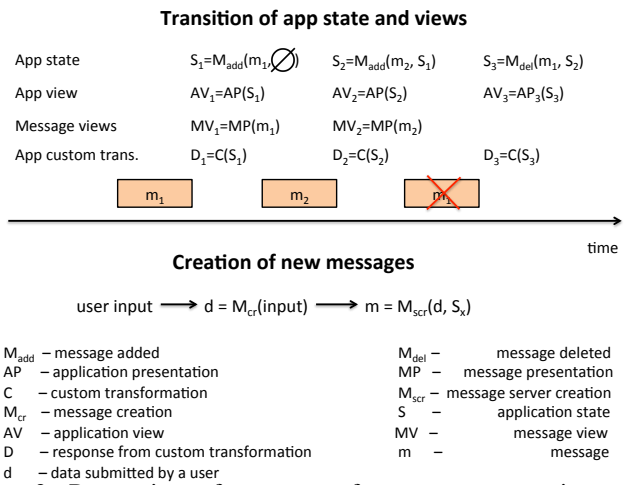


Fig. 3: Processing of messages for content extraction and presentation in web browsers.

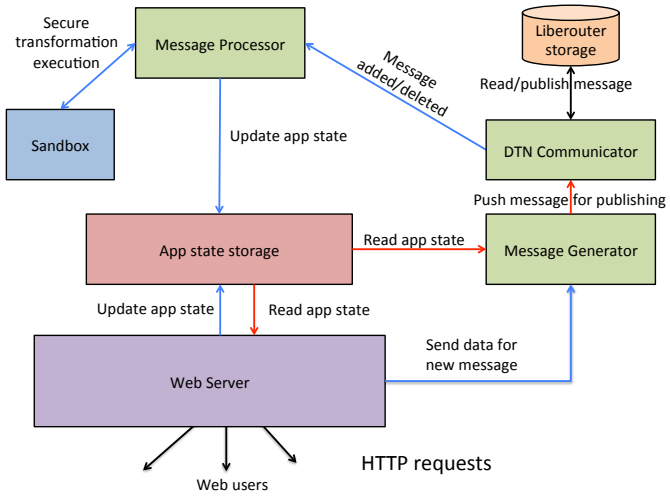


Fig. 4: Framework architecture.

produces a new context for the message, whereas the message response transformation uses context of the message to which the user responds as the input parameter. In the example of a photo app, the message creation transformation could be used for submitting new photos, while the message response transformation is used for commenting on an existing photo.

For the app controller transformations, we define four types: *message addition*, *message deletion*, *message server creation*, and *message server response*. The message addition transformation is executed always when a new application message is received by the framework. It takes the application state and the newly arrived message content to generate the updated application state as the output. Similarly, the message deletion transformation is executed when an application message is deleted. It takes the application state and the content of the deleted message and produces the updated state. The message server creation and the message server response transformations can be used for modifying data submitted via the message creation or the message response transformation respectively before the actual new message is created. This is useful for custom operations specific to the app that may not be performed by a frontend logic (e.g., uploaded file must be compressed to a specific format).

B. Framework Design

The system design for the framework is shown in Figure 4. It comprises five main components: *Message Processor*, *Message Generator*, application state storage, web server and sandbox. In addition, to decouple the framework from an underlying opportunistic system, the framework defines a *DTN Communicator* which 1) tracks changes in the opportunistic router storage (e.g., a *Liberouter* in the figure), and 2) acts as an adapter for the message format of the underlying system. Note that the opportunistic router storage contains only raw messages, whereas the application state storage contains states of applications generated by transformations from the opportunistic messages as they are received.

The *Message Processor* handles processing of newly arrived (or deleted) messages by supervising the execution of an appropriate controller transformation and updating the application state in the storage. The execution of the transformations are done by the sandbox for security reasons (see Section III-D). Change of the application state triggers execution of appropriate view transformations that are affected by the updated state.

The web server is the user facing component. It presents application content to the web user by exposing a set of endpoints, exposes application data to the front-end logic and handles requests for new content creation. The presentation of content is provided by a view transformation, which is executed inside an endpoint handler. Exposure of application data is realized by a custom transformation. Finally, the web server, upon receiving a request to create new application content, validates it, executes an appropriate controller transformation if required and passes it to *Message Generator* component. The *Message Generator* creates a new message from the submitted data and publishes it using the *DTN Communicator*.

C. Bootstrapping Applications and Nodes

So far in our design the transformations are always shipped along with the content. This ensures compatibility between the content and the application logic. However, it also implies that web users can generate content for a web application only if the framework they connect to has already stored a message for the application (which contains the necessary transformations). In practice, this means that every web application must be bootstrapped from its native equivalent. To overcome this limitation, we take two further steps: 1) distribute native applications within the framework to increase their availability in the opportunistic network, and 2) allow creation of initial content from the web applications.

To address 1), the framework offers the disconnected "app store" functionality by making all native applications available to web users for download via the web browser. For 2), we also include transformations in the messages used to distribute native applications. A user connected to a framework instance

can then choose to install a native version of an application, or create new content using just the Web version.

D. Security considerations

To guarantee robust and secure operation of the framework, our security considerations focus on two issues: 1) secure execution of a transformation and 2) message authenticity. To this end, our threat model assumes an adversary that: 1) disrupts the framework functionality by executing malicious code, 2) obtains unauthorized access to the framework or application state storage, and 3) impersonates another user by sending a message that masquerades its originator for another user. Furthermore, our threat model assumes also: 1) the presence of basic Linux platform security mechanisms, and 2) presence of a disconnected public key distribution system [27] which assigns a public key to an identifier of the message originator in the opportunistic network. Examples of such PKI systems are SocialKeys [16] and *PeerShare* [13].

Secure execution of transformations. Execution of transformations of an unknown origin poses a serious threat to the secure functioning of the framework. A malicious transformation may include system calls causing disruption of the framework operation and possibly even the whole device the framework is running in (e.g., the transformation switches off all network interfaces). The other set of threats comes from unauthorized content access. These include pollution of content by generation of fake messages, deletion of messages and illegal modification of another application state. These threats motivate implementation of file system level isolation and system call filtering. The file system level isolation constrains the transformation to access only the data contained inside the message and parts of the shared memory that are related to it. As system calls are still available to the transformation (via system libraries), the system call filtering must prevent invocation of any system calls other than I/O operations on the message content to which the transformation belongs (see section IV for details). Since transformation access to the file system is constrained and it cannot invoke system calls other than file system I/O, the security requirements are fulfilled.

Message authenticity. We address the threat of user masquerading by providing mechanisms to verify message authenticity. This requires that the messages are signed by their originators. Verification of the message authenticity is realized in the web browser (via Web Cryptography API) and it assumes that the public key is accessible in the browser for the web application (e.g., it is present in the IndexedDB). Section IV provides implementation details.

IV. Implementation

We now turn to the implementation details. We first cover the implementation of the core framework components, followed by the security component details. Then we explain the web application transformations and finally conclude this part with a brief performance evaluation of the framework.

Core framework. The *Message Processor* is written in Java and processes updates of the *Liberouter* storage by: 1) parsing the metadata of received messages and storing them inside Redis[1] (acting as the app state storage) and 2) clearing Redis of data removed from the *Liberouter* storage. It also uses a Redis queue to notify the web server about the changes.

The web server is written as a Node.js application. It implements the generic Web application model as a set of HTML templates with empty *div*s, which are filled with transformation outputs. The template engine uses EJS[2]. In addition, the web server enables web applications for querying app data using a REST API. As a result, modern web frameworks (e.g., React) can be used for the web app development. The generic web application model enables content generation for the web users through application specific HTML forms. The web server validates submitted forms and sends them to the *Message Generator* using a Redis queue. Finally, the web server provides the "app store" functionality by allowing web users to upload native applications via an HTML form.

The *Message Generator* is a native Java application that: 1) reads a new content request from the message queue sent by the web server, 2) performs application-specific encoding of the message, and 3) publishes it to the *Liberouter* cache.

Security components details. Recall from Section III-D that our threat model calls for implementating the file system level isolation and the system call filtering. We do the file system level isolation by putting all message related data inside one directory, which is put inside a `chroot` jail. The system call filtering is realized using `seccomp-bpf` [1]. It allows whitelisting of a subset of system calls that a transformation has a permission to access. Since the transformation should be given access to all the metadata carried by the message, our `seccomp-bpf` profile whitelists system calls that open and operate on files. The entire sandbox is implemented in C.

Verification of message authenticity requires a public key availability in the browser via HTML5 local storage or access to the device persistent storage (e.g., the File API). Both of these features are supported by all modern browsers. The authenticity verification is realized fully inside the JavaScript code via the Web Cryptography API.

Web application transformation. The transformations are implemented in Python, and must follow the basic guidelines described in Section III-A. The framework gives the transformations access to the Bootstrap library, so that they can generate a sophisticated HTML presentation of the content without carrying additional CSS style files.

Framework performance evaluation. We evaluated our framework implementation by measuring the execution time of each component while processing a single message of the GuerrillaTags application [7]. The experiments were performed on a Macbook Pro (2.66 GHz Intel Core i7 CPU with

[1]https://redis.io/
[2]https://ejs.co/

8GB of RAM) running OS X 10.10.1 and repeated 30 times to obtain statistical significance.

The *Message Processor* and the sandbox are the most resource consuming components (Fig. 5). Further analysis showed that their performance was I/O bound, and both of them perform high number of I/O operations compared to the other components. Overall, the average execution time of a new message in the storage is about 400ms, while generation of a new message takes about 15ms. Our implementation has not been optimized for performance and we believe significant gains could be made.

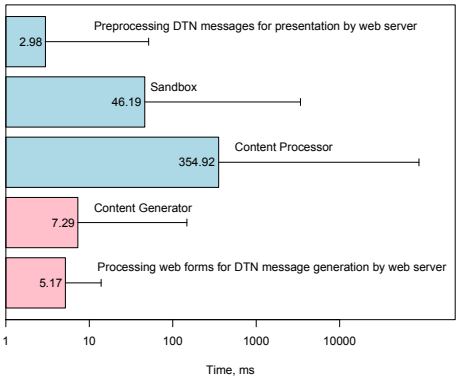


Fig. 5: Mean execution time (measured in milliseconds) taken by each framework component (logarithmic scale).

V. Integration with a DTN platform

Our framework is designed to be independent and not bound to any specific opportunistic platform. To enable integration with every platform that follows the messaging model described in Section II, we provide the *DTN Communicator* component. In the simplest case, the framework operations can be purely local file system reads and writes, and no dissemination happens beyond the *Liberouter* device. To enable content dissemination beyond the local device, the framework needs to be integrated with some opportunistic platform. To demonstrate this concept in practice, we integrated framework with SCAMPI, an opportunistic networking platform.

SCAMPI. SCAMPI is an opportunistic network platform that fulfills all the requirements of the opportunistic platform. It provides: 1) a data cache, 2) a publish-subscribe communication model, and 3) a network layer implementation capable of delivering messages based on the store-carry-forward networking. Our tech report [14] contains more details on SCAMPI.

Implementation of the *DTN Communicator* for SCAMPI requires: 1) developing a cache monitor as a directory tracker (SCAMPI implements cache as a set of file system directories), and 2) using the SCAMPI*AppLib* library to publish new content to network.

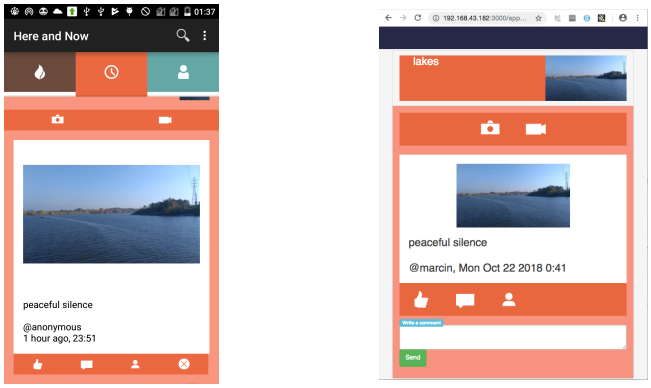
Developing web applications. To build a web version of an existing native application, the developer must implement the transformations required for the application functionality and attach them to the application messages as metadata. As a bare minimum to support presentation of content, *message addition*, *message deletion* and *application presentation* transformations must be implemented. The remaining transformations are required for providing more sophisticated functionality.

To facilitate the development process, we also built a simple transformation development environment. It allows the developer to test correctness of the implementation by: 1) executing the transformation, and 2) verifying that the generated HTML view matches expectations.

We present two applications: *People Finder* and *Here and Now*, as examples of web application development for our framework. We focus on these two applications, as they are relatively complex, but additionally we have also created web versions of *Guerrilla Pics* and *Guerrilla Tags* [7] apps.

People Finder. *People Finder* is the adaptation of Google Person Finder web application into disconnected environments. Google's version of the application has proven its usability in various disastrous scenarios starting from 2010 Haiti earthquake. The application allows generating records of a missing person and attaching notes to those records. Each message generated by the application contains the record and a set of notes related to the record known by the sender. The application view is generated from the person records. Notes attached to a person are displayed in the detailed view, which is generated by the *message presentation* transformation. Adding a new note for an existing record is done through the *message response* transformation, which gets the original record (and notes) as a parameter. This transformation appends a new note to the existing ones, and generates a new aggregate message with the record and all known notes. A new missing person record is generated by the *message creation* transformation.

Here and Now. *Here and Now* is an experience sharing application. It allows to share media content (i.e., photos and videos) among people that are physically collocated. Users can also comment on the content they see and give "likes" to it. Media content together with comments and "likes" are grouped into topics. The application offers two different views for presenting topics. The first one is called "trending topics" and it includes topics sorted according to the number of "likes" in the descending order (topics that lack a single "like" are not presented in this view). The second one is called "happening now topics" and it includes all topics sorted according to the most recent content posting time. Finally, the application provides also the third view called "around me" in which users that were observed in the last one minute are listed. We have implemented the web version of the application using React, as huge number of user interactions makes "classic" JavaScript state management challenging and we want to take advantage of the state management provided by React. We implement the whole application UI as the React app to provide the same functionality as its native equivalent. User interactions like adding a new photo, or commenting on an existing one are implemented as AJAX requests. These requests are handled by the *message server response* transformation. In addition, to provide the good user experience, the React app regularly sends requests to the *custom* transformation to fetch the latest content. The whole React app is the compiled piece of code



(a) Native version (b) Web version

Fig. 6: Comparison of native and web-based user interface for *Here and Now* app.

that is attached to messages as meta data. It is loaded by the *application presentation* transformation. Figure 6 shows the native Here and Now, and its web version.

VI. Validation

Our framework offers, in principle, content access to web users (in addition to native app users). To achieve this, the framework requires code to be shipped along with message state updates, which incurs overhead. In the following, we first evaluate the overhead and its impact. We then turn our attention to how many web nodes could be *reached* by content if those nodes choose to look at messages. Finally, we evaluate our framework implementation by running a series of experiments in a testbed environment.

A related question is if web users moving between access points would also contribute to the connectivity of an opportunistic network. We have shown that they can make a difference if the Liberouter nodes instrument the web storage of mobile browsers for relaying messages [15].

A. Overhead

To evaluate overhead, we measure the actual message sizes of our implementation for a text messaging and photo sharing applications with sample contents. For a text messaging application, the message size grows almost 50-fold from 350B to 16kB. However, this is only due to the small size of the native messages. If the content size of the application messages increases, the overhead becomes more reasonable. For photo sharing, with small photo size of 65–120 KB, including the framework code adds just 5–10% overhead.

Obviously, the overhead added via the framework is a function of the complexity of the logic required to interpret, render, and construct messages: simpler applications will need less code. The overhead is obviously also a function of the content size so that more elaborate content will cause, even if more complex code is needed, limited overhead only. One can argue that trends in web site complexity and size[3] show that increasing amounts of effort are put into conveying

[3]http://www.websiteoptimization.com/speed/tweak/average-web-page/

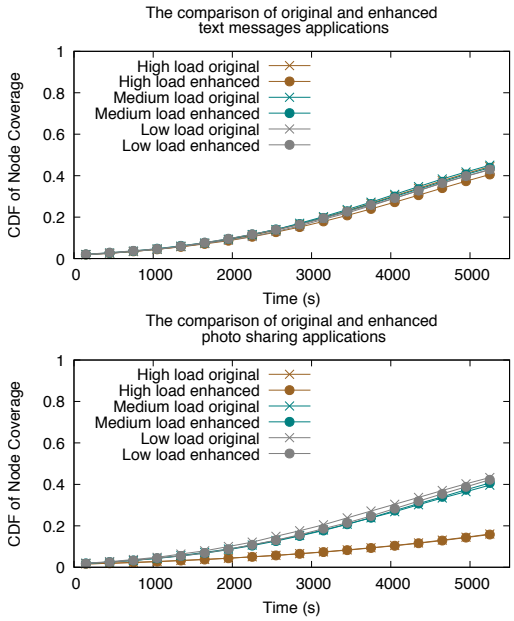


Fig. 7: Impact of the overhead introduced by the framework (reflected in different message sizes) on the coverage for messaging (top) and photo sharing (bottom) with all three loads for the SPMBM model.

probably roughly the same amount of content. Thus, adding more sophisticated interaction framework code for a better experience mirrors what is already done on the Web.

In opportunistic networks, the most important question is if and how the larger message sizes affect message delivery performance. To this end, we carried out simulations using the ONE simulator [9] with two different mobility models: 1) SPMBM: Shortest Path Map-Based Movement between waypoints chosen from the Helsinki downtown map ($4.5 \times 3.4\text{km}^2$) [9] for 50, 100 and 200 pedestrians moving with speeds $v = U(0.5, 1.5)$ m/s without predefined points of interest. 2) Same as 1) but additionally we introduce 10, 65 and 325 static access points. Our nodes communicate at a net bit rate of 2 Mbit/s with a radio range of 50 m. The nodes use simple epidemic routing [26]. We choose a random node to generate a new message every 12 s, 60 s, and 300 s, referred to as high, medium, and low load, respectively. Messages expire after 5400 s. The message sizes correspond to those for native and framework-enhanced messages for the text and photo sharing applications to pick two extremes. We measure the fraction of nodes that obtain a copy of each message, termed *coverage*, and plot the average of ten 12-hour simulation runs.

As shown in Figure 7, we find that the overhead of the framework does not notably impact the performance results. The coverage remains the same both for using native application messaging and for the framework-enhanced messaging for the text chat application (top) as well as for photo sharing (bottom).

Further, a maximum message rate limit was also observed in past experiments, which have shown that the per-message overhead of protocol implementations appears to be more

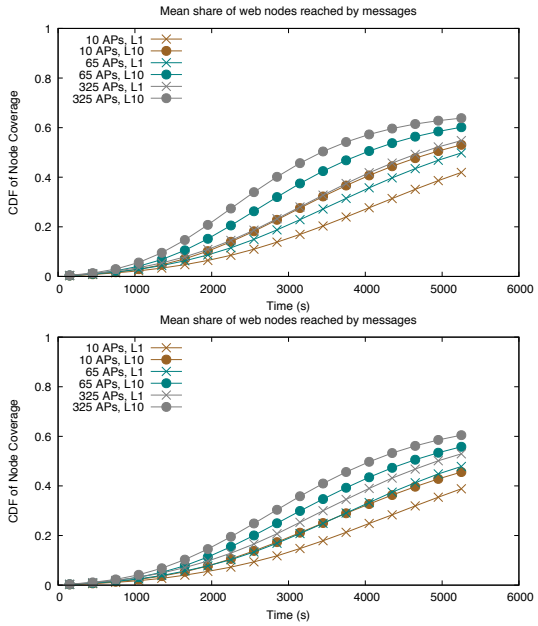


Fig. 8: Coverage achieved for 50 and 500 web nodes for the SPMBM model with $10-325$ access points for the medium load for text messaging (top) and photo sharing (bottom).

dominant in communication performance than the per-byte overhead. This was, for example, found in a comparison of three different DTN bundle protocol implementations [12], particularly for growing the payload size from 10 bytes to 10 KB and beyond. Our own (not yet statistically significant) experiments seem to confirm this. While implementation details play an important role here, there are also systematic aspects to consider: nodes that meet need to exchange vectors which messages they have, decide which ones to replicate, and then perform a forwarding process for each message, which causes per-message overhead. Researchers also found that neighbor discovering and pairing with peers is expensive and takes easily tens of seconds [18] while a 20 KB data transfer takes only 160 ms even assuming just 1 Mbit/s data rate. We therefore argue that the framework overhead is not of substantial importance in practice.

B. Content Reach

The previous subsection suggests that the overhead introduced by our framework won't degrade performance for native nodes. But how well does the framework allow reaching out to web nodes? We conduct further simulations to answer this question, using largely the same setup as above, but we introduce $10-325$ *access point nodes* (APs) which, besides running the DTN middleware protocols, serve as WLAN access points and run the server side of the interaction framework (cf. Figure 4). We also add *web nodes* that only interact with the access points, but neither with each other nor with regular DTN nodes. We choose the number of web nodes to be equal (L1), five-fold (L5), or ten-fold (L10) the number of DTN nodes.

Obviously, how many web nodes we can reach will depend on the movement patterns of those nodes and where the access point nodes are located, and how they move. However,

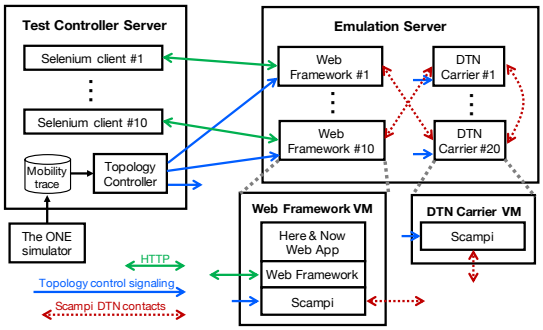


Fig. 9: System model for the testbed evaluation.

our simulation results shown in Figure 8 hint that we can notably increase the visibility of content generated by native applications; without the web framework this content would not be accessible by web nodes. In our simulation results, we find that the content coverage may come close to that of the native nodes and reach close to 40% of the web nodes with only 10 APs deployed. Increasing number of access points causes significant improvement in the content coverage, as almost 60% of web nodes are reached with 325 APs available. Comparing this to Figure 7, content availability is roughly equal for web nodes and native nodes. Note that, the fraction of web nodes reached gets bigger as their number increases from L1 to L5 to L10, so the absolute number of additional nodes reached grows even more.

C. Testbed Evaluation

To validate the system implementation, we conducted a testbed evaluation with 10 APs and 20/30 DTN carriers running in virtual machines (VMs). We generate and receive content of the Here and Now application from the web framework via Selenium[4]—a web app testing framework using a real web browser (Chromium) as a driver—with a single Selenium instance continuously attached to each web framework instance. We do not emulate movement for the web clients, instead focus on how content produced in one framework instance spreads to the other ones via the DTN carriers. This provides the upper bound for end-to-end web client performance, with the actual performance being dependent on the mobility patterns—i.e., how frequently the web clients meet framework instances.

The system model for the testbed is shown in Figure 9. Our scenario with 30/40 VMs was run on a Dell rack server with Intel Xeon E5-2640 v4 2.2 GHz CPUs with a total of 40 threads and 756 GB of RAM available to the VMs running on OpenNebula. Each DTN carrier VM had 4 GB RAM and 0.5 virtual CPUs, and each web framework VM had 6 GB of RAM and 1 virtual CPU. The DTN carrier VMs were running a SCAMPI instance on a Debian 9.6 distribution, while the web framework VMs ran the full web framework and a SCAMPI instance on a Debian 9.7 distribution. Network characteristics were emulated by instructing the SCAMPI instances to open and close TCP links between each other via a host-local virtual network in real time based on a connectivity trace generated

[4]https://www.seleniumhq.org/

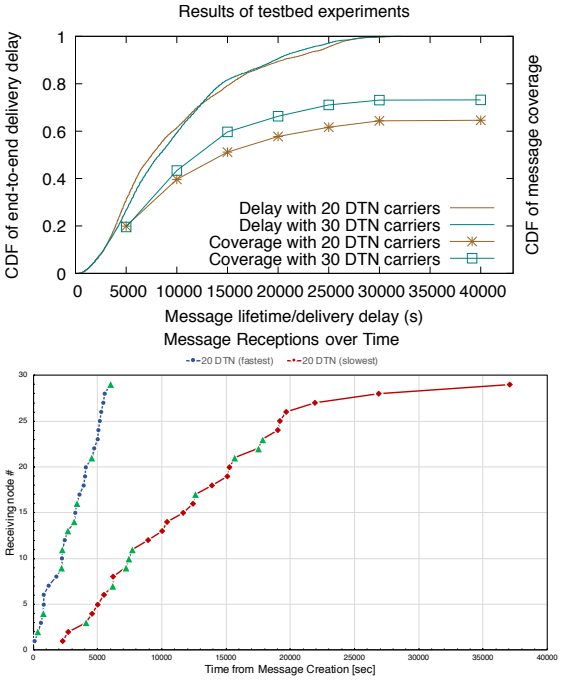


Fig. 10: Results of testbed experiments for 20 and 30 DTN carriers. The top figure shows end-to-end delivery delay experienced by web users and the coverage of APs achieved. The bottom shows an example message dissemination process for the slowest and fastest message to reach full coverage with 20 DTN carriers.

from a simulation. The trace was similar to the ones used in the simulations above, with 10 web framework instances placed in intersections in downtown Helsinki and 20/30 DTN carriers moving according to the SPMBM model for 12 hours. Another host (Intel Xeon E5-2640 v3 2.6 GHz, 32 threads, 132 GB RAM), connected over a wired 10 Gbps link to the emulation host, was used to run the TCP link controller and the Selenium clients to generate HTTP test traffic for the web frameworks. One Selenium client was connected to each web framework instance to publish one Here and Now application message every 10 minutes and to poll for new content every 5 seconds. We base our results on the logs of the Selenium instances, which give us the full end-to-end behavior equivalent to the experience of a web user of the Here and Now application connected to a framework instance.

The results shown in Figure 10 (top) establish an upper bound for the web client performance, i.e., when the content becomes available in each AP—actual performance will be worse since mobile web clients would have to wait to reach an AP to pick up new messages. These are not directly comparable to the simulation cases, as we cannot run as many nodes in the testbed as we can in a simulator, and we also do not emulate web user mobility (only DTN carriers). The results are, however, qualitatively similar to the simulations with the coverage—fraction of APs that receive a message published in another one—stabilizing at $60-80\%$ within $4-7$ hours of message propagation, with a median delay of around two hours. Interestingly, we note that adding carriers does not seem to reduce the delivery delay, but improves the coverage.

In summary, when web users connects to an AP, they see content that is typically two hours old and each AP contains a minority of the total content that has been generated—reinforcing our point from Section II-A that the apps must be designed without assuming global consistency of data. Furthermore, for each individual message, the dissemination pattern varies significantly as shown in Figure 10 (bottom). First, not every message reaches every AP, but for those that do, the delay varies greatly—in the figure we show the fastest message reaching all framework instances in about 100 minutes, while in that time the slowest one has only reached a single AP and taking 300 minutes to reach them all.

Although the above results are strongly determined by the mobility patterns assumed for the DTN carriers, they validate the practical feasibility of our design by demonstrating full browser-to-browser app interactions on real software running on constrained VMs significantly less powerful than typical hardware devices (e.g., smartphones).

VII. Related Work

Our framework borrows concepts from different fields of related work to create a unique combination. Most important is the concept of embedding programs into messages and executing them in network nodes, discussed in the past as active networking [25] and mobile code. Lee et al. [10] present a a node architecture allowing to deploy in-network services in a next generation Internet. Its main contribution is the concept of making the core network become a distributed service execution environment. It also describes an architecture for extensible router allowing for implementing new router features. Similar concepts of extensible router architecture can be also seen in the work of Router Plugins [2]. SOFTNET [29] and PLAN [6] are examples of active networking systems that assume network packets to contain programs, which are used to manage network nodes. All these systems concentrate on executing code inside the network in order to improve network capabilities, while our solution takes advantage of transformation execution to enable content access to web users. Moreover, active networks focus on individual (small) packets, thus limiting the amount of code that can be carried, while our message-based system is not limited by MTU size.

Security aspects of mobile code are covered by Arden et al. [17]. He introduces a new architecture for secure mobile code that developers can use, publish and share mobile code securely across trusted domains. Older work presenting security aspects of mobile code are Rubin et al. [21] and Zachary [28]. Kosta et al. [22] present the concept of using mobile code for improving code execution on mobile devices by offloading part of code execution to the cloud. Similar work by Simanta et al. [24] describe an architecture for offloading mobile code execution in hostile network environments. Unlike these works, our system uses mobile code as a tool for message content presentation and our main concern is offering an isolated execution environment for the application code so

that the code does not harm the node running it (which is similar to protecting routers in Active Networks).

Douceur et al. [5] shows an alternative approach for using native applications in the web browser by the web users. This system requires web servers to have an (application-specific) gateway installed that translates a native app into a web app. Our framework is more flexible, as it carries all transformation code inside the messages themselves so that no node requires prior knowledge of specific applications.

VIII. Conclusion

In this paper, we have presented a DTN based system for enabling modern web applications to be developed for and deployed in opportunistic networks. Our design is based on the idea of distributing small, self-contained pieces of application content directly in the network, rather than sending it to a centralized database, and bundling the presentation and interaction logic as code together with the message. This approach enables browser-to-browser interactions in scenarios without a well-connected infrastructure network.

Beyond the design of our system, we showed and evaluated the practicality of our approach through a prototype implementation on our *Liberouter* platform. Further, we demonstrated how to apply our design to two existing applications to develop their web versions. Finally, we showed through simulations that the overhead imposed by our design is low enough that it does not have a significant negative impact on the message dissemination by the underlying networking platform. We further validated our system by running a series of real-world experiments in the testbed.

There are a number of open issues for our future work. These include enhancing security and privacy functionality of the framework by enabling access to encrypted content for web users and provide support for closed communication groups.

References

[1] J. Corbet. Seccomp and sandboxing. http://lwn.net/Articles/332974/.

[2] Dan Decasper and Zubin Dittia and Guru Parulkar and Bernhard Plattner. Router plugins: A software architecture for next generation routers. In *IEEE/ACM TON*, pages 229–240, 1998.

[3] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl. Network of information (netinf) – an information-centric networking architecture. *Elsevier COMCOM*, 36(7):721 – 735, 2013.

[4] M. Doering, S. Lahde, J. Morgenroth, and L. Wolf. IBR-DTN: an efficient implementation for embedded systems. In *Proc. of ACM CHANTS*, 2008.

[5] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proc. USENIX OSDI*, 2008.

[6] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: A packet language for active networks. In *Proc. of ICFP*, 1998.

[7] T. Kärkkäinen and J. Ott. Liberouter: Towards Autonomous Neighborhood Networking. In *Proc. of IEEE WONS*, 2014.

[8] T. Kärkkäinen, M. Pitkänen, P. Houghton, and J. Ott. SCAMPI Application Platform. In *Proc. of ACM CHANTS*, 2012.

[9] A. Keränen, J. Ott, and T. Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In *Proc. of SIMUTools*, 2009.

[10] Lee, Jae Woo and Francescangeli, Roberto and Janak, Jan and Srinivasan, Suman and Baset, Salman and Schulzrinne, Henning and Despotovic,Zoran and Kellerer, Wolfgang. NetServ: Active Networking 2.0. In *ICC FutureNet IV Workshop*, June 2011.

[11] V. Lenders, M. May, G. Karlsson, and C. Wacha. Wireless Ad Hoc Podcasting. *ACM/SIGMOBILE MC^{2}R*, 12(1), 2008.

[12] J. Morgenroth, T. Pögel, S. Schildt, and L. C. Wolf. Performance Comparison of DTN Bundle Protocol Implementations. In *Proc. of ACM CHANTS*, 2011.

[13] M. Nagy, N. Asokan, and J. Ott. PeerShare: A System Secure Distribution of Sensitive Data Among Social Contacts. In *Proc. of NordSec*, October 2013.

[14] M. Nagy, T. Kärkkäinen, A. Kurnikov, and J. Ott. A Web Browser-based Interaction Framework for Mobile Opportunistic Applications (Full Version). http://arxiv.org/abs/1506.03108, 2015.

[15] M. Nagy, T. Kärkkäinen, and J. Ott. Enhancing Opportunistic Networks with Legacy Nodes. In *Proc. of ACM MobiCom workshop on Challenged Networks (CHANTS)*, September 2014.

[16] A. Narayanan. Social Keys: Transparent Cryptography via Key Distribution over Social Networks. In *The IAB Workshop on Internet Privacy*, 2010.

[17] Owen Arden and Michael D. George and Jed Liu and K. Vikram and Aslan Askarov and Andrew C. Myers. Sharing Mobile Code Securely with Information Flow Control. In *Proc. IEEE S&P*, 2012.

[18] A. K. Pietiläinen and C. Diot. Experimenting with Opportunistic Networking. In *Proc. of the ACM MobiArch Workshop*, 2009.

[19] A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot. MobiClique: Middleware for Mobile Social Networking. In *Proc. of ACM WOSN*, 2009.

[20] T. Reenskaug. Thing-model-view-editor – an example from a planning system. http://heim.ifi.uio.no/ trygver/1979/ mvc-1/1979-05-MVC.pdf, May 1979.

[21] A. D. Rubin and D. E. G. Jr. Mobile Code Security. *IEEE Internet Computing*, 2(6):30–34, 1998.

[22] S. Kosta and A. Aucinas and P. Hui and R. Mortier and X. Zhang. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proc. IEEE INFOCOM*, 2012.

[23] J. Scott, P. Hui, J. Crowcroft, and C. Diot. Haggle: A Networking Architecture Designed Around Mobile Users. In *Proc. IFIP WONS*, 2006.

[24] Soumya Simanta and Grace A. Lewis and Edwin J. Morris and Kiryong Ha and Mahadev Satyanarayanan. A Reference Architecture for Mobile Code Offload in Hostile Environments. In *Proc. of IEEE/IFIP WICSA/ECSA*, 2012.

[25] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. In *Proceedings of ACM SIGCOMM*, 1996.

[26] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical Report CS-200006, Duke University, April 2000.

[27] K. Viswanathan and F. Templin. Architecture for a Delay-and-Disruption Tolerant Public-Key Distribution Network (PKDN). Internet Draft draft-viswanathan-dtnwg-pkdn-00.txt, Work in progress, April 2015.

[28] J. Zachary. Protecting Mobile Code in the Wild. *IEEE Internet Computing*, 7(2):78–82, 2003.

[29] J. Zander and R. Forchheimer. Softnet: An approach to higher level packet radio. In *AMRAD Conference*, 1983.

[30] W. Zhao, Y. Chen, M. Ammar, M. Corner, B. Levine, and E. Zegura. Capacity Enhancement using Throwboxes in DTNs. In *Proc. of IEEE MASS*, October 2006.