Computer Communications 73 (2016) 56–65

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Dynamic framework for building highly-localized mobile web DTN applications

Kartik Sankaran [a,*], Ananda Akkihebbal L. [a], Mun Choon Chan [a], Li-Shiuan Peh [b]

[a] School of Computing, National University of Singapore, Singapore
[b] Department of EECS, Massachusetts Institute of Technology, United States

ARTICLE INFO

Article history:
Available online 7 September 2015

Keywords:
Delay-Tolerant Networks
Smartphone
Web applications
Dynamic framework
Context-awareness

ABSTRACT

Proximity-based mobile applications are increasing in popularity. Such apps engage users while in proximity of places of interest (malls, bus stops, restaurants, theatres), but remain closed or unused after the user goes away. Since the number of 'places of interest' is constantly growing and can be large, it is impractical to install a large number of corresponding native applications on the phone when each app engages the user for only a small period of time.

In this paper, we propose a dynamic framework for deploying highly-localized mobile web applications. Such web applications are deployed locally to users in proximity, and can be opened in the browser. Communication in the web app is performed over the Delay-Tolerant Network of mobile users, removing the need of an Internet connection. DTN protocols can be dynamically added or removed at run-time, allowing each application to use a protocol best suited to its needs. After usage, the web application is closed either manually by the user, or automatically when the user goes away from the place of interest.

To restrict deployment of web apps to only those users in a relevant context (e.g.: people walking nearby a store), and to automatically switch off DTN protocols when user context changes, we have extended the framework to be 'context-aware', using the low-power barometer sensor on the phone to detect when the user is *idle*, *walking*, or in *vehicle*. Applications can specify which protocols to run in these user contexts, and when to switch them off, reducing power consumption.

We have implemented the framework on Android. Our analysis of the framework show that the memory and performance overhead incurred is small. Using this framework, we have written a simple DTN web application for bus-stops to help the physically challenged. Using real-device measurements, we show that adding context awareness to the framework can reduce power consumption by at least 53%.

1. Introduction

Proximity-based mobile applications have recently gained increasing popularity. In these applications, users interact with other users around them. Table 1 lists some of the popular proximity applications, along with the number of users who downloaded the application, and their average rating out of five. Many of these have more than 10 million downloads, and have high user ratings.

The rise of proximity applications has sparked an interest in scalable and energy-efficient device-to-device technologies such as LTE-direct and Bluetooth LE beacons. These technologies are expanding proximity applications to include not just interaction with people, but with physical places as well, such as stores, theatres, and restaurants. For example, users can check for daily specials in nearby restaurants, and movie combo offers in nearby theatres.

Each place of interest is typically associated with its own dedicated application on the phone. Users need to install these apps in order to use them. However, as the number of places of interest grows, installing large number of apps quickly becomes wasteful and annoying to the user. To solve this problem, what is needed is lightweight and convenient installation of proximity applications *when the user is near the place of interest*, in addition to good device-to-device communication. When users go away, the apps should no longer be active nor installed on the phone. This allows users to have highly-localized interactions, with apps engaging users only when necessary, perhaps even only for a brief period of a few minutes.

* Corresponding author. Tel.: +65 92249727.
E-mail addresses: kar.kbc@gmail.com (K. Sankaran), ananda@comp.nus.edu.sg (A. Akkihebbal L.), chanmc@comp.nus.edu.sg (M.C. Chan), peh@csail.mit.edu (L.-S. Peh).

http://dx.doi.org/10.1016/j.comcom.2015.08.017
0140-3664/

Table 1
Examples of social-proximity applications on Android.

Application	Description	Downloads	Rating out of 5
Foursquare[a]	Find interesting places nearby, check-in for discounts	10,000,000+	4.2
Badoo[b]	Chatting, dating, making friends with people nearby	10,000,000+	4.5
Groupon[c]	Finding local deals and discounts	10,000,000+	4.6
Skout[d]	Discovering and meeting new people around	10,000,000+	4.1
Circles[e]	Finding people nearby with mutual interests	1,000,000+	4.5
Sonar[f]	Connect with friends and like-minded people nearby	1,000,000+	4.1
GrabTaxi[g]	Finding and booking nearby cabs	100,000+	4.1

[a] https://play.google.com/store/apps/details?id=com.joelapenna.foursquared
[b] https://play.google.com/store/apps/details?id=com.badoo.mobile
[c] https://play.google.com/store/apps/details?id=com.groupon
[d] https://play.google.com/store/apps/details?id=com.skout.android
[e] https://play.google.com/store/apps/details?id=com.discovercircle10
[f] https://play.google.com/store/apps/details?id=me.sonar.android
[g] https://play.google.com/store/apps/details?id=com.grabtaxi.passenger

One possible solution is to deploy native mobile apps 'on-the-fly' to users who are in proximity to places of interest over device-to-device communication link. This eliminates the need to install apps beforehand and need for Internet connection to the server. Delay-Tolerant Networks (DTN) [1] are best suited to deploy apps since they exploit device-to-device technologies, working in the face of high user mobility. Unlike client-server solutions, DTN does not require an Internet connection to a central server, nor does it need access to a user's location, being inherently locality-specific.

Installation of native apps 'on-the-fly' is however still not lightweight. More importantly, users are wary of giving permission to unknown apps to access their phone's storage and private details. Use of *web applications*, as opposed to native applications, solves this problem as web applications run in the browser's security sandbox. Installation is lightweight since it only involves opening a web page. The browser informs the user when a web app attempts to access private details like location, which can be denied. Users are willing to allow such interactions since it is more akin to browsing a website.

While web apps do not have the full freedom of native applications, they are still quite powerful, having access to location, camera, and even the phone's sensors. However, they are currently limited to communication over sockets. To enable their full potential, web apps *need access to communication over the DTN*, thus making use of upcoming device-to-device technologies.

We propose and implement a dynamic framework for developing and deploying highly-localized mobile web DTN applications [2]. This framework deploys web apps to users near the places of interest. The phone notifies the user of received web apps, and if found interesting, can be opened in the mobile browser. After use, the web app can be closed either manually, or automatically when the user leaves the place of interest.

In this journal paper, we extend the framework developed in [2] to be 'context-aware', using the low-power barometer sensor on the phone to detect when the user is *IDLE*, *WALKING*, or in a *VEHICLE* [3].

Context-awareness reduces power consumption of the framework in the following two ways: First, it *restricts deployment* of web apps to only those users in a relevant context (e.g.: users walking nearby a store). Second, it *enables automatic switching off* of plugged-in DTN protocols when the user context changes (e.g.: when the user gets into a bus). Consequently, context-awareness drastically reduces unnecessary communication between phones, saving power.

We have implemented our framework on Android, and ported it to desktop. It supports both native and web mobile applications. Our analysis of the framework shows that the memory and performance overhead incurred is small. Using real-device measurements, we show that adding context awareness reduces power consumption by at least 53%. In addition, we show via trace-based simulation of real-world public bus transport data that unnecessary communication between phones in a bus is reduced by 87%.

As an example application, we have implemented a simple DTN web app for bus stops to help the physically challenged. The app informs users when buses are arriving at the pick-up point, and is customized to physically challenged users to help them inform bus drivers that they would like to board.

By supporting both Android and web applications, the framework exposes DTN to the large community of developers, making it more likely for DTN applications to be developed for general use. Since protocols are plugged in dynamically, it is easy to modify to adapt to current advances in DTN protocols and device-to-device communication without re-compilation of the framework.

The rest of the paper is organized as follows: Section 2 discusses related work and provides a motivation for our framework. Section 3 describes the design of the framework, while Section 4 extends the framework to be context-aware. Section 5 describes our sample application. Section 6 evaluates the framework. Section 7 discusses future work, while Section 8 concludes the paper.

2. Related work and motivation

In this section, we discuss related work under different categories. By describing their limitations, we also provide motivation for development of a dynamic framework.

2.1. HTTP-over-DTN browsing

Efforts have been made to use DTN for web browsing [11–14]. These papers concentrate on techniques for serving browsing requests over DTN, such as bundling of HTTP requests, pre-fetching, and caching. The underlying DTN is hidden from webpages.

Our framework focuses on deploying DTN web *apps*, as opposed to web pages. Web apps are similar to mobile apps: they are self-contained, i.e. they contain all the scripts and web pages required for the app to work. Also, DTN web apps are fully aware of the underlying DTN, using the DTN API exposed by our framework.

2.2. Web-based DTN apps

Web apps such as Facebook and blogging have been written to use DTN [15,16]. While these apps are 'DTN-aware', the work concentrates on how the apps work using DTN, and does not support localized deployment of web apps and protocols on-the-fly.

2.3. PhoneGap

PhoneGap is a framework for creating cross-platform mobile apps using web technologies. Each app runs in the PhoneGap container, which is essentially a 'super-browser': apps can access phone details

Table 2
Existing DTN frameworks.

Framework	API exposed to developers	Brief description
Haggle [4]	Publish-Subscribe API (attribute-based)	Uses a search-based data-centric protocol
Mist [5]	Publish-Subscribe API (topic-based)	Uses a reliable broadcast with fragmentation
MaDMAN [6]	Sockets API	Switches between TCP/IP and DTN protocol stack
ubiSOAP [7]	Service-Oriented API	Floods WSDL files and SOAP messages
MobiClique [8]	Social-Networking API	Built on top of Haggle
DoDWAN [9]	Publish-Subscribe API (attribute-based)	Floods WSDL files and SOAP messages (with attributes)
Bytewalla [10]	Bundle protocol API	First implementation of the Bundle protocol on Android

(such as user contacts) via PhoneGap, normally not accessible to regular web apps. PhoneGap apps, while written in Javascript, are installed like native apps. The advantage is that several code versions are not required for different mobile platforms.

However, since the app must be installed like a native application, installation of PhoneGap apps do not meet the lightweight and convenience requirements of localized proximity applications. In addition, they lack access to DTN APIs.

2.4. QR codes

QR codes are useful to direct mobile users to web pages online by scanning codes using their camera. While these codes are convenient to post near places of interest, they require users to look for and manually scan the codes. Discovering web apps is not 'automatic' like in our framework.

2.5. DTN middleware for mobile

Several middleware have been written on mobile for development of DTN applications. Table 2 provides a list of existing middleware, along with the type of API exposed, and a brief description of each. To the best of our knowledge, these middleware do not expose their API to web applications (with an exception of Bytewalla, discussed below), limiting their use to native mobile applications only.

In addition, unlike our framework, these middleware are static, i.e. the underlying protocols are fixed at compile-time and shared by multiple applications. It is not possible to load and unload protocols on-the-fly, a feature required by 'use-and-discard' proximity web applications.

2.6. Service-adaptation middleware

The work in [17] proposes a middleware that acts as a bridge between DTN apps written in different languages and DTN bundle service daemons running on different platforms. Bytewalla is the daemon running on Android, while PCs run the DTN2 service daemon. This middleware enables web applications to access DTN. However, like the web-based apps discussed earlier, it does not support localized deployment of web apps and protocols on-the-fly.

2.7. Dynamix

Dynamic frameworks are quite popular in the context-aware computing domain. In particular, a framework called Dynamix [18] provides context-awareness to web applications, by means of context components loaded at run-time. Architecturally, this framework is closest to our framework.

Although architecturally similar, Dynamix focuses on context awareness: its APIs are oriented around receiving 'context events'. In contrast, our framework's (DTN) APIs are communication-oriented. Dynamix's context-aware components are self-contained, while our protocol components are linked in the form of protocol stacks for each application.

To summarize this section, existing work have limitations with respect to the requirements of lightweight and convenient localized DTN web applications. Our framework has been designed to address these limitations and make such dynamic DTN applications possible.

3. Design and implementation

In this section, we give a high-level overview of the design and implementation of our framework. As shown in Fig. 1, it consists of three parts: the framework itself, the deployment application, and the Android/Web applications.

The framework consists of APIs, and protocol components implementing these APIs, all loaded at run-time. To support dynamic loading of code, it uses Apache Felix. It runs as a background (bound) service in Android.

We have written a simple Forwarding Layer API for applications to access routing protocols. This API supports multi-hop message transfers over the DTN. We also have a Link Layer API for one-hop communication, which supports neighbour discovery and connection-oriented communication, implemented by link layer components (Bluetooth, WiFi-direct), and used by forwarding layer components. Dynamically loaded APIs are advantageous since OSGi

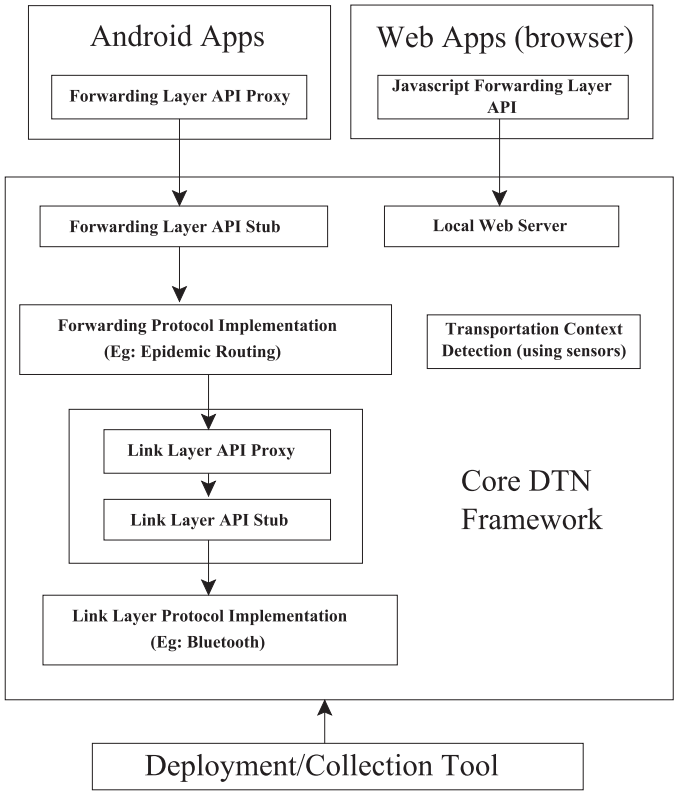


Fig. 1. Design of the framework.

Table 3
Important methods of the Forwarding Layer API.

Method	Description
getDescriptor(appName, userName)	Gets a descriptor for this (appName, userName) endpoint
sendMessage(descriptor, message, destination)	Sends (maybe broadcasts) a message over multiple-hops
addMessageListener(descriptor, listener)	Adds a listener for received multi-hop messages

Table 4
Important methods of the Link Layer API.

Method	Description
getDescriptor(upperLayerID)	Gets a descriptor for this Upper layer ID (similar to EtherType)
addNeighbourListener(listener)	Adds a listener for discovered one-hop neighbours
openConnection(descriptor, neighbour)	Opens a connection to neighbour(s), may be one-to-many
addConnectionListener(descriptor, listener)	Adds a listener for incoming connections
sendMessage(connection, message)	Sends (maybe broadcasts) a message on this connection
receiveMessage(connection)	Blocking receive for a message on this connection

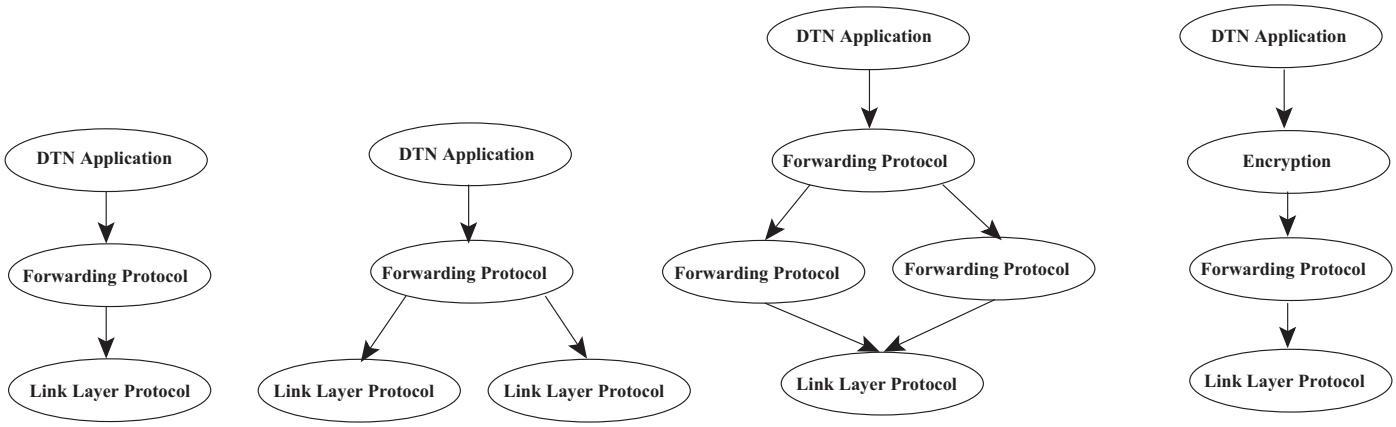


Fig. 2. Some of the possible ways to create a protocol stack.

allows multiple incompatible versions of the API to co-exist without breaking applications.

For clarity, a subset of the methods of these two APIs is listed in Tables 3 and 4. They support the basic send and receive primitives for communication. Although these APIs are low-level, they are powerful enough for developers to write applications.

Although Fig. 1 shows only two protocols and a single protocol stack, the framework supports multiple protocol stacks, with protocols dependencies arranged in a directed acyclic graph (Fig. 2). For example, a forwarding layer can use multiple link layers, and a component can be added for encryption. Every application can load and use its own protocols, or even share protocol stacks.

Protocol components are given a user-readable name in their configuration files. Applications can request for protocols with the specified config name. Changing protocols involves loading a different protocol and giving it the same config name.

When the middleware is first launched, it creates two folders named `install` and `deploy` in the phone's `sdcard`. Protocol and APIs are plugged into the middleware by placing their `jar` files into the `install` folder (To deploy them to *all* phones, rather than just the current phone, the `jar` files are placed into the `deploy` folder). To unload them, the `jar` files are removed. The `install` folder is checked every two seconds by an OSGi utility called FileInstall (The `deploy` folder is checked every two seconds by the deployment tool).

API components are broken into proxy and stub parts, in accordance with Android's inter-process communication (AIDL). The proxy and stub parts contain logic that shields upper layers from change in underlying protocols at run-time by saving state information, and hides underlying AIDL.

The deployment app is a 'special' DTN application that is used to deploy web apps, protocols components (`jar` files), and even native applications placed inside the `deploy` folder (it also supports collection of logs over DTN for debugging purposes). The user is notified of received web apps, which are opened in the browser, while protocol stacks are loaded into the framework.

3.1. Web app support

Web apps are provided with two Javascript libraries `DtnMessage.js` and `FwdLayerAPI.js`. The first contains convenience methods for creating DTN messages, while the second exposes the Forwarding layer API.

The framework runs a local embedded web server which receives DTN API calls from web apps via AJAX, and translates them into corresponding Java calls. To overcome the same-origin policy restriction, the server supports Cross-origin resource sharing[1]. To enable web apps to receive DTN messages, the Javascript code uses AJAX long polling.

The framework runs both on Android and PCs. A subset of the Android libraries were implemented on PC so that it can compile and run largely without modification. The framework currently has full

[1] en.wikipedia.org/wiki/Cross-origin_resource_sharing

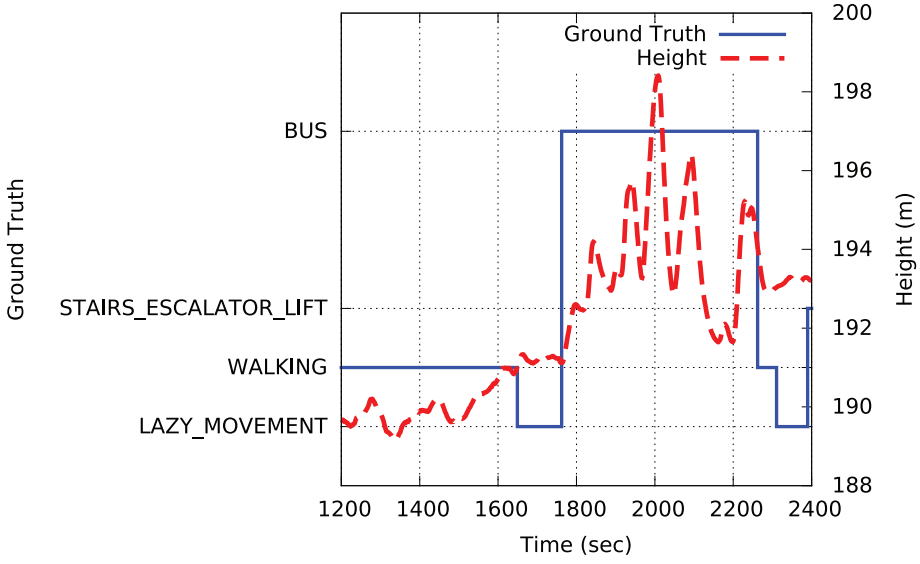


Fig. 3. Variation in altitude measured by barometer in different user states.

support for native Android applications, while web app support is in the prototype stage.

4. Adding context-awareness

4.1. Motivation for context-awareness

In this journal paper, we extend the framework to solve two practical problems faced during on-the-go deployment of web apps. The first problem is that applications are deployed to *everyone* in proximity of a place of interest. Not only does this waste power, but this unnecessarily disturbs users who may not be a suitable target of the application. For example, a shop that wants to advertise an ongoing special sale using a web app would prefer to deploy the app to users walking nearby, but not to users travelling in vehicles on a nearby road.

The second problem is that once users finish interacting with the web application deployed to them, they may forget to close the browser tab (for example by minimizing the browser and switching to another mobile app), leaving the DTN protocols running in the background even when the user goes away from the place of interest, wasting power.

To solve these two problems, we extend the framework by adding context-awareness (specifically, awareness of the user states *IDLE*, *WALKING*, and *VEHICLE*), determined using the low-power barometer sensor on the phone. Context-awareness enables the framework to:

1. **Restrict deployment to relevant users:** Web apps are deployed to only those users in a relevant context. For example, users who have been idle in the same place for a long time, or users travelling in vehicles on nearby roads should not receive web apps.
2. **Automatically switch off DTN protocols:** Once the user goes away from a place of interest, indicated by a change in context (for example leaving the mall by car/bus), the DTN protocols deployed with the application are switched off automatically.

By restricting deployment and switching off protocols, power consumption of the framework is significantly reduced (see the evaluation in Section 6).

4.2. Implementation of context-awareness

We have implemented a low-power context detection of the states *IDLE*, *WALKING* and *VEHICLE* using the barometer sensor on the phone in an earlier work [3]. In this section, we give an overview of how this context detection works, and then describe how it is integrated with the framework.

4.2.1. Context detection using barometer

The three user states *IDLE*, *WALKING*, and *VEHICLE* can be detected using the barometer sensor on the phone. This sensor measures the surrounding air pressure, which can then be translated into height above sea level (altitude). It operates at low-power, and is increasingly available on smartphones today[2]. It is sensitive to even small changes in height, capable of measuring height changes of less than a metre.

We have implemented barometer-based context detection in an earlier work [3]. Here we describe the intuition behind using barometer for context detection, and then describe how it is integrated with the framework to save power.

Fig. 3 shows an example of the variation in altitude measured by the smartphone's barometer in different user states (Note that the term 'Lazy movement' in the figure refers to the user state *IDLE*). As can be observed in this example, users in vehicles typically experience rapid changes in height, since roads are not perfectly flat, and usually follow the terrain of the land. They also experience higher number of peaks and valleys in the barometer signal. On the other hand, users who are walking, due to their lower speed, experience smaller changes in height. Users who are idle experience almost no change in height.

These differing characteristics of height change and number of peaks and valleys in the height signal can be used to distinguish between the states *IDLE*, *WALKING*, and *VEHICLE* using only the barometer sensor on the phone at low-power.

Using barometer for context detection is advantageous over traditionally-used sensors like accelerometer since, by measuring air pressure, it is inherently phone position (hand, bag, or pocket) and orientation-independent, and is unaffected by hand movement. Thus,

[2] Barometer can be found in Galaxy S3/4/5/6, Nexus 3/4/5/6, iPhone 6, and many more

it avoids the confusion cases typically faced by approaches utilizing accelerometer when an idle user moves the phone.

This section only described the intuition behind barometer-based context detection. For additional details, we refer the reader to [3]. In the following section, we describe how we integrate this context detection into the framework to save power.

4.2.2. Integration into the framework

The context detection system developed in [3] is integrated into the framework to smartly manage deployment and protocols to save power. Knowledge of the user states *IDLE*, *WALKING*, and *VEHICLE* is used in the following two ways.

First, based on the user state, the deployment app is switched off. Users who are idle for a long time in the same place, and users travelling in vehicles do not receive on-the-go web apps since deployment is switched off. This saves significant power since people often stay indoors (home/office) for extended periods of time, and there is no need to unnecessarily run the deployment app during those periods. Users in vehicles that rapidly pass by places of interest are also not disturbed by deployed apps.

Second, protocols deployed along with web apps are turned off when the user changes state. For example, if the user forgets to close the web app, and goes away from the place of interest by bus/car, the protocols are turned off to prevent unnecessary communication. Applications can control in which states their protocols should be turned off by modifying their configuration files.

Section 5.1 describes the usefulness of context-awareness with respect to our sample application. Section 6 evaluates the power saved by adding context-awareness to the framework.

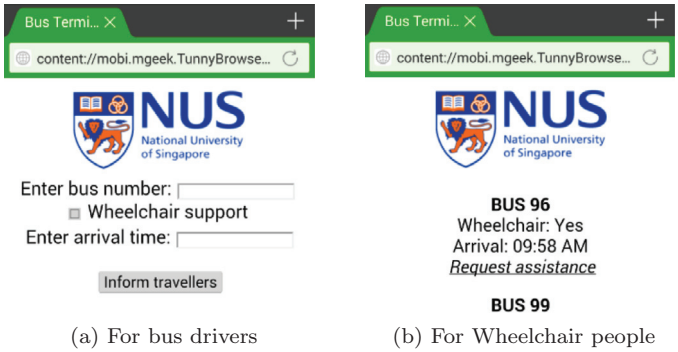
5. Sample DTN web application

To demonstrate the usefulness of localized web apps, and illustrate how these apps work from the user's perspective, we wrote a simple app for bus stops and terminals to help the physically disabled as well as regular commuters board buses. This app is a web version of a DTN Android application written by students of the National University of Singapore (the framework has been used for two semesters by student groups in the Wireless and Sensor Networks course to build DTN apps for project work).

In bus terminals, commuters would like to know when the bus driver has been instructed to go to the pick-up point. Rather than install the LTA (Land Transport Authority) application from the play store beforehand, they can use the web app for a more localized and brief interaction. Physically disabled, such as wheelchair commuters, require assistance to board buses at bus stops and terminals. They need to inform drivers in advance so that they can board first, using a customized version of the app to do this. Our web app uses DTN to enable bus drivers to announce their allotted pick up time, regular commuters to receive this information, and wheelchair commuters to request drivers to assist them while boarding.

Since the framework has been ported to desktop, and supports multiple applications and users on a single device, the web app was developed locally before deploying it to mobile devices. This is especially important since it is easier to debug code using tools available in desktop browsers.

A device (laptop/mobile) located at the bus stop (alternatively can be placed on buses) deploys the web app wirelessly over the DTN to commuters nearby. Users carrying mobile devices running the framework receive the deployed app on-the-fly. In our prototype, received web apps are displayed in the notification bar. If interested, users can open the app in their browser. The user can choose a customized interface: for example, wheelchair people can choose the web app specialized to help them.



(a) For bus drivers (b) For Wheelchair people

Fig. 4. Bus stop web app.

The app for regular users only displays arrival information sent by drivers (Bus driver's interface is shown in Fig. 4a). Wheelchair people have the additional capability to inform drivers in advance that they would like to board, as shown in Fig. 4b. Customized DTN protocols can be optionally bundled with the web app, and plugged into the framework at run-time. After usage (i.e. the commuter has boarded), the app can be simply closed in the browser. The framework automatically releases resources used by the app.

Our sample application demonstrates the advantages of localized web apps: localized interactions, lightweight installation, and secure execution in the browser. Most importantly, these apps exploit device-to-device communication. In the future, our app will be extended to use swipe gestures and audio for the blind.

5.1. Use of context-awareness

Our sample application targets users who are standing near a bus-stop. However, without context-awareness, the app is also unnecessarily deployed to users travelling in vehicles on the road, or to users in nearby home and office buildings.

Furthermore, once the user finishes using the application and gets into the bus, he/she might forget to close the browser tab (by minimizing the browser), leaving the DTN protocols running in the background. As the bus travels from stop to stop and passengers get in and out, the running protocols unnecessarily waste power by communicating with newly boarded users.

These problems are reduced using the context-awareness of the framework. By turning off deployment in phones of users who are in vehicles, or who have been idle for a long period of time in the same place, deployment to these users are avoided. In addition, by specifying that the protocols must be switched off when the user is in a vehicle, the framework automatically switches off protocols once the user has boarded the bus. This prevents unrequired communication, and consequently saves power. Section 6 evaluates the power saved by adding context-awareness to the framework.

Another example of the use of context-awareness is the application *vWant*[3] developed by a student group using the framework to automatically pull music preferences of users sitting around a multimedia screen, and play relevant music based on majority crowd preference. Their application determines whether the user is idle or walking, and does not pull music preferences from those who are walking.

Table 5 gives a summary of the Android applications developed by students using our framework. More detailed description of our students' apps, documentation, APIs, and tutorials are available at the framework's website[4].

[3] Demo video: https://www.youtube.com/watch?v=DAm9gAY_uAo

[4] Website: http://www.comp.nus.edu.sg/~kartiks/nusdtn/

Table 5
Students' applications using the framework (note that these are Android, not web apps).

Application	Description
MaxTix	Last-minute movie ticket sales, re-selling, and ticket transfer
TunePulze	Sharing fitness information and songs during workouts
LiftMeUp	Rapid response and aid to elderly people who have fallen down
DeleCab	Collaboration between users to share the same cab
ChallengeMe	Interactive competitions between people in extreme sports
DisabledPersonTransport	Notifying and helping physically challenged people to board buses
vWant	Multimedia streaming based on crowd-preference
MyRadius	Share and discover information about local events and special sales
SoChat	Share ideas, files, ask questions to students around
WhereToFirst	Collects and displays crowd/queue information of nearby shops

6. Evaluation

In this section, we first compare the use of centralized server versus device-to-device communication with respect to power usage and latency experienced. We then evaluate the performance and memory overhead of our framework. Finally, we analyse the power saved by adding context-awareness to the framework.

6.1. Server versus device-to-device

Existing proximity applications have to use a central server to calculate whether a user is close to a place of interest. The phone uploads its location to the server, which informs it when it is nearby interesting places. Uploading over the cellular network is costly in terms of power. Use of device-to-device technologies can reduce power consumed, but requires periodic 'device discovery'. In this section, using power measurements on the Monsoon power meter, we quantify and compare the power usage of server-based versus device-to-device technologies, and show that there is indeed a power saving in spite of the device discovery process.

The power consumption depends on the frequency of location updates (for server-based solution) and on frequency of device discovery (for device-to-device technologies). The lower the frequency, the lower the power consumed, but at the expense of latency. We assume the device at the 'place of interest' (e.g.: bus stop) is powered externally, and only focus on the user's phone's power usage here.

We consider the case where location updates to the server occur over the LTE network, while the device-to-device technology used is WiFi-direct. Using the Monsoon power meter, we measured the power profile for sending a small (ping) packet to a server on a Galaxy S3 phone, as well as the power profile for device scanning. Table 6 lists the power values measured. Unlike WiFi-direct, LTE suffers from a long tail (more than 12 s) after the packet has been sent.

Based on these measurements, we calculated the power consumption at different frequency of location updates and scans, shown in Fig. 5. For the same latency, server-based approaches would consume higher power. For example, at a (reasonable) 20 s worst-case latency, the power saving of using device-to-device technologies is 86%. Thus, use of device-to-device communication can benefit future proximity applications by being more power-efficient. Although the power to transfer web apps is not included in Fig. 5, we expect that the higher bandwidth between devices would make such transfers faster and lower power than LTE as well.

Table 6
Monsoon power meter measurements.

Operation	Power (mW)
CPU (asleep)	25
CPU (awake)	85
LTE (active)	2000
LTE (tail)	490
WiFi (scan)	300

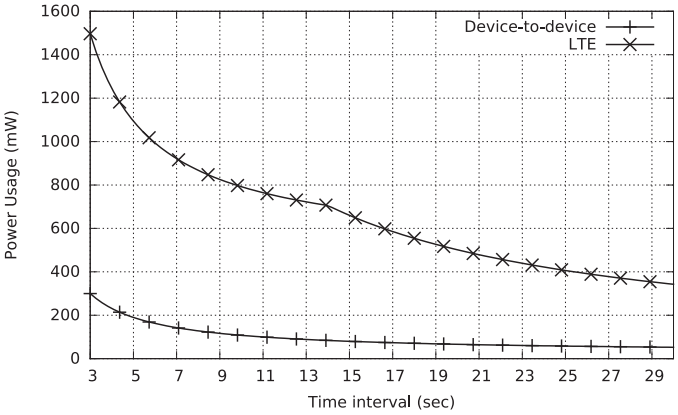


Fig. 5. Power of server (LTE) v/s device-to-device (WiFi).

6.2. Deployment latency

The latency between a device arriving at a place of interest and receiving the deployed web app is important to users. As explained earlier, this is a function of the discovery interval used (set to 10 s in our deployment tool). We measured the deployment latency and found it to be 6.4 s on average, which is reasonable. This can be modified to tradeoff savings in power (Fig. 5).

6.3. Performance overhead

In DTN, devices exchange information when they come into range of one another. It is critical that data is transferred as quickly as possible during the limited contact duration time. Here we measure the performance overhead introduced by the framework during the data transfer.

Two aspects of the framework cause overhead during communication: the Inter-process communication (IPC) where data is copied from the DTN application to the framework, and the API Proxy/Stub (see Fig. 1). We expect the Proxy/Stub overhead to be independent of data size, since it does not involve any data copying. We expect IPC overhead to vary linearly with data size. Note that IPC occurs only when data is initially passed from the DTN app to the routing protocol. After the initial copy, it is buffered in the framework for forwarding to other devices opportunistically.

To reduce IPC overhead for large data (audio, pictures), the framework allows data to be transferred from the app via files in the phone's storage. This removes the need for data copy, and is more convenient for the app. Only (optional) 'metadata' needs to be copied via IPC. For example, a mall application advertising a special sale

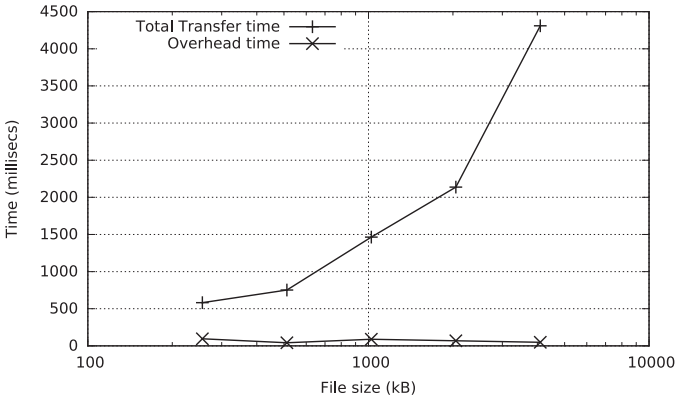


Fig. 6. Overhead during file transfer.

Table 8
Memory overhead.

Part of the framework	Memory
API proxy (application-side)	1.1 MB
Framework service (nothing plugged in)	8.9 MB
Framework service (2 APIs + 2 protocols, no messages)	9.1 MB

Table 9
Accuracy of barometer algorithm versus Google (Accl) and FMS (GPS+Accl) algorithms.

	Baro(%)	FMS(%)	Google(%)	Fusion(%)
Idle	76	33	76	76
Walking	54	46	79	88
Vehicle	81	90	31	77
Overall	69	68	56	81

would transfer product photos via files, while smaller textual data like name and price would be transferred via IPC.

Fig. 6 shows the overhead involved for file transfer between two phones running the framework, using TCP over a 802.11b interface. Each data point is an average of 30 trials. The overhead is small compared to the transfer time, especially for moderate to large file sizes. Table 7 shows a breakdown of the overhead. As expected, Proxy/Stub overhead is independent of data size. IPC overhead is due to the large metadata size (32 kB) used in the experiment, but is independent of the file size. Overhead is 6% and lower for moderate to large file sizes. If IPC is not involved (i.e. data is already buffered), then the overhead is even lower.

6.4. Memory overhead

Here we measure the extra memory used by the framework. In our implementation, the API Proxy class occupies memory in the application memory space. The framework itself runs as a service, and occupies memory separately from the application. Table 8 shows the memory overhead, evaluated using the Eclipse Memory Analyser.

Android imposes a limit on heap, which varies with OS version. Assuming a 32 MB limit, this leaves 23 MB for buffering. If each message is 1 MB, we can buffer 20 messages, which is too few. However, bulk of data is in the form of pictures/audio stored as files on the `sdcard`, and not in heap. The heap contains only the message's metadata. If metadata is 32 kB, the phone can buffer about 700 messages. As newer phones have larger RAM, we do not expect the 9 MB overhead to be significant.

6.5. Evaluation of context-awareness

In this section, we evaluate the context-awareness added to the framework, in the following ways: First, we present results of accuracy of context detection on 47 h. of real-world transportation traces collected from 13 volunteers in 3 countries. Second, using real-device measurements, we analyse the power saved by adding context awareness to the framework to restrict deployment and turn off protocols. Third, using real-world bus transport data, we estimate the reduction in number of users involved in communication by switching off protocols. Lastly, we evaluate the latency of context detection and discuss its impact.

6.5.1. Accuracy of context detection

We have evaluated our barometer-based context detection in our earlier work [3] using 47 h of transportation traces collected during the daily commute of 13 volunteers in 3 countries (Singapore, Boston, and China). Here we present a summary of the results of accuracy of our approach.

We have compared the accuracy of our context detection of the states *IDLE*, *WALKING*, and *VEHICLE* to two other systems: Google's Activity Recognition (which uses accelerometer), and Future Mobility Survey (FMS - a travel survey application deployed to over 1000 users in Singapore and Boston which uses both GPS and accelerometer).

Table 9 compares the accuracy of all three approaches. It can be observed that FMS has good vehicle detection due to its use of GPS, but has poor walking and idle detection due to its low sampling rate of accelerometer (2 Hz). The Google algorithm has good walking detection, but poor vehicle detection, especially on 'smooth' vehicles like subways and trains where engine vibrations are minimal. The Google algorithm's idle detection is good if the phone is still (as was the case in these traces), but has several confusion cases if the phone is moved with the hand [3].

Our barometer algorithm has good idle and vehicle detection, and is resilient to hand movement unlike Google. It also works in vehicles where engine vibrations are minimal. Walking detection is poor on roads with small slope, but can be fixed by fusing with Google's better walking detection or by using low-power step detection chips [3], as shown in Table 9. The fusion algorithm has 81% overall accuracy.

The good idle and vehicle detection of our barometer algorithm makes it suitable for restricting deployment and switching off protocols. For additional results and evaluation of our algorithm, we refer the reader to our earlier work [3].

6.5.2. Power saved using context awareness

As explained previously in Section 4, we add context-awareness to save power by restricting deployment and switching off protocols,

Table 7
Breakdown of framework overhead during file transfer (time is in millisecs).

File size	Metadata size	Proxy	IPC	Stub	Transfer time	Overhead%	Overhead% (no IPC)
256 kB	32 kB	5.66	81.03	8.40	581.12	16.36	2.42
512 kB	32 kB	8.93	25.03	8.20	752.50	5.60	2.28
1 MB	32 kB	6.53	76.09	6.75	1464.98	6.10	0.91
2 MB	32 kB	7.32	25.04	36.18	2137.29	3.21	2.04
4 MB	32 kB	13.47	25.67	8.66	4309.97	1.11	0.51

Table 10
Comparison of power consumption of framework using context awareness versus without context awareness.

	Energy (mJ)	Duration (s)	Power (mW)
Without context awareness (no traffic) [Baseline 1]	84620	1800.42	47
Without context awareness (with traffic) [Baseline 2]	136833	1800.43	76
With added context awareness (with traffic)	39616	1800.72	22

reducing unnecessary phone-to-phone communication. This is particularly significant when users are idle in the same place for extended periods, such as in home or office, and phones should not be running the deployment app and application protocols. However, performing context detection all the time adds to the power consumption of the system.

In this section, using real-device measurements, we evaluate the additional power consumption overhead of running a framework with context detection against a framework running without context awareness, and calculate the energy savings.

Majority of the power consumption in the framework running without context awareness is due to neighbour discovery and phone-to-phone communication. Since the amount of communication involved (and hence the power saving of context awareness) depends on the traffic generated by the mobile web application, we measure the power consumption in two cases when the framework runs without context-awareness: without any communication traffic (only neighbour discovery runs every minute), and with communication traffic (each device sends a 10 kB message every 2 min), which act as two baselines for comparison.

In this experiment, we use 5 Android phones (1 Galaxy S4, 3 Galaxy S3, and 1 Nexus 4) deployed in different locations in a room, with the Galaxy S4 phone connected to the Monsoon power meter. The phones perform neighbour discovery and phone-to-phone communication using Bluetooth LE, while running the framework and a traffic generating application. Other subsystems, including screen and WiFi, are turned off during the experiment. Neighbour discovery is performed once every minute, and in the case where traffic is generated, each device sends a 10 kB message to all other phones every 2 min. Note that using additional phones or higher traffic would only increase the power consumption of the baselines.

We measure the average power consumption over 30 min for each of the following three cases:

1. Framework running *without* context awareness (*no traffic*) [Baseline 1]: Since no traffic is generated, we expect majority of the power consumption to be from neighbour discovery at the link layer protocol.
2. Framework running *without* context awareness (*with traffic*) [Baseline 2]: Since traffic is generated by the application, we expect a larger power consumption due to phone-to-phone communication in addition to neighbour discovery.
3. Framework *with* added context awareness (*with traffic*): We expect the context detection to report that the user is idle, and protocols and deployment to be switched off automatically. So, the power consumption is expected to be largely due to the overhead of context detection.

Table 10 lists the power consumption from measurements on the power meter for the three cases above (excluding the CPU base power of 95 mW).

From the power measurements, we can see that by using context awareness, we save 53% power of the first baseline, and 71% power of the second baseline running without context awareness. Thus, by using context, we achieve significant savings in the power consumption by switching protocols and deployment off, especially important when the user is at home or office for several hours.

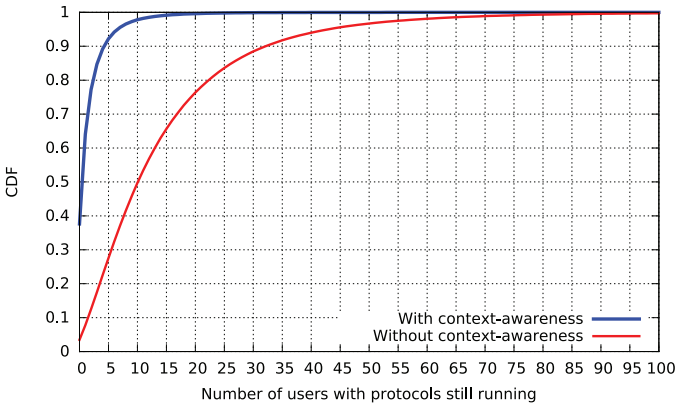


Fig. 7. CDF of number of users on a bus with protocols still running.

6.5.3. Reducing unnecessary communication

While the previous section evaluated the power savings using real-device measurements, in this section we evaluate using trace-based simulation of real world bus transport data how context-awareness reduces unnecessary communication between phones with respect to our sample application, which is deployed to users waiting at a bus-stop.

Once users have finished interacting with the application, they board the bus. However, users may forget to close the application, leaving it running the background while they are in the bus. As the bus travels from stop to stop, and other users get in and out, the protocols unnecessarily waste power by communicating with newly boarded users. If the bus is crowded, this can lead to a large power wastage. However, by automatically switching off protocols using context-awareness, this problem can be avoided.

To analyse the impact of using context-awareness in the real world, we have written a trace-based simulator using a day of real-world transport data from public buses in Singapore, containing data of over 1,000,000 users, and over 2000 buses. The transport data contains the timestamped information of when buses arrive at every bus-stop, and information of users getting in and out (using their transport ezlink card). The simulator uses this data to keep track of which users are present in what buses, and how crowded the buses are at each stop.

Fig. 7 shows the CDF of the number of users in a bus with protocols unnecessarily still running without context-awareness when the bus arrives at a bus-stop. Without switching off protocols, these users unnecessarily communicate with other newly boarded users, wasting power.

Assuming that context-detection kicks in with some delay (see Section 6.5.4), protocols are switched off once the bus starts moving. Fig. 7 shows the CDF of number of users on the bus with protocols still switched on after using context-awareness. We find that the number of such 'active' users is drastically reduced compared to the case without context-awareness. On an average, there is an 87% reduction in number of users involved in unnecessary communication.

Thus, we can see that in a real-world bus system, context-awareness and automatically switching off protocols drastically reduces the unnecessary communication between users in a bus. Since

the simulation does not consider users walking outside the bus, the communication saving is expected to be even larger in the real-world compared to simulation.

6.5.4. Latency of context-detection

While context-awareness can reduce unwanted communication by switching off protocols, the drawback is deployment latency when the user goes out of the house or office, and latency of switching off protocols when the user goes away from the place of interest (with respect to our sample application, the latency of *VEHICLE* detection delays switching off protocols after users board the bus).

We have evaluated our context-detection system using 47 h of sensor traces collected with the help of 13 volunteers from 3 countries during their daily commute [3], and used these traces to calculate the latency of context-detection. The latency of *IDLE*, *WALKING*, and *VEHICLE* detection was found to be 3 min, 2.6 min, and 3.6 min respectively.

The deployment latency of 3 to 4 min when the user goes out of the house is acceptable, since analysis of over 2,256,911 bus trips from the transport data used in Section 6.5.3 shows that the average duration of a journey is 14 min., and can be as large as 156 minutes. Hence, we expect the latency of deployment and switching off protocols to be acceptable, since it is still a fraction of the travel time, especially for long journeys.

7. Discussion and future work

7.1. Use of WebSockets

We will be re-writing our code to use WebSockets, now increasingly supported in mobile browsers, suitable for push-based notifications of messages received, to replace AJAX long polling.

7.2. Security

While web apps run in the browser sandbox, protocols loaded in the framework have dangerous access to Android libraries. In the future, we will use OSGi's fine-grained access control to restrict a protocol's access.

8. Conclusion

In this paper, we proposed a dynamic framework for deployment of localized DTN web apps. The apps free users of the burden of installing multiple native apps on the phone. They are easy to open/close in a browser, and operate only during proximity interactions. To demonstrate their usefulness, we wrote an app for bus stops to help the physically disabled.

We extended the framework to be 'context-aware', to restrict deployment of apps to only those users in relevant context, and to automatically switch off DTN protocols when the user goes away from the place of interest.

Our analysis shows that the framework has low overhead. Using real-device measurements, we show that adding context awareness can reduce power consumption by at least 53%. In the future, we plan to enhance web app support, and add better security to the framework.

Acknowledgments

This research was supported in part by the National Research Foundation Singapore through the Singapore-MIT Alliance for Research and Technology (SMART) program, under grant number R-252-000-551-592.

References

[1] K. Fall, A delay-tolerant network architecture for challenged internets, in: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, in: SIGCOMM '03, ACM, New York, NY, USA, 2003, pp. 27–34, doi:10.1145/863955.863960.

[2] K. Sankaran, A.L. Ananda, M.C. Chan, L.-S. Peh, Dynamic framework for building highly-localized mobile web dtn applications, in: Proceedings of the 9th ACM MobiCom Workshop on Challenged Networks, in: CHANTS '14, ACM, New York, NY, USA, 2014, pp. 43–48, doi:10.1145/2645672.2645675.

[3] K. Sankaran, M. Zhu, X.F. Guo, A.L. Ananda, M.C. Chan, L.-S. Peh, Using mobile phone barometer for low-power transportation context detection, in: Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems, in: SenSys '14, ACM, New York, NY, USA, 2014, pp. 191–205, doi:10.1145/2668332.2668343.

[4] E. Nordström, P. Gunningberg, C. Rohner, Haggle: a data-centric network architecture for mobile devices, in: Proceedings of the 2009 MobiHoc S3 workshop on MobiHoc S3, in: MobiHoc S3 '09, ACM, New York, NY, USA, 2009, pp. 37–40, doi:10.1145/1540358.1540370.

[5] M. Skjegstad, F. Johnsen, T. Bloebaum, T. Maseng, Mist: A reliable and delay-tolerant publish/subscribe solution for dynamic networks, in: 5th International Conference on New Technologies, Mobility and Security (NTMS), 2012, 2012, pp. 1–8, doi:10.1109/NTMS.2012.6208757.

[6] A. Petz, C. Julien, The madman middleware for delay-tolerant networks, in: Poster at HotMobile 2010 (Proceedings of the 11th Workshop on Mobile Computing Systems and Applications), 2010.

[7] M. Caporuscio, P.-G. Raverdy, H. Moungla, V. Issarny, ubisoap: A service oriented middleware for seamless networking, in: Proceedings of the 6th International Conference on Service-Oriented Computing, in: ICSOC '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 195–209, doi:10.1007/978-3-540-89652-4_17.

[8] A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, C. Diot, Mobiclique: middleware for mobile social networking, in: Proceedings of the 2nd ACM workshop on Online social networks, in: WOSN '09, ACM, New York, NY, USA, 2009, pp. 49–54, doi:10.1145/1592665.1592678.

[9] F. Guidec, Y. Maheo, Opportunistic content-based dissemination in disconnected mobile ad hoc networks, in: International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, in: UBICOMM 2007, 2007, pp. 49–54.

[10] H. Ntareme, S. Domancich, Security and performance aspects of bytewalla: A delay tolerant network on smartphones, in: IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2011, 2011, pp. 449–454, doi:10.1109/WiMOB.2011.6085360.

[11] J. Ott, D. Kutscher, "Bundling the Web: HTTP over DTN", WNEPT 2006 Workshop on Networking in Public Transport, QShine Conference, Ontario (2006).

[12] A. Balasubramanian, Y. Zhou, W.B. Croft, B.N. Levine, A. Venkataramani, Web search from a bus, in: Proceedings of the Second ACM Workshop on Challenged Networks, ACM, 2007, pp. 59–66.

[13] J. Chen, L. Subramanian, J. Li, Ruralcafe: Web search in the rural developing world, in: Proceedings of the 18th International Conference on World Wide Web, in: WWW '09, ACM, New York, NY, USA, 2009, pp. 411–420, doi:10.1145/1526709.1526765.

[14] M. Pitkanen, T. Karkkainen, J. Ott, Opportunistic web access via wlan hotspots, in: Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom), 2010, IEEE, 2010, pp. 20–30.

[15] A. Lindgren, Social networking in a disconnected network: fbdtn: facebook over dtn, in: Proceedings of the 6th ACM Workshop on Challenged Networks, ACM, 2011, pp. 69–70.

[16] L. Peltola, DTN-based Blogging, Special Assignment, Helsinki University of Technology, Department of Communications and Networking, 2007.

[17] H. Zhuang, H. Ntareme, Z. Ou, B. Pehrson, A service adaptation middleware for delay tolerant networks based on http simple queue service, in: Proc. of the 6th Workshop on Networked Systems for Developing Regions (NSDR'12), 2012.

[18] D. Carlson, B. Altakrouri, A. Schrader, Ambientweb: Bridging the web's cyber-physical gap, in: 3rd International Conference on the Internet of Things (IOT), 2012, IEEE, 2012, pp. 1–8.