HINT: from Network Characterization to Opportunistic Applications

Gwilherm Baudic
gwilherm.baudic@isae.fr

Antoine Auger
antoine.auger@isae.fr

Victor Ramiro
victor.ramiro@isae.fr

Emmanuel Lochin
emmanuel.lochin@isae.fr

Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-SUPAERO)
Université de Toulouse, 31055 Toulouse Cedex 4, France

ABSTRACT

The increasing trend on wireless-connected devices makes opportunistic networking a promising alternative to existing infrastructure-based networks. However, these networks offer no guarantees about connection availability or network topology. The development of *opportunistic applications*, i.e., applications running over opportunistic networks, is still in early stages. One of the reasons is a lack of tools to support this process. Indeed, many tools have been introduced to study and characterize opportunistic networks but none of them is focused on helping developers to conceive opportunistic applications. In this paper, we argue that the gap between opportunistic applications development and network characterization can be filled with network emulation. As proof of concept, we propose and describe HINT, a real-time event-driven emulator that allows developers to early test their opportunistic applications prior to deployment. We introduce the architecture and corresponding implementation of our proposal, and conduct a preliminary validation by assessing its scalability.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Store and forward networks*

Keywords

DTN, Opportunistic networks, Emulation, Architecture

1. INTRODUCTION

Opportunistic networks are a special case of DTNs [7] where nodes systematically exploit their mobility to benefit from contacts to forward messages. This mobility introduces delays when a node cannot forward its message. It also allows routing protocols to exploit opportunistic contacts, in absence of a stable end-to-end path, as a means to create a temporal path for delivery. Opportunistic networks are also suitable for communications in pervasive environments saturated by other devices. The ability to self-organize using the local interactions among nodes, added to mobility, leads to a shift from legacy packet-based communications towards a message-based communication paradigm.

However, dealing with the dynamics of opportunistic networks is complicated [5]. The continuously changing topology, due to the nodes mobility and interactions, leads to an explosion of the number of states needed to characterize the behavior for any algorithm to be deployed. All these factors impact network performances, introducing delay, packet losses or retransmissions. Up to now, the main focus of research was to define an optimal routing strategy, without considering the final application development.

With the current trend on connected devices, the idea of *opportunistic applications*, i.e., applications running over opportunistic networks, is getting closer to being a reality. However, several obstacles prevent a massive deployment over this paradigm, one of the biggest being the conception of applications working on these networks. This can be explained by the complexity to evaluate the performance of an opportunistic application before a real deployment.

Simulations offer a convenient way of getting insight in the behavior of the network [4, 9, 10]. Unfortunately, they do not give a simple way of thinking in terms of real applications, and typically focus on purely network-related performance. Testbeds [3, 11, 8, 17, 19, 13, 12, 18] can offer an almost real world feedback, but they are really expensive to deploy in a development process.

Developers of opportunistic applications must not only deal with network characterization, but also with its impact on the application. Current development tools should be able to fill this gap between network characterization and application development. However, even the ability to quickly test a simple DTN messaging application is missing today. *We need to better integrate how developers consider network metrics obtained from the characterization phase into the development process of opportunistic applications.*

In this work, we focus on helping developers to conceive their opportunistic applications. In particular, we propose

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

a new hybrid emulation system for opportunistic networks. In our system, nodes can be either real or virtual. Our main contribution is two-fold: *(i)* we propose HINT, a centralized low complexity topology emulator for opportunistic networks and *(ii)* we perform a first evaluation of its scalability. We develop this emulator as a distributed real-time event-driven system.

The rest of the paper is structured as follows. We highlight the challenges in Section 2, before describing our proposed emulation platform in Section 3. Then, we present a preliminary evaluation of the scalability of our proposal in Section 4. Section 5 reviews existing emulators. Finally, we conclude in Section 6.

2. DEVELOPMENT CHALLENGES OF OPPORTUNISTIC APPLICATIONS

In this section, we discuss the challenges when developing opportunistic applications from two perspectives: the way developers deal with the network characterization, and the way they assess the network impact on the application.

2.1 Opportunistic Networks Characterization

We now discuss the main challenges from a developer's point of view. We present the current available alternatives (and their drawbacks) to characterize opportunistic networks.

2.1.1 Analytical modeling

Several analytical models have been presented to characterize opportunistic networks. The main goal of analytical models is to provide a closed formula for a specific characteristic. However, most of these models assume simplifying hypotheses or cannot scale. Indeed, the number of states needed to model DTNs increases with the interactions parties have in the network, making these problems highly combinatorial ones. Most of them belong to the NP-class.

2.1.2 DTN simulators

Opportunistic networks simulators [10, 4, 9] mainly focus on nodes mobility and routing. Indeed, most of the DTN research has focused on message routing as the application.

On the one hand, we find many custom-made simulators for specific cases. On the other hand, we find an effort to standardize the results with the ONE [10]. The ONE simulator does provide a simulated network stack, but its application layer is just a basic handler class for messages passed by from the simulated routing protocol. For a developer, being unable to think in terms of a real user application, independently of these complexities, is still a huge problem.

2.1.3 Traces collection

Another effort to better understand opportunistic networks and their dynamics has been the collection of contact traces to characterize the contact and intercontact time distributions. Ideally, this abstraction should be independent from the link layer, but this is not the case in reality. This makes traces less representative than needed, and therefore less useful to application developers.

2.1.4 DTN emulators and testbeds

Emulation naturally provides a bridge between the repeatability and scalability of simulation and the realism of real world testing [2], by putting together real and simulated components in a single system. Real parts are most of the time the application and underlying operating system, while the network is simulated. However, scalability is often achieved with testbeds, which are costly to setup.

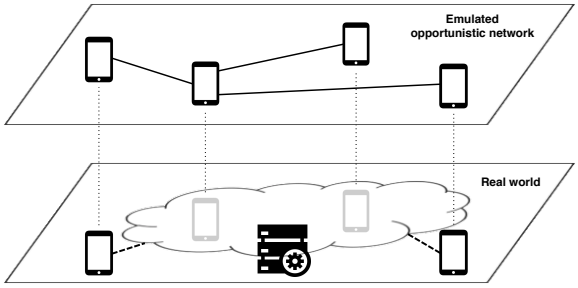

Figure 1: High level architecture, showing the relation between real world and emulated topology. Solid lines denote emulated connections, dashed lines represent real connections with the emulator.

2.2 Application Development

In an opportunistic network context, end-to-end delays, delivery ratio and drop ratio are very important factors that developers want to study before deploying their applications. However, this set of network-related metrics is not sufficient to describe the impact of those networks: developers also need to test their applications from a user viewpoint, and answer the question: *Is my application working as expected?*

On the one hand, the properties of opportunistic networks, such as the non guarantee of end-to-end paths, make impossible to ignore network characterization when developing opportunistic applications. On the other hand, developers may not be able to benefit from the network characterization. Instead, they want to know, preferably in a quick and simple way, if *their* applications will *still* work within an opportunistic use case. Making hypotheses on the underlying network characteristics often leads to over provisioning and resources waste, which is highly problematic in the already challenging context of opportunistic networking.

Hence, there is a gap in the way developers deal with network metrics obtained from network characterization.

3. ARCHITECTURE OF AN OPPORTUNISTIC EMULATOR

In [1], we addressed the limitations of current approaches, highlighting the gap between network characterization and development. This is a challenge that developers have to face when conceiving opportunistic applications. We stated the adequacy of an emulator to fill the gap between the underlying network and applications. In the following, we summarize the requirements stated in [1] and define the architecture of our emulator based on these requirements.

3.1 Requirements in a Nutshell

In [1], we defined several requirements for an opportunistic network emulator. We argued that such an emulator should be able to run in real-time (E1). To abstract the complexity of node mobility, we define contact-oriented emulation (E2). To observe and tune the parameters during an

experiment, we also need real-time tuning (E3) and monitoring (E4). Then, to ensure validity of the results derived, we highlighted the need for transparency from an application point of view (E5) and repeatability (E6) to ease debugging. Finally, to make an emulator really useful to application developers, availability (E7) is also a desirable feature.

3.2 High Level Architecture

We define two interaction levels: the real world and the emulated world. In the real world, *real devices* (physical Android mobile phones) run the application to be tested and the emulator service. In the emulated world, *virtual devices* and *real devices* interact in an opportunistic way. In the following, the general term *opportunistic node* describes either a *virtual* or a *real device*.

Our emulator creates and manages *virtual nodes* according to user requirements. Therefore, the resulting emulated opportunistic network is composed of several *opportunistic nodes*. The emulator also defines the connections between *opportunistic nodes* at the emulated level, and applies changes in real-time according to contact opportunities. Note that *real nodes* can only communicate with the emulator (at the real world level), and not directly with each other. Hence, we ensure that all connections go through the emulator. Several network topologies can be drawn, according to the considered user scenario.

Figure 1 shows both interaction levels with the projection of the different *opportunistic nodes* on the emulation plane. Solid lines indicate that two *opportunistic nodes* are in range (contact opportunity) while dashed lines represent real connections (a mobile phone connected to our local network hosting the emulator through Wi-Fi).

3.3 System Architecture

In this section, we propose the architecture of our real-time event-driven emulator, called HINT. It stands for *HINT Is Not a Testbed*. Since our system targets developers, it should therefore be lightweight enough to fit into existing development environments. This is the meaning of requirement (E7) about availability. A second feature is the ability to give developers *hints* on their application behavior when running on an opportunistic network, without requiring a real world trial. Figure 2 presents the global architecture of our proposal, which can be decomposed in five main parts:

3.3.1 Core Emulator

It is a real-time event-driven system, in charge of running the emulation scenario. A scenario specifies the following parameters: number of real and virtual devices, node characteristics, contacts and intercontacts management, message creation patterns, message forwarding or routing strategy and buffer management. Events, such as contact durations, intercontact durations, message creation and forwarding, are created and scheduled for execution in an event queue. Routing decisions and buffer management are also handled for all nodes. Although the functionalities listed above are very similar to those offered by a simulator (such as The ONE [10]), the Core Emulator supports real-time message delivery (and the corresponding payloads).

3.3.2 Message Broker

This module manages the interactions between nodes. We use the message broker in order to store both system messages and nodes' messages. Indeed, events from the Core Emulator generate meta-messages (i.e., announce connection start and duration) that are sent to the broker to a system queue. Regular messages contain the actual application data exchanged through the network and are stored in dedicated queues for each node. Hence, the message broker allows the communication between each pair of nodes (real or virtual). Since the number of pairs can be large, a specialized service that can scale is needed.

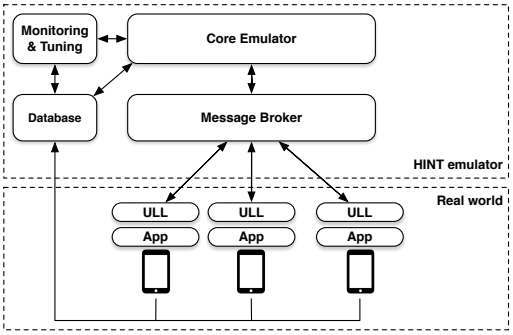

Figure 2: HINT network emulator architecture.

3.3.3 User Link Layer (ULL)

It abstracts the communication between the emulator and the real application, thus making the application unaware of the emulator. It communicates messages from the node queues to the real application, or from the application to the node queues. This layer could also be used to plug-in different DTN stack implementations.

3.3.4 Database

This module interacts with the Core Emulator to manage nodes. It stores the characteristics of each node: node type, ID, connection parameters, message creation frequency, etc. It also contains data for the monitoring part, in order to avoid unnecessary polling of the Message Broker queues.

3.3.5 Cross-layer Monitoring and Tuning

It is a cross-layer module where statistics are collected and displayed in real-time. This layer also allows to change parameters such as connections between nodes, message creation frequency, node buffer size or link speed.

The defined architecture complies with the requirements presented in [1]. The Core Emulator is a real-time system (E1). It produces a contact-oriented emulation (E2). The Cross-layer Monitoring and Tuning part validates (E3) and (E4), while the User Link Layer abstracts the communications with the emulator. This achieves (E5). Finally, the Core Emulator will be able to replay a scenario with the same contacts, routing and message generation settings, which realizes (E6). To show the availability (E7) of our architecture, we can note that deploying it only requires real nodes, a computer to run the emulator, and an access point to connect all these devices to the same network. This is a much lighter setup than a full testbed.

3.4 Implementation

We now discuss a basic proof of concept of the proposed architecture. We developed this prototype in Python 3.5

with the RabbitMQ message broker and MongoDB database.

3.4.1 Core Emulator

The Core Emulator defines an emulation scenario using the World class, which contains Nodes. Node interactions are modeled as events, divided in 3 categories. First, we have the usual Contact and Intercontact events, required by our contact-oriented approach. Then, to perform routing, we define Create, Copy, Forward, Delivered, Dropped and Expired Messages. These events are also used to trigger database updates for the monitoring system. Contact, Intercontact and Create Message are generated in batches, and we also define Refill events as system messages to create new events when batches are almost consumed. This regeneration happens on a pairwise basis. Finally, the Tuning system creates Tuning events to propagate the changes requested by the user through the web interface.

We use a priority task scheduler to handle all the events. Each event keeps information about the pair of nodes involved and its expected execution time. Events are then chronologically executed in real-time. Contacts and intercontacts are generated according to user inputs, and can come from real traces or statistical distributions as needed. The number of virtual and real nodes is currently fixed for the whole experiment length.

3.4.2 Message Broker

We use RabbitMQ, taking advantage of its high scalability to enable the execution of all events in real-time with an increasing number of nodes. Each node (real or virtual) is responsible for creating three message queues that it will use as buffers. Thus, each node has an *Inbox queue*, an *Outbox queue* and a *Storage queue*. Each time that a connection is started between a pair of nodes, both of them will process the messages contained within their Inbox queues. All messages where the node is the final destination are moved to the Storage queue. The rest of the messages are moved into the Outbox queue for later processing by the specific routing algorithm used by the node.

3.4.3 User Link Layer

For convenience and since final applications should communicate through it, we implemented the User Link Layer as an Android service. Any developer who wants to use the HINT emulator should include this service in its application and provide suitable methods for communicate with it. Thus, each real node (i.e., Android device) runs both the application and the User Link Layer. No DTN stack is currently supported.

3.4.4 Cross-layer Monitoring and Tuning

We implemented a web-based interface for Monitoring and Tuning. Currently, three views are offered: full network, node pairs and single node. Each of these views present real-time node, network and pair statistics, respectively. We are able to present the evolution over time of DTN performance metrics such as message delay, delivery ratio or contact time distribution, to name a few. Indeed, these metrics will be required to validate the emulator results against other approaches like simulators. This web interface also allows us to tune parameters during the experiment both at the network or node level, such as node buffer size or message generation frequency. Evaluation of the application behavior is done directly on the real nodes.

3.4.5 Database

We use MongoDB. We store node characteristics, such as their connection parameters, current list of neighbors or degree. We also store the Message events (creation, copy, forward, drop, delivery, expiration) required to compute and display network metrics in the Monitoring system. Note that this stage only require the storage of message metadata, without the payload. Finally, general data on the emulation scenario (such as the number of nodes, message generation, contact and intercontact parameters) is also recorded.

3.4.6 User Application

Finally, we propose OppChat, a simple messaging application developed for Android. OppChat takes advantage of the User Link Layer to seamlessly exchange messages with other nodes through the HINT Core Emulator, which handles routing and buffer management. To send a chat message, the application just needs to call the appropriate User Link Layer method. To receive messages, the application declares a standard Broadcast Receiver subclass that appropriately filters and handles incoming chat messages from the User Link Layer service.

3.4.7 Discussion

Emulation provides results that are at least as good as the ones from simulation, while also allowing real devices interaction [2]. Although removing the communication layer can be seen as too radical, we have defined a contact-oriented emulation, where the total event length is known in advance. This differs from simulators, which instead use a connection-oriented approach. Contacts can come from real traces, statistical distributions or synthetic mobility models.

The three-layered architecture provides a good modularity with separation of concerns (Core Emulator, Message Broker and User Link Layer). This helps to keep track of the big number of events and messages of the emulated network, thus ensuring scalability. Also note that the components themselves run on different machines: application and User Link Layer run on the real nodes, while the other modules are found in the emulation machine. The Core Emulator can be extended with more routing algorithms or buffer management policies. The use of a message broker helps to multiplex events and provides an original way to implement node buffers. Also, several message queues (three per each opportunistic node) are used and allow to provide node metrics such as buffer occupancy. The User Link Layer can be replaced by DTN stacks, such as IBR-DTN [14].

In terms of scalability, the different layers are designed as modules that can be distributed if necessary. Finally, we provide a real-time execution emulator, with a negligible cost compared to a testbed or a real world deployment.

4. ARCHITECTURE EVALUATION

We now evaluate the performance of our emulator implementation presented in Section 3. Before using it for actual application development, we first need to check that it can effectively fulfill the requirements introduced in [1].

4.1 Method

As a first step, we choose to validate our system by assessing its capacity to scale to a large number of events per

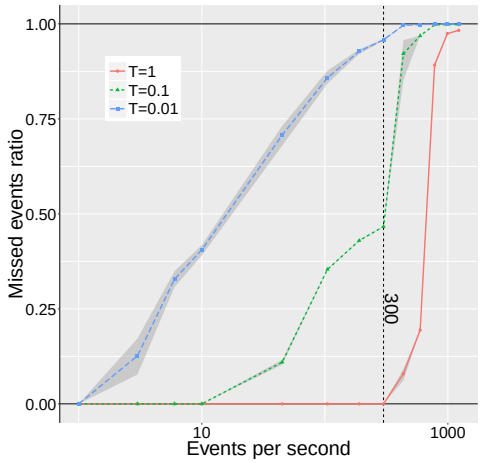

Figure 3: Number of missed events for different time thresholds (T) and system loads (events/s).

second. In this experiment, we are interested in how many contact (and intercontact) events our system can process each second experiencing an acceptable delay. By doing so, we want to evaluate the ability of the emulator to handle real-time operation.

We only consider virtual nodes for this evaluation. Since all buffers and events are managed within the Core Emulator, independently of the node type, this should not restrict the applicability of the results. The emulation machine where resides the Core Emulator is a MacBook Pro with 16 GB of RAM and with a 2.5 GHz Intel Core i7 processor. We use contact and intercontact events with a fixed duration of 1 second. The experiment duration is set to 20 seconds to allow us to repeat it several times. Although such parameters are not realistic in an opportunistic network setup, this experiment aims to find out the limitations of our system rather than evaluate network performance.

4.2 Results

The results are presented in Figure 3. It shows the ratio of delayed events (denoted as "missed events") for different time thresholds ($T = 0.01$, 0.1 and 1 s) and system loads. System load refers to the number of events scheduled per second. We compute each ratio by performing the mean over 5 emulation runs. Please note that we use a logarithmic scale for the X-axis. From the chart, it can be seen that a time threshold of $T = 0.01$ s is not achievable with the current configuration of the emulator. As a result, for this threshold, the number of missed events increases exponentially as a function of the system load. On the contrary, the emulator can handle up to 10 events per second with a constraint of $T = 0.1$ s. Finally, with a time threshold of $T = 1$ s, our emulator is able to schedule and execute up to 300 events per second with zero additional delay.

Limitations come from the internal Python scheduler we use. Indeed, with our current implementation, when an event starts to experience a delay, the scheduler will fall behind and the delay will be propagated to the following events. Unsurprisingly, the values also depend on the hardware specifications of the host computer running the scheduler, as well as various other options of the Python interpreter.

Trace	Average	Maximum
Rollernet	26	146
MIT 180 days	0.008	34
Infocom 2005	0.22	26

Table 1: Average and maximum number of events per second for three real traces.

One could argue that the number of events that HINT can handle per second is rather small, thus impeding the scalability requirement of our proposal. At this point, it could be interesting to compare the number of events that we are able to execute in one second with the ones experienced in real traces. We choose three traces for this analysis, and provide for each one the average and maximum number of events per second. Rollernet [16] and Infocom 2005 [15] are considered in full, without any filtering. The MIT dataset [6] is considered for 180 work days, only with internal devices. The aim of these choices is to leave as little data behind as possible, to derive meaningful upper bounds that HINT should be able to handle. The values are presented in Table 1.

We should note that even on a very dense real trace like Rollernet, the maximum number of events recorded in a single second is 146. These figures are obtained on the raw, unfiltered data, comprising all possible events and all 1112 nodes observed during the experiment. Actual values of number of events per second are much smaller: for example, the average value on the Rollernet dataset is 26 events per second, which is the highest value of the three traces considered here. Furthermore, in our case the same event load is applied consistently during the whole experiment, while such values on an actual trace are only temporary and would typically be surrounded by smaller values.

In this work, we provided a first evaluation of our emulation system by assessing its ability to handle an increasing load of contact and intercontact events in real-time. However, several other evaluations are required before we can actually use HINT for application development. As a first step, we will add message events in addition to contacts and intercontacts. Then, it is possible to vary the total number of nodes to derive the memory occupancy on the emulation computer, although this factor is closely tied to the buffer size chosen. In a second time, we will assess the realism of our emulator, for example by comparing performance results with other approaches like the ONE [10] in terms of delay, delivery ratio, or any metric presented by the web-based monitoring interface. Finally, we will include real nodes in the network to test real applications.

5. RELATED WORK

A popular solution for network emulation is to use testbeds. In this case, several hosts are used, each one running virtual machines representing the nodes of the studied network. Examples include QOMB [3, 2], TUNIE [11], MoViT [8] or the On/Off-based mobility emulator from [17].

Assembling real and virtual nodes in the same network allows easier scaling without totally sacrificing realism. The attempts made in this direction are called *hybrid emulation* techniques. Indeed, the only real part in the above solutions

is software, other elements being virtualized. For instance, in TWINE [19], simulated, emulated and real nodes interact in the same testbed. The TROWA testbed [13] relies on a discrete event simulation and also binds real and simulated nodes. The authors of [12] chose to use real machines communicating over a synthetic network, with a VPN solution being used to apply degradations.

In [3], the authors provide guidelines to include real nodes in an existing testbed, and propose a solution to connect them to both the experimental and control networks from the testbed. The closest proposal to our work is [18]. In this work, the authors use a central computer to emulate a DTN with several nodes, while both ends of the network are running on two other computers.

To the best of our knowledge, this is the first time that a lightweight emulation system puts together real and virtual nodes in a DTN context. Furthermore, none of the above proposals consider application development.

6. CONCLUSIONS

Opportunistic networking is a promising alternative to infrastructure-based networks, but its inherent complex dynamics make application development very challenging. Usually, some network characterization is needed to better understand the challenges we will face later on the development phase. Existing tools like DTN simulators or testbeds do not provide any integration with development. We argued that current development tools should fill the gap between network characterization and application development.

To bridge this gap, we designed HINT, a lightweight hybrid emulation system aimed at helping developers. Based on requirements, we proposed an architecture for this opportunistic emulator. Our proposal is a centralized low-complexity topology emulator for opportunistic networks, with real and virtual nodes. We developed this as a distributed real-time event-driven system. We also provided a proof of concept with a first implementation, as well as an assessment of the scalability of our proposal in terms of number of events per second. We show that HINT can successfully handle a number of events per second of the same order of magnitude as those observed in real traces.

As future work, we plan to continue the performance evaluation to validate our approach and integrate real nodes. Then we will carry on real case studies of opportunistic application developments. Finally, we will release an open source version of HINT.

7. ACKNOWLEDGMENTS

The authors are grateful to Tanguy Pérennou for his suggestions to improve this work. This research was supported in part by the French Ministry of Defense through a financial support of the Direction Générale de l'Armement (DGA).

8. REFERENCES

[1] G. Baudic, A. Auger, V. Ramiro, and E. Lochin. Using emulation to validate applications on opportunistic networks. Available: http://arxiv.org/abs/1606.06925, June 2016.

[2] R. Beuran. *Introduction to network emulation*. Pan Stanford Publishing, 2013.

[3] R. Beuran, S. Miwa, and Y. Shinoda. Making the Best of Two Worlds: A Framework for Hybrid Experiments. In *ACM WiNTECH '12*, 2012.

[4] X. Chang. Network simulations with OPNET. In *ACM WSC'99*, 1999.

[5] M. Conti and S. Giordano. Mobile ad hoc networking: milestones, challenges, and new research directions. *IEEE Communications Magazine*, January 2014.

[6] N. Eagle and A. S. Pentland. CRAWDAD data set mit/reality. Downloaded from http://crawdad.org/mit/reality/, July 2005.

[7] K. Fall. A delay-tolerant network architecture for challenged internets. In *ACM SIGCOMM*, 2003.

[8] E. Giordano, L. Codecà, B. Geffon, G. Grassi, G. Pau, and M. Gerla. MoViT: The Mobile Network Virtualized Testbed. In *ACM VANET '12*, 2012.

[9] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14, 2008.

[10] A. Keränen, J. Ott, and T. Kärkkäinen. The ONE simulator for DTN protocol evaluation. In *Simutools '09*. ICST, 2009.

[11] Y. Li, P. Hui, D. Jin, and S. Chen. Delay-tolerant network protocol testing and evaluation. *IEEE Communications Magazine*, 53(1), Jan. 2015.

[12] J. Liu, S. Mann, N. Van Vorst, and K. Hellman. An Open and Scalable Emulation Infrastructure for Large-Scale Real-Time Network Simulations. In *IEEE INFOCOM 2007*, May 2007.

[13] K. Maeda, K. Nakata, T. Umedu, H. Yamaguchi, K. Yasumoto, and T. Higashinoz. Hybrid Testbed Enabling Run-Time Operations for Wireless Applications. In *PADS '08*, June 2008.

[14] S. Schildt, J. Morgenroth, W.-B. Pöttner, and L. Wolf. IBR-DTN: A lightweight, modular and highly portable bundle protocol implementation. *Electronic Communications of the EASST*, Jan 2011.

[15] J. Scott, R. Gass, J. Crowcroft, P. Hui, C. Diot, and A. Chaintreau. CRAWDAD data set cambridge/haggle. Downloaded from http://crawdad.org/cambridge/haggle/, January 2006.

[16] P. Tournoux, J. Leguay, F. Benbadis, V. Conan, M. Dias de Amorim, and J. Whitbeck. The accordion phenomenon: Analysis, characterization, and impact on DTN routing. In *IEEE INFOCOM*, pages 1116–1124, April 2009.

[17] H. Yoon, J. Kim, M. Ott, and T. Rakotoarivelo. Mobility emulator for DTN and MANET applications. In *ACM WINTECH '09*, pages 51–58, 2009.

[18] Z. Zhang, Z. Jin, H. Chen, Y. Shu, and C. Zhao. Design and Implementation of a Delay-Tolerant Network Emulator Based in QualNet Simulator. In *WiCom '09*, pages 1–4, Sept. 2009.

[19] J. Zhou, Z. Ji, and R. Bagrodia. TWINE: A Hybrid Emulation Testbed for Wireless Networks and Applications. In *IEEE INFOCOM 2006*, Apr. 2006.