MadApp: A Middleware for Opportunistic Data in Mobile Web Applications

Venkat Srinivasan and Christine Julien
Center for Advanced Research in Software Engineering
The University of Texas at Austin
Email: venkat.s@utexas.edu, christine.julien@mail.utexas.edu

***Abstract*—Mobile computing increasingly often entails applications that embody *opportunistic* or *delay-tolerant* communication, and while much work has focused on refining and optimizing the technical underpinnings for providing delay-tolerant communication constructs, there is almost a complete lack of support for integrating opportunistic communication functionality at the application level. This paper introduces MadApp, an application-level development framework that provides tailored abstractions and support infrastructure for creating dynamic web pages that can incorporate received content from various opportunistic communication channels on-the-fly. We describe multiple application scenarios in which these constructs can seamlessly apply, and provide a complete conceptual and concrete architecture and implementation for MadApp. We evaluate MadApp's support for opportunistic mobile computing web applications using two different mobility trace data sets collected from the real-world. This paper demonstrates that MadApp enables opportunistic mobile computing applications to begin to leverage the significant advances in delay-tolerant communication research, opening doors for even more dynamic and adaptive applications.**

I. Introduction

Imagine you are attending a festival, and before arriving, you download to your mobile device static content such as a map of festival locations or a schedule of events. As the festival unfolds, this static content can be supplemented by user-generated content that your device collects from peer devices at the festival. Such interactions can fill in peers' reviews of festival venues or photographs of shared experiences. Recent commercial efforts[1] support this kind of application using JavaScript enhancements to allow a site's *static* supplemental media (e.g., the images and file downloads within a web site) to be distributed among a traditional peer-to-peer network, but these approaches fail when the network exhibits canonical delay-tolerant network characteristics, such as a high churn in connectivity, highly dynamic sets of available peers, and unpredictable and large delays. However, many of today's mobile computing applications implicitly incorporate some form of *delay-tolerance*, in which data, connectivity, or some other resource is available only after a user-perceptible delay. Significant research attention has focused on the technical components associated with getting delay-tolerant communication to work, specifically focusing on strategies for forwarding, caching, and routing data in these inherently intermittently connected environments [3], but it is increasingly evident that applications are not being developed to take advantage of these new styles of communication.

[1]e.g., peerCDN (https://peercdn.com/), using WebRTC (http://webrtc.org/)

We introduce MadApp, which bridges the gap between expressive communication capabilities for delay-tolerant or opportunistic mobile computing networks and applications, which, to date, remain largely inflexible and delay-*intolerant*. MadApp's key innovation is to enable the content available to a web application to be received across multiple *delivery vectors* without the application having to handle or even be aware of the different vectors. MadApp uses widely adopted web application strategies to support collecting the content from the multiple delivery vectors, and MadApp provides intuitive programming constructs that focus on enabling average programmers to create these highly flexible applications. MadApp layers on top of existing content delivery strategies and, as data arrives, demultiplexes it to already executing web applications.

Paper Overview. MadApp supports the development of delay-tolerant applications, specifically user-facing web applications. While its conceptual model is platform independent, our implementation of MadApp is integrated with the Django web framework, making it simple and straightforward for application developers to integrate delay-tolerant delivery paradigms into otherwise ordinary web applications. We describe MadApp's context within existing work (Section II). Then we describe the conceptual model that underlies MadApp (Section III) and give the details of the MadApp implementation (Section IV). We demonstrate MadApp's use with a case study (Section V) and evaluation over traces of real mobile computing network interactions (Section VI). We demonstrate that, by incorporating existing delay-tolerant network data dissemination approaches, MadApp can support today's opportunistic mobile computing application needs, connecting the applications to dynamic, user-generated content seamlessly and flexibly. In a simple, streamlined, and intuitive way, MadApp brings the potential of opportunistic data networks to real, user-facing web applications, both decreasing the burden for web content delivery experienced by potentially overloaded "centralized" servers and making it possible to receive real-time, dynamic data directly from peers in an opportunistic mobile computing environment.

II. Related Work

In mobile networks, content downloading can be enabled using existing infrastructure supplemented by opportunistic peer-to-peer interactions in a delay tolerant network (DTN) [10], [15]. Work specifically targeted at enabling web browsing has used, for example Internet proxies and prefetching [4] and, more recently, supplementing these proxies using mobile-to-mobile interaction [1]. Theoretical results have shown that relaying content through DTN nodes can significantly increase the system's overall throughput [13].

Most research in DTNs focuses on one aspect or another of how to get packets from a source to a destination. Middleware efforts generally focus on encapsulating intelligent forwarding and routing paradigms [14] or on providing abstractions of these communication primitives (e.g., in the publish-subscribe paradigm [12]). Other efforts have adopted a *content* abstraction [6], which is important for applications in general, but, to date, efforts have still not incorporated these content abstractions into *programming* abstractions to support the development of applications. The DTN Service Adaptation Middleware (DSAM) [22] does specifically attempt to ease the programming burden associated with DTN applications; DSAM is effectively an *adaptor* between a DTN daemon and the application layer. This middleware shares our application level motivation but focuses on connecting layers on a single device, across programming languages instead of connecting across devices via shared content.

Prior work has enabled peer devices to directly exchange streaming media between devices in peer-to-peer networks [19]–[21]. The focus has been on how to *route* data; further, the approaches assume a traditional network, where delays are small and manageable; even the routing strategies are not appropriate for DTNs, whose data needs are very different. Cabernet [8] more directly addresses the needs of intermittently connected web clients by developing constructs to set up content delivery connections over dynamic wireless connections, even when those connections experience high churn caused by extreme mobility. Approaches like Cabernet are orthogonal to our goals; integrating a Cabernet-style approach to managing connections would only magnify the benefits of MadApp's application-level abstractions.

SCAMPI [17] uses social interactions to enable shared usage of opportunistically available resources. SCAMPI's focus is orthogonal to ours: SCAMPI abstracts the opportunistically available services into a "service layer," including integrating aggregate primitive service functionality into more sophisticated emergent services. In the sense that these services are just opportunistically available resources, MadApp could be combined with SCAMPI to enable novice programmers to integrate services available in SCAMPI's opportunistic service layer into their MadApp enabled web applications. Other work has enabled limited capabilities for rapidly prototyping applications [9]; our approach is similar in structure but focuses on refining a usable and accessible set of programming abstractions for real deployed applications.

III. MadApp: Conceptual Model

The MadApp middleware that aids developers of web-based applications in naturally integrating DTN interactions into otherwise ordinary webpages. Fig. 1 shows the conceptual model underlying MadApp.

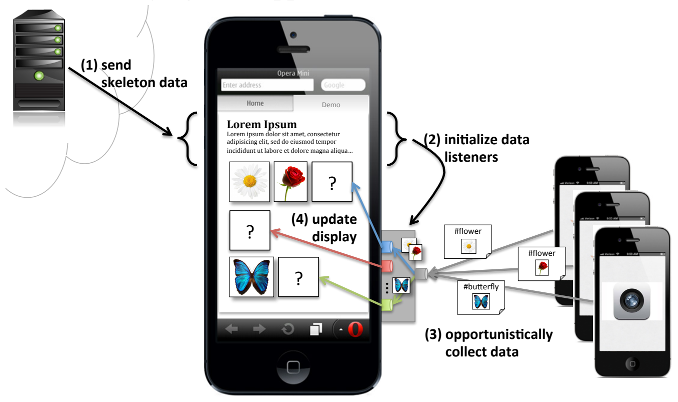

Fig. 1. MadApp Conceptual Model

Loading a MadApp enabled web application on a mobile device triggers the following sequence of events. (1) A web server sends an apparently ordinary webpage to the mobile device; this page includes "holes" for transiently received data and event listeners that receive and fill in data dynamically. (2) Upon receiving the skeleton webpage, the mobile device displays it; the device subsequently initializes the data listener(s) that will receive and dynamically fill in the holes according to the application logic embedded in the skeleton webpage. (3) As the user moves through a dynamic mobile computing space, the device may encounter data that can fill the specified holes; the listeners opportunistically collect this data. (4) Received data is filled in on the dynamically displayed webpage as dictated by the page's application logic.

Building a MadApp Enabled Web Application. From the developer's perspective, constructing a MadApp enabled web application requires two steps in addition to traditional tasks: identifying "holes" in the webpage that are expected to be filled in later, which includes identifying the "types" of the expected information, and implementing the logic for how content should be reflected in the webpage as it arrives dynamically. MadApp allows the developer to delegate all of the data and network connectivity handling to the middleware.

A MadApp enabled web application must identify (i.e., register for) the types of opportunistically received content of interest. We assume that all content items are labeled with "hash tags"[2]. Along with each of the webpage's registered hash tags, the developer defines a listener that is invoked when content matching the registration is received from peers encountered opportunistically in the mobile computing space. The body of this call back defines whether and how received data is incorporated into the application. At the most basic level, a MadApp enabled web application can simply add the received content to the displayed page (e.g., by updating the web document definition), changing the user's view of the displayed information, but the action taken can be anything within the application's purview.

[2]This simple mechanism can easily be exchanged for a more sophisticated and potentially semantically hierarchical naming system without impacting any other functionality in MadApp

Enabling MadApp's Listeners. MadApp relies on a local *neighbor discovery service* that hands the identities and connection information of discovered peers to MadApp, which creates content listeners that wait to receive incoming content from connected peer devices. As a webpage is loaded on a client device, it registers interests in specific content, and MadApp connects these registrations to the peer listeners by filtering any opportunistically received data and forwarding "matches" to registered pages. Fig. 2 shows the sequence of steps that get a MadApp enabled web application up and running. The init steps initialize the content event streams via the neighbor discovery service and the MadApp enabled web application received from an external web server and launched on the client device. The latter init action relies on MadApp's `addEventListener` function to initialize data structures in the MadApp middleware that connect ⟨#type, callback⟩ pairs and enable MadApp to forward received content to registered web applications. Steps 1 through 4 in Fig. 2 follow the path of opportunistically received content through the MadApp middleware to the web application.

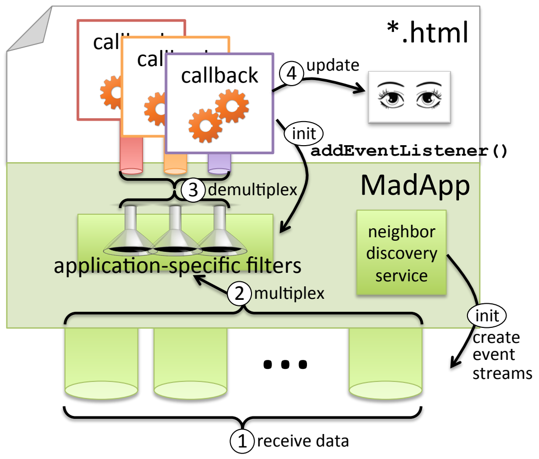

Fig. 2. The MadApp client-side architecture.

Opportunistic Data Collection. Upon arrival at the client device, a MadApp enabled web application is incomplete, but logic embedded in the webpage dictates how the missing information should be filled in as the client device interacts with discovered peers via direct connections. These peers are depicted at the right side of Fig. 1. These peer devices run MadApp *push servers* that are connected into the client-side MadApp middleware via the neighbor discovery service; as these content providing push servers have data available to share with peers, they generate events that encapsulate the content and stream into the MadApp middleware on the client side. These events arrive through the pipes depicted at the bottom of Fig. 2. MadApp multiplexes these received event streams onto its application-level filters (step 2 in Fig. 2). If a piece of received content matches a #type for which a live web application has registered, MadApp invokes the associated callback, passing the received content into the application layer (step 3).

On the content providing device, the push server is a dedicated application that collects content and shares it in a local area via direct peer to peer connections. The nature of this application is one of crowd-sourcing or crowd-sensing, where the data may be generated automatically by peer devices (e.g., automatically sensed environmental data) or at a more human time-scale (e.g., eye-witness photos taken by people nearby an unfolding event). As such, the push server application can be a user-interactive one or an application that runs "in the background." We provide additional details on our specific prototype push server in the next section.

For our prototype implementation and case study demonstration described in the subsequent sections, we assume a MadApp client device can collect information from "one-hop" neighbors, i.e., neighbors to which the client device is directly connected via a wireless link. However, MadApp is itself independent of the underlying networking implementation. Therefore, the MadApp model and middleware implementation can seamlessly integrate with more advanced delay-tolerant routing approaches (e.g., those that enable multi-hop forwarding or those that implement probabilistic routing based on context) to distribute content even more widely.

In this paper, when we refer to MadApp clients and MadApp push servers, we speak of disjoint sets of devices. In the typical deployment we envision, however, the average device serves as both a client device (a consumer of information) and a push server device (a provider of locally relevant content). These two functions are completely independent so we treat them as such in the remainder of this paper.

IV. Implementation

Our implementation of MadApp is based on a combination of javascript, the Django web framework[3], and Server-Sent Events (SSE) [11]; we use the python implementation of SSE[4]. We build on the widely used Django web framework to enable a smooth transition for otherwise ordinary web developers. In our prototype implementation, we demonstrate the capabilities of MadApp by assuming a statically defined set of peer devices. We do not assume that the client device is persistently connected to all of the peers; instead our MadApp implementation connects only to peers in this static set as they become available (and can disconnect and reconnect to them as mobility dictates). In general, we assume the availability of approaches already described in the literature that can discover available connected devices in the mobile computing environment, e.g., via bluetooth or wifi connections [7], which can easily replace our static neighbor definition component.

Fig. 3 depicts the architecture of a complete MadApp deployment, including the *client*, which initially requests a webpage and opportunistically receives content to fill holes in that web application, the *HTML server*, which provides the basic MadApp enabled web application, and the *HTML push server*, which runs on peer devices that the client opportunistically discovers in the mobile computing environment. We omit the implementation details for brevity, but the complete details

[3]https://www.djangoproject.com
[4]https://github.com/niwibe/sse

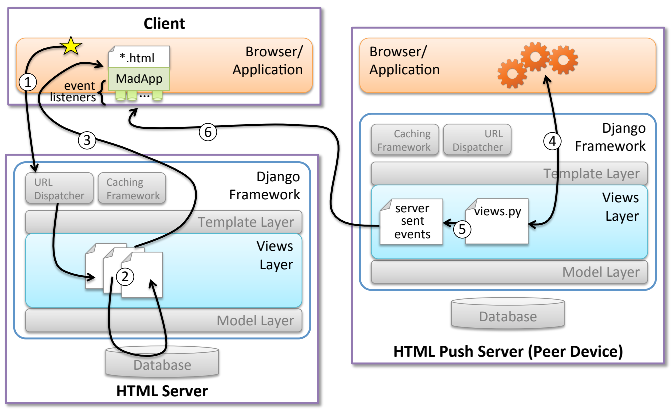

Fig. 3. Architecture of Django-based implementation. A client requests a MadApp enabled webpage (step 1), which is retrieved via Django in the traditional way (step 2). The webpage is delivered to the client and loaded on top of MadApp (step 3). Asynchronously, peer devices' push servers generate content, which is passed into the peer device's Django views layer (step 4), encapsulated in a server-sent event (step 5) and sent to the client device using the sse implementation (step 6).

can be found in [18]. We have implemented the architecture for both a full-fledged browser (e.g., on a laptop) and on Android.

V. Case Study

To study the use of MadApp and to evaluate the performance of its ability to opportunistically deliver dynamic content in a variety of opportunistic mobile computing scenarios, we implemented a simple MadApp enabled web application that opportunistically collects and dynamically displays photos from nearby tourists depicting local sights.

The client's web browser retrieves the basic page from the HTML server and loads it for display into the browser. The page that is initially retrieved contains simple plain formatted text and the code that interacts with MadApp to prepare to receive the photographs opportunistically. The webpage's callback is the simplest possible use of MadApp; it simply updates the displayed webpage by appending any received photos. The code of this callback is shown in Fig. 4; `source` is an `EventSource`, connected to an application-level MadApp event stream. The webpage's state includes an array of received photos (`imgs[]`), and the number of received images (`numImgs`). The received content is added to the list of images, and, after preparing the received image for display, the image is added to the body of the webpage.

```
source.addEventListener("touristPhoto",
                        function(e) {
  indexof=(indexof+1)%numImgs;
  imgs[indexof]=new Image();
  imgs[indexof].src=e.data;
  imgs[indexof].setAttribute("width", "20%");
  document.body.appendChild(imgs[indexof]);
  events_dom.html("<div>" + e.data + "</div>");
});
```

Fig. 4. MadApp Example callback. The callback ("`function`" in the figure) simply appends the received photo on the displayed webpage.

On the push server side, we implemented multiple options. In our Windows deployment, we developed a user facing application in C# that allows a user to select a photo from the file system to "share." When the user selects the photo, the push server implementation immediately pushes the content to any client that is connected to the push server and has loaded the webpage in its browser. Recall that loading a MadApp enabled webpage caused the client to "register" to receive the photo. For testing and evaluation purposes, we also created an automated content generator that periodically (according to a specified schedule) creates data and shares it with the push server implementation. Finally, we created an Android version of the push server that runs a user-facing file browser in QPython on the Android operating system.

To accomplish the described functionality, every push server implements the code shown in Fig. 5 within the `views.py` definition in the views layer of the Django framework.

```
...
myopen=open(fileURL)
self.sse.add_message("touristPhoto",
                     unicode(myopen.readline()))
...
```

Fig. 5. MadApp push server functionality. This code snippet assumes that `fileURL` contains a local handle to the photograph to be pushed; this file is created by the user-facing application when the user selects a photo to send.[6]

We have used this case study to demonstrate the capabilities of MadApp in supporting opportunistic content collection for delay-tolerant mobile computing applications by executing the entire deployment on a variety of types and numbers of devices, connected over a mobile ad hoc network in which peer devices can join and leave the network, generating content as they are available. These efforts demonstrate MadApp's feasibility and potential usefulness. In the next section, we also use this implementation to evaluate the *performance* of MadApp with respect to its ability to distribute opportunistically shared data in a pair of real-world scenarios.

VI. Evaluation

We used MadApp with our case study to evaluate the enabled opportunistic data sharing in two real-world opportunistic mobile computing scenarios. We used two connectivity traces from the CRAWDAD repository[7] to dictate the one-hop connectivity among peer devices.

Evaluation Setup. The first data set, St. Andrews [2], was generated from 27 study participants at the University of St. Andrews (undergraduate students, postgraduate students, and staff members) with mobile devices. The participants carried the devices for 79 days, during which time the devices collected *contacts* consisting of the encountered device, the start time of the encounter, and the end time of the encounter.

Our second data set, SIGCOMM [16], was generated during the SIGCOMM conference in 2009, from 76 conference participants with mobile devices that collected their pairwise interactions over the conference. A contact consisted of the encountered device and a timestamp. We used these timestamps as the initiation of a contact and filled in the duration of the contact according to a normal distribution with a mean of 30

[6]Due to idiosyncracies of our implementation, we convert image files to base64 data before sending and convert the data back to images on the client.

[7]crawdad.cs.dartmouth.edu

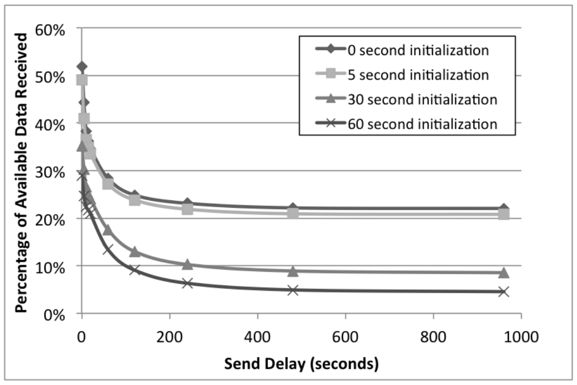

Fig. 6. Percentage of Available Content Received at Client

minutes and a standard deviation of 15 minutes to mimic a conference scenario, where contacts are likely to be of a long duration (e.g., the length of a "session" of the conference).

A single "run" randomly selects a device in the trace file to be the client, which requests the webpage from an always available web server. A randomly selected subset of the remaining devices are peer push servers. The number of push servers used varies between runs. Each push server made available at most 50 tourist photos that were shared opportunistically as the push server encountered the client device as governed by the contact trace.

We generated photos on push servers to mimic automatically generated and human-generated data. Each run is parameterized by a *send delay*, which specifies the delay between sending two photos. If the send delay is 0, all content is available at the beginning of the scenario, so any delays in receiving content at the client are governed purely by contact availability. A run is also parameterized by an *initialization delay*, which models the amount of time it takes to set up a connection between peers. The initialization delay captures an application delay that might be incurred, for example, if the user is required to give permission for the devices to pair.

Results. We measure and report two types of data. First, we report the amount of the content that the client device was able to receive and display. We compute both the raw number of content items the client receives and the percentage of the total available content that is received. We also plot the cumulative acquisition of data, measured at 60 second intervals, over the entire period of the contact trace. We average across 10 unique runs, consisting of a different device acting as the client and a different selection of push servers.

Fig. 6 shows the results for the first metric as we vary our automator's *send delay*. The four curves depict four different *initialization delay* values; due to space limitations we show only the chart for the St. Andrews data set, in which we had a single client and 20 push servers. The amount of content received by the client when both the send delay and initialization delay are set to 0 is effectively the best possible outcome. Any data not delivered in this scenario was not delivered not because of MadApp or its networking behavior, but simply because the contact opportunities did not allow the push server holding the data to meet the client and transmit the files. The results in Fig. 6 show that increasing the *initialization delay* has a significant impact on the performance in the St. Andrews traces (this was not the case at all in the SIGCOMM traces). This is due to the highly transient nature of the St. Andrews traces. When the contact opportunities are fleetingly brief, any time spent kickstarting the exchange is time that is not spent exchanging content, resulting in a dramatic decline in the percentage of content the client receives. This has no statistically significant impact in the SIGCOMM traces, simply because the contact durations are so much longer that the initialization delay (even at 60 seconds) is a very small fraction of the contact interval and therefore does not significantly disrupt the exchange of content.

The results for the St. Andrews data set also show a surprisingly steep decrease in the percentage of content received as the *send delay* increases. This is a further testament to the transient nature of the opportunistic contacts. As the send delay increases, MadApp does not have all of the content available from the beginning of the trace, so contact opportunities, especially those earlier in the trace, are potentially *lost* opportunities to share content that has not been generated yet.

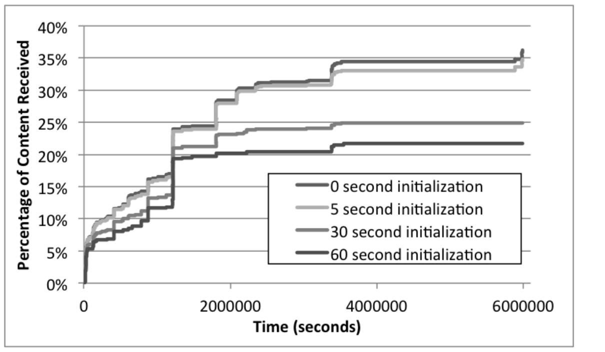

Fig. 7. Cumulative Data Acquisition (St. Andrews data set, 15sec. send delay)

To get a better picture of the collection rate of the opportunistic content, we also look at plots of the cumulative acquisition of the available data. Fig. 7 shows this metric for the St. Andrews data set, given a 15 second send delay and varying initialization delays. Fig. 8 shows this metric for the SIGCOMM data set, using a 60 second send delay; the four different curves in Fig. 8 correspond to varying numbers of push servers available in the mobile computing environment.

In the St. Andrews data trace (Fig. 7, which executed over 79 days (the first 70 days are shown in the figure), a very large portion of the content that was ultimately delivered was delivered in the first three weeks. Increments after this time period indicate data that was generated early on (again, within the first few hours of the execution) and was carried around by the push servers for days or weeks before the push server happened to encounter the client and deliver the content. The usefulness of content received this long after its generation time is clearly application dependent. Concretely, in our specific example, before 2 hours completed in the St. Andrews trace, the client, on average, received not only the static web page content but also between 9.6 and 14 live tourist photos supplementing that static content and shared locally by

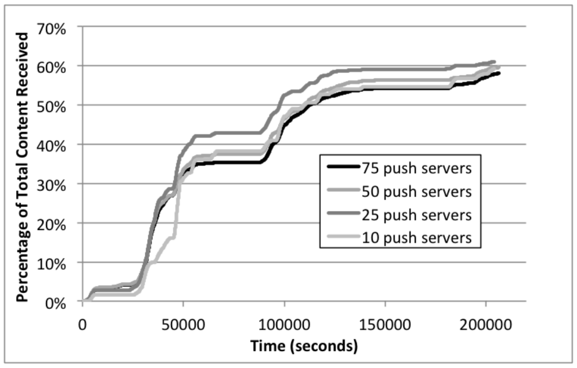

Fig. 8. Cumulative Data Acquisition (SIGCOMM data set, 60sec. send delay)

peers in the mobile computing network.

The SIGCOMM trace shows a more gradual accumulation of content (over the course of the two day conference). Flat periods in the graph indicate either that no contacts were made during this time (e.g., it was nighttime) or that no novel contacts were made. The results in Fig. 8 show no significant difference in the *percentage* of total available content delivered when the number of push servers increases; obviously, when there are more push servers generating content, there is more content to be collected. While the percentage of content collected is the same for different numbers of push servers, the *diversity* of that information is higher when there are more content sharers; this is a boon for the types of applications MadApp targets. In these scenarios, the first tourist photo always arrived within the first hour (after an average of 57 minutes for 10 push servers), but came, on average, within the first seven minutes when there were 75 push servers. This "first content" metric measures the user's perceived "responsiveness" of the webpage; in this regard, MadApp strikes a balance between traditional (nearly instantaneous) web browsing and existing asynchronous browsing of approaches [5].

VII. Conclusion

We introduced MadApp, a middleware for supporting development of applications that incorporate inherently delay-tolerant data in mobile computing environments. We outlined MadApp's conceptual model and implementation and used a case study to demonstrate the feasibility and potential performance aspects of MadApp. Applications built on MadApp can accomplish myriad tasks using the dynamic content they receive from connected peers. For now we have focused on the application-level display of webpages and the updating of those webpages to reflect newly arriving content. Other application capabilities can easily be layered on top of MadApp, taking advantage of all of its dynamic content delivery aspects and just doing something different with the content when it arrives. This simply requires "routing" the incoming data streams appropriately on the local device. While existing work has made opportunistic mobile computing applications technically feasible, MadApp is a key step in fully realizing these applications by augmenting the traditional networking focus with a focus on equally important application abstractions.

Acknowledgments

This work was funded, in part, by the National Science Foundation (NSF), Grant #CNS-0844850 and the Department of Defense (DoD), Grant #H98230-12-C-0336. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies. The authors would like to thank Agoston Petz for his initial work on the project and input on the implementation.

References

[1] A. Balasubramanian, B. Levine, and A. Venkataramani. Enhancing interactive web applications in hybrid networks. In *Proc. of MobiCom*, pages 70–80, 2008.

[2] G. Bigwood, D. Rehunathan, M. Bateman, T. Henderson, and S. Bhatti. Exploiting self-reported social networks for routing in ubiquitous computing environments. In *Proc. of WiMob*, pages 484–489, 2008.

[3] Y. Cao and Z. Sun. Routing in delay/disruption tolerant networks: A taxonomy, survey and challenges. *IEEE Communications Surveys and Tutorials*, 15(2):654–677, 2013.

[4] S. Chava, R. Ennaji, J. Chen, and L. Subramanian. Cost-aware mobile web browsing. *IEEE Pervasive Computing*, 11(3):34–42, 2012.

[5] J. Chen, S. Amershi, A. Dhananjay, and L. Subramanian. Comparing web interaction models in developing regions. In *Proc. of DEV*, 2010.

[6] L.-J. Chen, C.-H. Yu, C.-L. Tseng, H.-H. Chu, and C.-F. Chou. A content-centric framework for effective data dissemination in opportuntistic networks. *IEEE J. on Selected Areas in Communications*, 26(5):761–772, 2008.

[7] T. Clausen, C. Dearlove, and J. Dean. Mobile ad hoc network (manet) neighborhood discovery protocol (nhdp). IETF RFC 6130, http://xml2rfc.tools.ietf.org/html/rfc6130, 2011.

[8] J. Eriksson, H. Balakrishnan, and S. Madden. Cabernet: Vehicular content delivery using WiFi. In *Proc. of MobiCom*, 2008.

[9] A. Goliath, J. A. B. Link, and K. Wehrle. Demo: A versatile architecture for DTN services. In *Proc. of ExtremeCom*, 2012.

[10] B. Han, P. Hui, V. Kuman, M. Marathe, J. Shao, and A. Srinivasan. Mobile data offloading through opportunistic communications and social participation. *IEEE Trans. on Mobile Computing*, 11(5):821–834, 2012.

[11] I. Hickson. Server-sent events. *W3C Candidate Recommendation CR-eventsource-20121211, http://www.w3.org/TR/eventsource*, 2012.

[12] P. Jiang, J. Bigham, E. Bodanese, and E. Claudel. Publish/subscribe delay-tolerant message-oriented middleware for resilient communication. *IEEE Communications Magazine*, 49(9):124–130, 2011.

[13] F. Malandrino, C. Casetti, C. Chiasserini, and M. Fiore. Content downloading in vehicular networks: What really matters. In *Proc. of INFOCOM*, pages 426–430, 2011.

[14] A. Petz and C. Julien. An adaptive middleware to support delay-tolerant networking. In *Proc. of ARM*, pages 17–22, 2008.

[15] A. Petz, A. Lindgren, P. Hui, and C. Julien. MADServer: A server architecture for mobile advanced delivery. In *Proc. of CHANTS*, 2012.

[16] A.-K. Pietilänen and C. Diot. Dissemination in opportunistic social networks: the role of temporal communities. In *Proc. of MobiHoc*, pages 165–174, 2012.

[17] M. Pitkänen et al. SCAMPI: Service platform for social aware mobile and pervasive computing. *ACM SIGCOMM Computer Communication Review*, 42(4):503–508, October 2012.

[18] V. Srinivasan and C. Julien. Madapp: A middleware foropportunistic data in mobile web applications. Technical Report TR-UTARISE-2014-02, The Center for Advanced Research in Software Engineering, The University of Texas at Austin, 2014.

[19] D. Tran, K. Hua, and T. Do. ZIGZAG: An efficient peer-to-peer scheme for media streaming. In *Proc. of INFOCOM*, pages 1283–1292, 2003.

[20] J. Wu, Z. Lu, B. Lu, and S. Zhang. PeerCDN: A novel P2P network assisted streaming content delivery network scheme. In *Proc. of ICCST*, pages 601–606, 2008.

[21] M. Zhang, J.-G. Luo, L. Zhao, and S.-Q. Yang. A peer-to-peer network for live media streaming using a push-pull approach. In *Proc. of MM*, pages 287–290, 2005.

[22] H. Zhuang, H. Ntareme, Z. Ou, and B. Pehrson. A service adaptation middleware for delay tolerant network based on HTTP simple queue service. In *Proc. of NSDR*, 2012.