# JMS for Opportunistic Networks

Abdulkader Benchi, Pascale Launay, Frédéric Guidec

# JMS for Opportunistic Networks

Abdulkader Benchi[a], Pascale Launay[a,*], Frédéric Guidec[a]

[a]*IRISA Laboratory, Université de Bretagne-Sud, Vannes, France*

**Abstract**

Opportunistic networks constitute an appealing solution to complement fixed network infrastructures –or make up for the lack thereof– in challenged areas. Although many papers published in the last few years address the problem of supporting message dissemination in opportunistic networks, very little of them consider the problem of designing distributed applications capable of running in such networks. This article presents JOMS (Java Opportunistic Message Service), a carefully designed message-oriented middleware (MOM) system that is meant to ease the development of opportunistic distributed applications. JOMS fully supports the standard Java Message Service (JMS) specification, but unlike other JMS providers it implements a server-less model: message queues, topics, and a directory service are fully distributed among mobile devices, which collaborate to share information network-wide. JOMS has been evaluated in real conditions using netbooks and Android-based smartphones. The results of this evaluation are also presented in this article.

*Keywords:* opportunistic networking, Java message service, message-oriented middleware

## 1. Introduction

Opportunistic networks constitute an appealing solution to complement fixed network infrastructures –or make up for the lack thereof– in challenged areas. In this type of mobile networks, contacts between devices are intermittent and are hardly predictable. Additionally, because of the sparse and irregular distribution of mobile devices, neither end-to-end connectivity nor transmission delays can be guaranteed. The many forwarding protocols designed for Mobile Ad hoc Networks (MANETs) over the last two decades are usually ineffective in such conditions. Unplanned radio contacts between neighbor devices must be exploited opportunistically instead, and network-wide information dissemination can only rely on the store-carry-and-forward principle: any

*Corresponding author. Tel: +33297017218, Fax: +33297017279

*Email address:* pascale.launay@univ-ubs.fr (Pascale Launay)

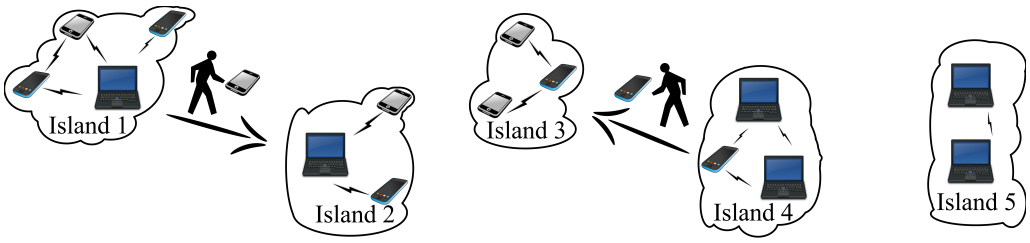



Figure 1: Example of opportunistic network composed of user-carried mobile devices

mobile device that gets an opportunity to obtain a message from a neighbor can serve as a mobile carrier (a.k.a. data mule) for this message for a while, and forward the message later to one or several other devices [1].

A typical opportunistic network is shown in Figure 1. In this example the mobile devices are smartphones, laptops or tablets carried by users. The network appears as a collection of distinct, continuously changing "islands" rather than as a single, connected network. Although no end-to-end path exists for example between islands 1 and 2, a user moving –deliberately or by chance– from island 1 to island 2 can serve as a carrier for messages addressed to devices located in island 2. The lack of end-to-end connectivity in the network is thus tolerated thanks to the mobility of devices. Yet there is no guarantee that a message ever gets delivered to its destination(s): message dissemination in an opportunistic network is a best-effort activity, which is directly constrained by how devices move and get in radio contact.

Many papers published in the last few years address the problem of supporting message dissemination in opportunistic networks (surveys can notably be found in [1, 2, 3, 4, 5]). In contrast, very little of them consider the problem of designing distributed applications capable of running in such challenged networks. Indeed designing applications that can tolerate long transmission delays and occasional –and sometimes frequent– message loss is a tedious task for developers. It is generally admitted that a peer-to-peer architecture should be preferred over a client-server one, because in most scenarios no device can be considered as stable and accessible enough to provide services to all other devices. Additionally, any form of synchronous interaction should be proscribed between different parts of a distributed application. Asynchronous communication is required in order to cope with long transmission delays.

The concept of middleware has long proved efficient in easing the development of distributed applications for traditional, fully-connected networks. It can be expected that carefully designed middleware systems bring similar benefits for opportunistic computing, that is, computing over opportunistic networks. Among the various types of middleware, message-oriented middleware (MOM) allows communication between application components by asynchronous message exchanges. This paradigm achieves decoupling of application components very naturally, making it well suited for developing

distributed applications for opportunistic networks.

Despite a lack of standards defining MOM application programming interfaces (API), protocols and message formats, JMS (Java Message Service) is a MOM that is widely used by Java developers. Yet most existing JMS implementations have been designed for traditional wired networks, and they rely on server-based message repositories and name directories. Such implementations are clearly not usable in opportunistic networks, whose disconnected and continuously changing topology precludes resorting to a server-based architecture.

In the remainder of this paper, we present JOMS *(Java Opportunistic Message Service)*, a provider for JMS we designed specifically for opportunistic networks. Unlike other JMS providers JOMS uses a server-less model: message queues and topics are distributed in the network, and a distributed directory service is used to discover and locate destination objects. Additionally, message dissemination in the network relies on a content-driven epidemic model, whereas all mobile devices collaborate to disseminate messages network-wide.

The next sections of this article are structured as follows. Section 2 presents the general architecture and principles of JOMS, whose implementation is evaluated in Section 3. Related work is discussed in Section 4, and Section 5 concludes this paper and describes our plans for future work.

## 2. Java Opportunistic Message Service (JOMS)

### *2.1. Background*

The Java Platform Enterprise Edition (Java EE) defines the JMS API, that specifies how MOM services are accessed, but not how those services should be implemented [6]. The JMS specification defines two communication models: point-to-point and publish-subscribe. The point-to-point model is built on the concept of *queues*. A *queue sender* produces a message for a specific queue, from which a *queue receiver* can consume it asynchronously. This model provides *one-to-one* communication, since a given queue may have multiple senders and multiple receivers, but each message addressed to this queue by a sender can be consumed by one receiver only. The publish-subscribe model is based on the use of *topics,* that can be subscribed to by *topic subscribers*. Messages are published to a topic by *topic publishers,* and they can then be received asynchronously by all the corresponding topic subscribers. One message can thus be consumed by multiple subscribers. This model complements the point-to-point model in that it provides *one-to-many* communication.

Applications can learn about the available topics and queues (a.k.a., *destination objects*) through the Java Naming and Directory Interface (JNDI). JNDI is a Java API that provides a common interface to access various naming and directory services [7]. According to the JNDI specification, any directory service must provide a hierarchical structure, referred to as a *namespace*. Each name in a namespace is bound to an object

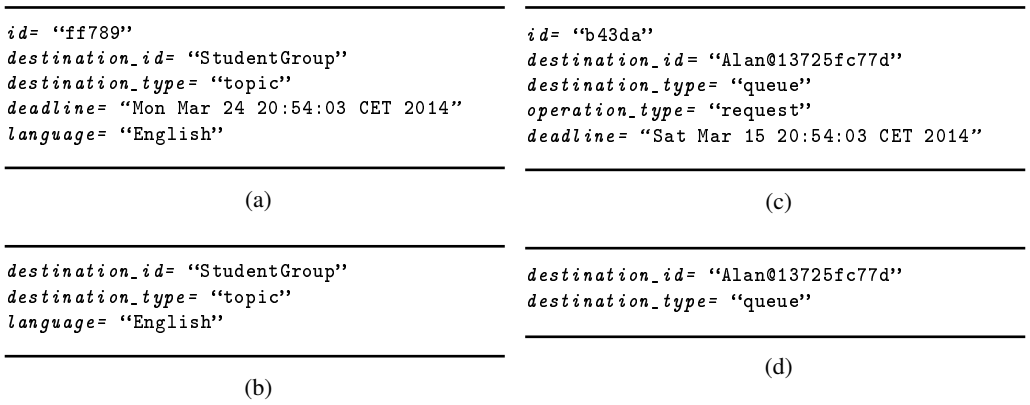



Figure 2: Examples of message descriptors (up) and selection patterns (down)

and resides there until it is explicitly unbound. In a JMS-based application, a destination object is meant to be created via some administration tool, and bound to a name in a JNDI repository. From then on, applications typically use JNDI to discover names and look up the destination objects they require.

The JMS API is widely used to develop distributed applications for traditional networks. It is commonly admitted that it helps software engineers build distributed systems that communicate in a loosely-coupled asynchronous manner. Many providers implement the JMS API for various deployment environments. Most of these providers rely on TCP, though, and they assume a synchronous or quasi-synchronous reliable communication layer. Some JMS providers have also been proposed for mobile ad hoc networks, but none of them is usable in opportunistic networks because they cannot cope with the absence of end-to-end connectivity in such networks.

JOMS (Java Opportunistic Message Service) has been designed specifically to meet the needs of application developers targeting opportunistic networks. It is architectured as a two-layer system: the lower layer is devoted to supporting communication in an opportunistic network, while the upper layer provides JMS on top of this communication service. The upper layer is actually composed of two parts: the JMS provider per se, and a distributed directory service that complies with the JNDI specification. These three elements are detailed below.

### *2.2. Communication Layer*

The communication layer implements a content-driven message dissemination model: messages flow towards interested receivers rather than towards specifically set destinations, and each receiver can serve as a "data mule", carrying messages for a while in a local cache so they can be transferred later to other interested receivers.

The communication layer in JOMS provides higher-layer services with a publish/subscribe API. Each message is given a *descriptor* that characterizes its content. This descriptor is

composed of *(name, value)* tuples, as illustrated in Figure 2a and Figure 2c. An application service that needs to receive a particular type of messages must use the API to set a subscription accordingly. A subscription is characterized by a selection pattern, whose structure is similar to that of descriptors (see Figure 2b and Figure 2d). The *interest profile* of a device is basically a compilation of all the selection patterns that characterize local subscriptions.

The interaction scheme implemented in the communication layer takes inspiration from the Autonomous Gossiping (A/G) algorithm [8], which itself defines a selective version of the epidemic routing model proposed in [9]. Each host periodically broadcasts an announcement in order to inform its neighbors (if any) about its identity and interest profile. By sending such an announcement periodically, a node informs its neighbors about its presence and about the kinds of messages it is interested in. Conversely, by receiving similar announcements a host discovers its neighbors, and learns about their own interest profiles. By matching its neighbor's profiles against the descriptors of the messages it maintains in its cache, a host can select descriptors of messages that might be of interest to at least one of its current neighbors. It can thus build an offer containing these descriptors, and incorporate this offer in its next announcement. Upon receiving such an offer, each host matches the descriptors it contains against its own interest profile in order to identify messages that match this profile and that are not already present in its local cache. If such messages are identified, then a request for these messages is sent to the announcer, which complies by sending the missing messages on the radio channel. Finally, when a host receives a message it has requested, this message is put in the local cache so it can later be proposed to other hosts met while moving in the network.

Figure 3 illustrates the gossiping between two hosts *H1* and *H2* as they exchange messages. In this example we can assume that *H1* and *H2* are within mutual transmission range, but that they are sometimes in suspend mode (though not necessarily at the same time). The same behavior could be obtained with these hosts moving and getting in a transient radio contact, though.

An application process *P2* on *H2* first publishes several messages *a, b, c... j* (label <1> in Fig. 3). Since *H2* has no neighbor at that time, these messages are simply put in the host's local cache. *H2* then enters suspend mode for a while (label <2>). *H1* is started (for the first time, or resuming from suspend mode), and an application process on that host subscribes to receive messages that match a specific selection pattern (label <3>). *H1* then begins announcing its presence, with an interest profile that includes the pattern specified by process *P1* (label <4>). These announcements are not received by any host, though, since *H1* currently has no neighbor.

*H2* resumes (label <5>), and shortly after that it receives an announcement broadcast by *H1* (label <6>). *H2* thus discovers this neighbor, and it simultaneously learns what kind of messages *H1* is interested in. Based on *H1*'s interest profile, *H2* can check its local cache and identify messages whose descriptors match this profile. *H2* can thus sends an offer to *H1* (label <7>). Upon receiving this offer *H1* determines which of the

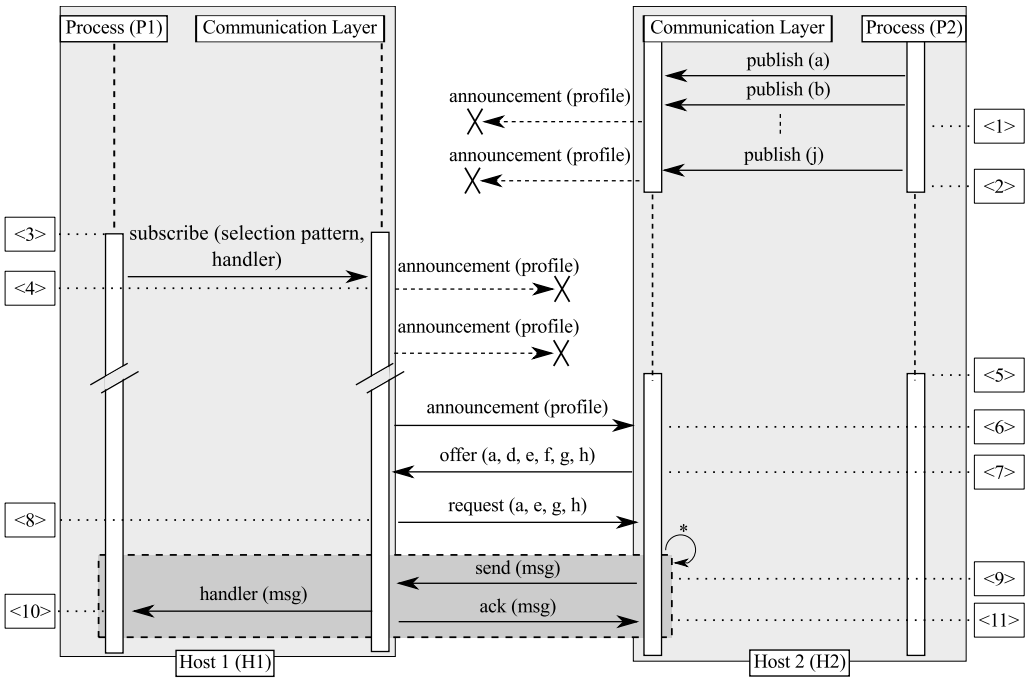



Figure 3: Interaction diagram between two hosts exchanging a message

offered messages it would actually like to receive (*i.e.,* messages that are not already in its own local cache), and sends a request accordingly to *H2* (label <8>). For each message thus requested, *H2* sends the message to *H1* (label <9>) and waits for an acknowledgement before sending the next message (label <11>). Each message received by *H1* is put in the local cache, and the handler of the application process is called in order to process the message (label <10>). Although this is not represented in Fig. 3 for the sake of clarity, this interaction pattern involving *H2* providing messages to *H1* is also run symmetrically: *H1* may likewise provide *H2* with messages that match its own interest profile. Moreover, *H1* and *H2* may simultaneously be interacting with several other neighbor hosts. Finally, the interaction procedure between two hosts is stateless and can thus be interrupted at any time: both hosts will simply detect that they have lost a neighbor, and act accordingly.

Further details about this interaction schema and about how it performs in real conditions can be found in [10].

As a general rule, a host that subscribes to receive a particular kind of message is expected to serve as a mobile carrier for this kind of message. Yet a host can also be configured so as to serve as an altruistic carrier for messages that present no interest to the application services it runs locally. This behavior is optional, though, and it must be enabled explicitly by an administrator of the DoDWAN platform.

Cache overloading and network congestion are avoided by setting a *deadline* for each message. Whenever a message gets out of date, all copies of this message are removed from caches, so it stops disseminating in the network.

### *2.3. Directory Service*

Application components must look up the JMS destination objects –topics or queues– they intend to use in a directory service. The JMS specification states that objects discovery must be made through the JNDI API [7]. JOMS implements a distributed directory service, which complies with this API. Each host maintains a local directory that acts as a local namespace, from which application services can look up destination objects. When a destination object is created on a device, an entry for this object is added to the local namespace, and JOMS starts disseminating an advertisement about it. Devices that receive this advertisement update their own namespace accordingly.

An entry in a namespace must be characterized by a unique name. Topics created on different devices with the same name are considered as being the same topic, for all messages published in that topic will be received by all subscribers. In contrast queues must have unique names, as each queue is to be maintained on a single device (as explained in the next section). When a queue is created, the name specified by the user is appended with the local device's identifier, which is provided by the communication layer (this identifier can for example be the IMEI on a smartphone, or an auto-configured link-local IPv6 address). Figure 2c and Figure 2d respectively show a descriptor and a pattern that both refer to a queue whose identifier is "Alan@0013725fc77d".

The JNDI API defines primitives to unbind objects from a repository. Disseminating an unbind advertisement for an object in an opportunistic network does not ensure that the dissemination of the corresponding binding advertisement will effectively be stopped. For this reason, JOMS adds a deadline to JNDI entries in order to delete obsolete destination objects. A refresh mechanism is used to reset the deadline of entries automatically, as long as they are used by application services: whenever a device sends or receives a message to/from a specified destination object, the deadline for the corresponding entry is systematically prolonged in this device's namespace.

### *2.4. JMS Provider*

According to the JMS specification, a JMS message is composed of a header, properties, and a body [6]. The header and properties contain fields for the identification, control and routing of messages. JOMS uses those fields to manage the opportunistic dissemination of messages. The message body carries the message content. Its content is ignored by JOMS, which considers it as a simple payload. JOMS introduces a few discrepancies with respect to the standard JMS properties, in order to deal with the nature of opportunistic networks. Specifically, fields *JMSExpiration* and *JMSMessageID* are required in JOMS, while they are optional according to the JMS specification.

The JMS publish-subscribe model maps well to the communication API of JOMS. Messages published in a given topic are tagged with the topic name, as shown in Figure 2a. All subscribers to a topic define a pattern with this topic's name in their interest profile (see Figure 2b). Thus, a message for a given topic disseminates in the network thanks to the topic's subscribers that serve as data mules and carry it in their local cache. Topic subscribers can filter out messages beyond the topic name, though, using a selector property as defined in the JMS specification. This property is added to the interest profile, so message filtering is processed by the opportunistic content-based communication layer.

With the point-to-point model, a message sent to a queue must be consumed by only one queue receiver. In order to enforce this rule, each queue created with JOMS is maintained by a single queue manager (QM), which is the device on which the queue is created. Messages sent to a queue, and requests to get messages from that queue, are forwarded by JOMS towards the corresponding QM. Permanent interest patterns defined in the communication layer ensure that each device serves as a mobile carrier for messages sent to or read from queues. Note that when several requests reach the same QM simultaneously, the QM ensures that each message read from the queue is sent to only one receiver, thus enforcing the above mentioned rule. Additionally, requests can include a selection pattern, so the QM searches in the queue for a message whose properties match this pattern. Figure 2c represents a request to get a message from the queue "Alan@0013725fc77d". A QM with the selection pattern shown in Figure 2d will receive all the requests and messages for this queue.

The JMS specification allows programmers to specify the expected degree of reliability (guaranteed vs best-effort delivery), through the so-called *persistent* and *non-persistent* delivery modes. Moreover, the standard *JMSPriority* property expresses the expected priority level when transmitting messages. Message delivery can hardly be guaranteed in an opportunistic network (unless specific assumptions are made on the characteristics of this network). JOMS however uses these JMS properties to define a metric that combines the requested delivery mode and priority level. This metric is defined in a *privilege()* function, which can be edited if needed by an application developer. It is used to increase the delivery probability for "important" messages by adjusting the cache management policy and the message selection policy of the communication layer. When a new message is created or received while the local cache exceeds its capacity, messages with lower privilege level are discarded first. When a contact is established between two devices, messages with higher privilege level are transferred first.

## 3. Experimental Evaluation

Evaluating the performance of a middleware system capable of running in an opportunistic network is a challenge. In the literature, protocols and middleware systems designed for such networks are often evaluated using simulators, and little or no effort is

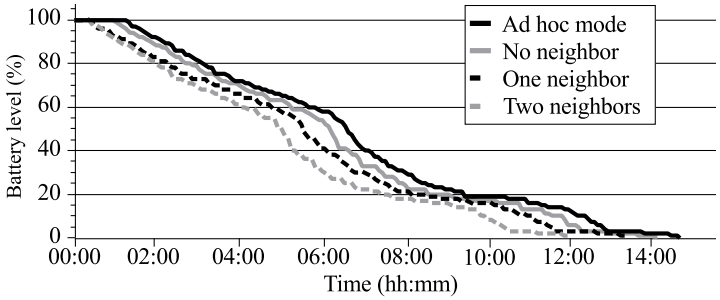



Figure 4: Battery drain when running JOMS in the background on a smartphone (in ad hoc mode)

devoted to producing code that can be used in a real setting. In contrast, a salient feature of JOMS is that it has been fully implemented, and is now distributed under the terms of the GNU General Public License[1]. Moreover its effectiveness and efficiency have been evaluated in real conditions. To this end, several measurement campaigns have been conducted, using either smartphones or netbooks as host devices. During these experiments it would have been most interesting to compare JOMS against other JMS providers. Unfortunately, JOMS is –to the best of our knowledge– currently the only JMS provider for opportunistic networks that is openly available for application developers.

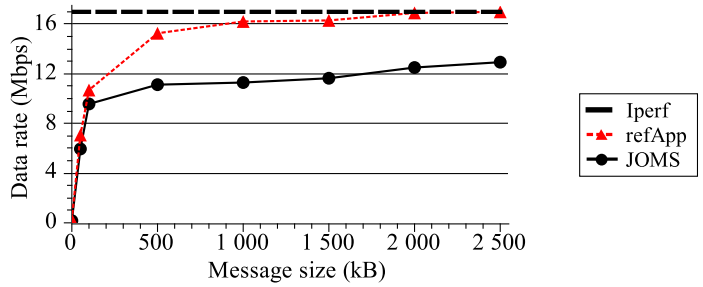
### *3.1. Resource Consumption*

An important factor in opportunistic networks is the resource-constrained nature of mobile devices, which are generally characterized by limited processing power, memory size, and power budget. We have developed an Android service using JOMS, on which a series of tests were performed to better determine how much resources it consumes on HTC Wildfire S smartphones.

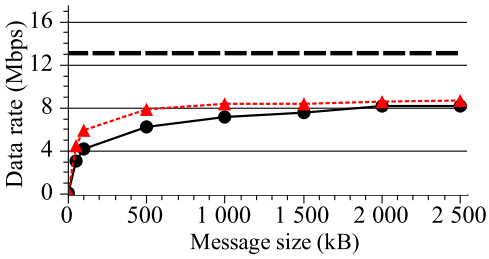
In the first test, JOMS was kept running as a background Android service, with no application-level traffic but the episodic gossiping used by JOMS for neighbor detection and coordination. During this test, which lasted for 14.5 hours before the battery was drained, Android services together consumed 9.25 minutes of CPU activity (about 1% duty cycle), while JOMS itself consumed 16 seconds of CPU activity (about 0.03% duty cycle). Besides, once installed JOMS consumed about 3.63 MB of memory (with an empty message cache), which represents less than 1% of the 418 MB available on a Wildfire S smartphone.

In a second series of tests we measured the effect of running JOMS on the battery life of a smartphone. As a baseline, we measured the battery drain on a smartphone running no application, with the Wi-Fi chipset enabled and operating in ad hoc mode.

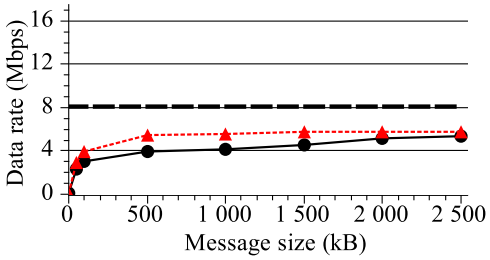
[1] http://www-casa.irisa.fr/joms




(a) Between 1-hop (direct) neighbors




(b) Between 2-hop neighbors




(c) Between 3-hop neighbors

Figure 5: Transmission throughputs observed in a single connected island (with or without multi-hop forwarding)

We then repeated the test several times with JOMS running and sending a new message every 3 minutes, with a varying number of neighbors. The evolution of the battery level during these tests is shown in Figure 4. It can be observed that the beaconing used by JOMS for neighbor discovery (when no neighbor is there to respond) reduces the battery life of a smartphone by about 5%, and that the gossiping with one neighbor reduces the battery life by an additional 15%. The battery drain does not grow significantly with a higher number of neighbors, though. This figure confirms that the battery drain in a smartphone is mainly driven by the ad hoc mode, and only marginally by the beaconing and gossiping performed by JOMS. Our system therefore allows a smartphone to run for a long time, without consuming too much resources while running as a background service.

### *3.2. Performance Assessment in a Connected Environment*

Before trying to observe how messages can propagate network-wide in an opportunistic network, we first strove to observe how they can propagate among connected devices, that is, within a single connectivity island.

To do so we used four netbooks deployed in adjacent rooms so as to form a straight line A-B-C-D. The connectivity links between these netbooks was such that each net-

book could only communicate with up to two neighbors (e.g., B could only reach A and C). The tests relied on the queue-based transmission model. Message queues were created on B, C, D, and netbook A was configured as a subscriber to all these queues. Messages with sizes varying between 1 kB and 2.5 MB were deposited successively in each queue (by local deposits), and we measured the time required for these messages to reach subscriber A. In order to get baseline values, we used the network performance measurement tool *Iperf*, which allowed us to measure the maximal throughput achievable on statically defined routing paths at IP level. We also developed a simple reference application (hereafter referred to as *refApp*), capable of sending, relaying and receiving data bundles over UDP, with data fragmentation and reassembly if needed. This application allowed us to measure the throughput achievable at message level, without paying the cost of neighbor discovery or gossiping among neighbors. Of course *Iperf* and *refApp* can both only be used in a connected island, as they require temporaneous end-to-end connectivity and multi-hop forwarding.

The results of the tests are shown in Figure 5a, 5b and 5c for transmissions over 1, 2 and 3 hops, respectively. Each value presented has been averaged over 150 rounds, for each configuration.

It can be observed that JOMS shows 5% to 20% overhead over *refApp*, which is an indication of the cost of discovering neighbors and gossiping with them before exchanging actual applicative messages. This overhead gets lower when multi-hop forwarding is required, for in that case *refApp* must receive messages before forwarding them, just like JOMS. Moreover, the observed transmission rates globally decrease as the number of hops increases. This is because when host B must serve as a relay between A and C, the radio channel around B is twice as busy as when B interacts only with host A. The same observation applies for host C when it must serve as a relay between B and D.

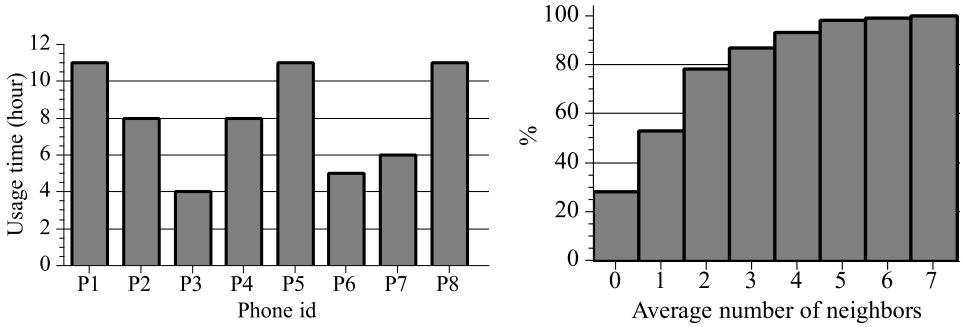
These results have been obtained with the queue-based transmission model, but similar tests involving topic messages show very similar results, so they are not detailed in this article: the curves obtained with queue-based and topic-based transmissions would actually be superimposed in the figures.

### *3.3. Performance Assessment in a Disconnected Environment*

The above mentioned tests demonstrated that JOMS can perform satisfactorily in a traditional connected environment, and show reasonable performances in such conditions. Yet the most interesting characteristic of JOMS is of course that it can ensure the delivery of messages in a disconnected network, where traditional JMS providers –or simple programs like *Iperf* or *refApp* for that matter– would be totally useless.

In order to demonstrate that JOMS can indeed support JMS messaging in an opportunistic network, eight volunteers were equipped with HTC smartphones. These devices ran a simple SMS-like[2] application based on JOMS. This application offers two services:

[2]SMS: Short Message Service.




(a) Usage duration of smartphones

(b) Cumulative distribution of the average number of neighbors

Figure 6: Activity and "sociability" of smartphones

public text and private text. The public text service relies on topics: each message published in a topic can be received by all subscribers. With the private text service, each message is addressed to a single message queue, which is hosted on the smartphone of the recipient user.

This experiment spanned over the working hours of two consecutive days for a total duration of 32 hours. The volunteers were asked to carry their smartphone whenever possible, and use both text services while moving inside our laboratory or in its surroundings.

Figure 6a shows how long each volunteer actually "used" his/her smartphone (i.e., with the screen lit) during the experiment. It can be observed that the smartphones remained untouched on average during three-fourth of the experiment duration. This does not imply that JOMS remained idle during that time, though, as it can keep gossiping with neighbor devices when running in the background.

The cumulative distribution function (CDF) of the average number of neighbors perceived by all smartphones is shown in Figure 6b. It can be observed that, in average, a smartphone did not have more than two neighbors simultaneously during almost 80% of the duration of the experiment, and that it was actually alone (i.e., with no neighbor) during about 30% of that time. These figures confirm the disconnected nature of the network formed by the smartphones, and demonstrate the need for opportunistic transmissions in order to maintain communication in such conditions.

A timeline of the average number of neighbors during the first day is shown in Figure 7. The high number of neighbors during the first two hours of the experiment is explained by the fact that all volunteers were in the same room at that time, as the purpose and conditions of the experiment were explained to them. Later on all volunteers dispersed in the laboratory, and the timeline only shows episodic contacts between their smartphones. These results again confirm that a communication system requiring end-

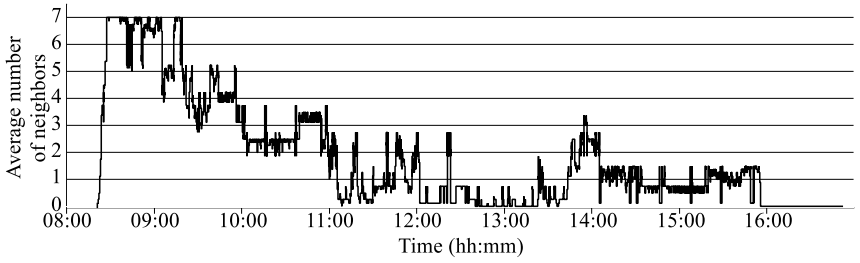
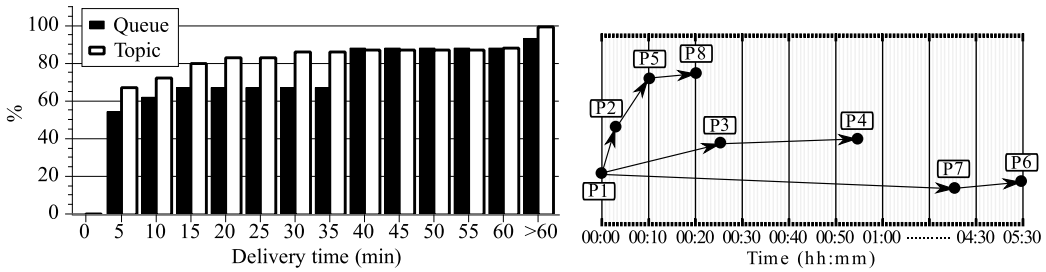



Figure 7: Timeline of the average number of neighbors during the first day




(a) Cumulative distribution function of the delivery time of received messages

(b) Timeline of the dissemination of a message

Figure 8: Message delivery time and dissemination of a message

to-end connectivity, and global access to any kind of server, would be totally impractical in such conditions.

The traces collected by JOMS on each smartphone show that 230 text messages were sent by the volunteers during the two-day experiment. Figure 8a presents the cumulative distributed function (CDF) of the delivery time for topic-based and queue-based messages. It can be observed that nearly 80% of topic-based messages got delivered in less than 20 minutes, whereas most queue-based messages took about 40 minutes to be delivered. This difference is due to the fact that a message for a queue has to get through the manager of the target queue before being sent to the final receiver. In contrast, a message for a topic disseminates directly from host to host before getting delivered to subscribers. In any case, the figures show that during the experiment most of the messages got delivered to their destination(s) in less than 1 hour. Yet, about 3% of the messages could not be delivered. This is the consequence of the unpredictable –yet perfectly legitimate– behavior of the volunteers, who sometimes moved away from the campus, or even switched their smartphone off in order to preserve its battery budget. By doing so they prevented any further radio contact between their smartphone and those of other users, and this of course led to message loss.

During the experiment, 5729 contacts were observed between smartphones, with

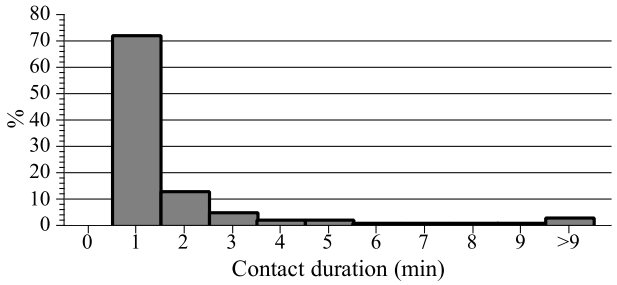



Figure 9: Distribution of contact durations

an average contact duration of 121 seconds. The distribution of contact durations is presented in Figure 9. The majority of radio contacts lasted for less than a minute. Furthermore, it is worth mentioning that 2.5% of radio contacts were not exploitable by JOMS because they were asymmetric, that is, one smartphone could detect a neighbor, but the reverse was not true. Such a situation usually happens for only a few seconds, and it occurs when the smartphones involved are too far from each other, so their signal is barely above noise level.

Figure 10 shows the number of messages received by each smartphone (in parentheses), and the distribution of the number of hops required for messages to reach that smartphone. It can be observed that most of the smartphones received at least half of the messages after a multi-hop trip. This observation confirms the interest of having smartphones serve as benevolent message carriers in an opportunistic network such as that considered in this experiment. In order to better illustrate this point, Figure 8b shows how one particular message disseminated during the experiment. This message was first published in the public topic by mobile *P1*. After only a few minutes *P1* established a radio contact with *P2*, which thus got a copy of the message and became a new carrier for this message. *P1* later managed to forward the message to *P3*, and later to *P7*, while *P2* forwarded it to *P5*. The message thus kept disseminating, until it reached the last subscriber *P6*, about 5 hours and 30 minutes after it was initially published.

The results presented in this section demonstrate that JOMS is effective and efficient in providing JMS services in a disconnected environment. An obvious advantage of implementing a provider that conforms with the JMS standard specification is that developers do not need to learn a new programming language, or get familiar with an exotic programming model or API. A developer can simply focus on writing a JMS application "as usual", and JOMS can take care of its execution in unusual conditions. Indeed, any pre-existing JMS application can theoretically be deployed using JOMS, and run satisfactorily in an opportunistic network. Yet there are a few common pitfalls developers should pay attention to while designing their code. Indeed, many standard applications based on JMS have been built with the implicit –but often wrong– assumption that the JMS provider can prevent message loss, and guarantee message ordering

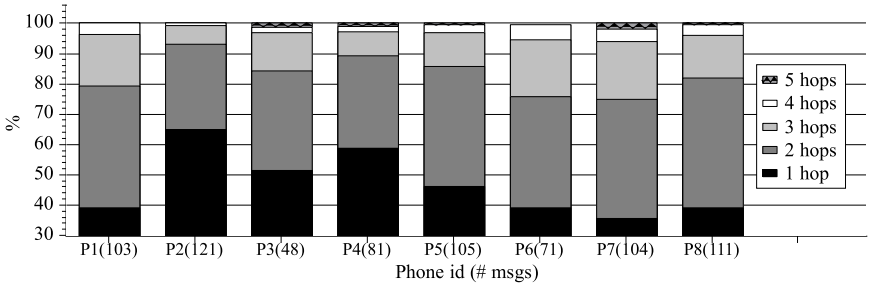



Figure 10: Amount of messages received in a direct/multi-hop manner by each smartphone

and transmission delay. As a general rule, such favorable properties cannot be guaranteed in an opportunistic network, so developers that wish to deploy their code in an opportunistic network should refrain from making similar assumptions.

## 4. Related Work

Message routing in delay/disruption-tolerant networks –and more specifically in opportunistic networks– has justified a fair amount of research during the last decade. Recent surveys can notably be found in [1, 2, 3, 4, 5]. Yet, as mentioned in [1], "rare are the [DTN] protocols that were implemented, tested in real-life and proven to be free of lethal stealthy assumptions."

A wide variety of application domains would benefit from opportunistic networking [11, 12]. Opportunistic computing is actually becoming a new distributed computing paradigm, involving transient and unplanned interactions between mobile devices [13]. Some middleware systems have been proposed in order to ease the task of application programmers for mobile ad hoc networks (MANETs) [14], adapting well-known middleware solutions or designing new ones. Our approach and those discussed below clearly fit in the first category, as they consist in developing JMS providers for ad hoc networks.

Vollset et al. describe a JMS implementation supporting non-persistent topics in MANETs [15]. This implementation uses a multicast routing protocol to provide publish/subscribe semantics by mapping JMS topics to multicast addresses. Since this protocol cannot disseminate messages beyond a single connected fragment of the network, it is hardly usable in opportunistic networks. It could probably be adapted, though, using a disruption-tolerant multicast routing protocol. In this system, no directory service (such as JNDI) is provided: all JMS clients are assumed to own a local copy of a configuration that contains information about all queues and topics. In the paper the authors propose some solutions to support JMS queues, but none of these solutions has actually been implemented. One of these solutions requires an agreement amongst mobile nodes through some consensus algorithm, which is hardly achievable in an opportunistic network.

JIMS (JMS Implementation for MANETs using Sociable nodes) has been designed during a Master's project at University College London [16]. It defines the notion of *social nodes*. A social node is one that is supposed to meet many other nodes while moving in a MANET. It also has a lot of resources, so it can carry many messages, and never needs to be turned off while roaming the network. This idea of having privileged nodes that can serve as effective data mules is an interesting one. A similar behavior can actually be obtained with JOMS, as the amount of resources used by the system on each device can be adjusted accurately.

Epidemic Messaging Middleware for Ad hoc networks (EMMA) is an adaptation of JMS that targets MANETs presenting connectivity disruptions [17]. EMMA assumes the availability of a so-called *synchronous protocol*, which can be used to reach mobile hosts that belong to the same *cloud* –or island– as the sender. An asynchronous epidemic routing protocol is used to disseminate messages towards remote clouds. EMMA manages queues in a manner that is quite similar to that of JOMS: each queue is maintained by a single holder, which advertises this object periodically with a set lifetime, and which can accept subscriptions from other hosts. EMMA and JOMS differ in the way they deal with topics. In EMMA topics are maintained by a single holder, which can accept subscriptions from other hosts. In JOMS topic subscriptions can be set locally on any host. Another difference is that in EMMA the gossiping mechanism between neighbor hosts is such that all messages are systematically considered, so very large lists of message identifiers can be exchanged between neighbor hosts. In JOMS this gossiping is constrained by the interest profiles of neighbors. Finally, EMMA defines its own communication protocol for route discovery and maintenance, while JOMS presents a two-layer architecture: the upper layer is concerned with queue and topic management and utilization, and the lower layer supports opportunistic communication.

Although [15], [16] and [17] have been published almost a decade ago, the JMS providers they describe have never been openly distributed. To the best of our knowledge, JOMS is currently the only provider that is available for developers that wish to develop JMS-based distributed applications for opportunistic networks.

## 5. Conclusion

Developing distributed applications for opportunistic networks can be quite a challenge, since in such networks very long transmission delays and possible message loss should be expected at any time. JOMS *(Java Opportunistic Message Service)* is a message-oriented middleware system that is meant to ease the development of such distributed applications. It is basically a JMS provider, but unlike other providers it does not rely on a central message repository or service directory. Message queues and topics are distributed in the network, and a distributed directory service is used to discover and locate destination objects. Information sharing relies on a content-driven epidemic scheme, whereas all mobile devices collaborate to disseminate messages network-wide.

This combination of unique features makes it possible for JOMS to perform effectively in an opportunistic network, while other JMS providers would be ineffective in similar conditions.

JOMS has been tested and evaluated in real conditions using Android-based smartphones (as reported in this article), but also using tablets, laptops, and netbooks. Future work involves leveraging JOMS to develop full-featured distributed applications that can benefit users of opportunistic networks, especially in disaster or economically challenged areas.

## References


[1] M. J. Khabbaz, A. Chadi M., F. Wissam F., Disruption-Tolerant Networking: a Comprehensive Survey on Recent Developments and Persisting Challenges, IEEE Communications Surveys and Tutorials 14 (2) (2012) 607–640.

[2] C. Boldrini, M. Conti, A. Passarella, Autonomic Behaviour of Opportunistic Network Routing, Inderscience International Journal of Autonomous and Adaptive Communications Systems 1 (1) (2008) 122–147.

[3] H. A. Nguyen, S. Giordano, Routing in Opportunistic Networks, International Journal of Ambient Computing and Intelligence 1 (3) (2009) 19–38.

[4] A. Triviño-Cabrera, S. Cañadas-Hurtado, Survey on Opportunistic Routing in Multihop Wireless Networks, International Journal of Communication Networks and Information Security 3 (2) (2011) 170–177.

[5] Z. Zhang, Q. Zhang, Delay/Disruption Tolerant Mobile Ad Hoc Networks: Latest Developments, Wireless Communications and Mobile Computing 7 (10) (2007) 1219–1232.

[6] M. Hapner, R. Sharma, R. Burridge, J. Fialli, K. Haase, Java Message Service API tutorial and Reference: Messaging for the J2EE Platform, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[7] R. Lee, S. Seligman, JNDI API Tutorial and Reference: Building Directory-enabled Java Applications, Addison-Wesley, Reading, MA, 2000.

[8] A. Datta, S. Quarteroni, K. Aberer, Autonomous Gossiping: A Self-Organizing Epidemic Algorithm for Selective Information Dissemination in Wireless Mobile Ad-Hoc Networks, in: International Conference on Semantics of a Networked World, 2004, pp. 126–143.

[9] A. Vahdat, D. Becker, Epidemic Routing for Partially Connected Ad Hoc Networks, Tech. rep., Duke University (Apr. 2000).

[10] J. Haillot, F. Guidec, A Protocol for Content-Based Communication in Disconnected Mobile Ad Hoc Networks, Journal of Mobile Information Systems 6 (2) (2010) 123–154.

[11] A. G. Voyiatzis, A Survey of Delay- and Disruption-Tolerant Networking Applications, Journal of Internet Engineering 5 (1) (2012) 331–344.

[12] A. Lindgren, P. Hui, The Quest for a Killer App for Opportunistic and Delay Tolerant Networks, in: Proceedings of the 4th ACM workshop on Challenged Networks, 2009, pp. 59–66.

[13] M. Conti, S. Giordano, M. May, A. Passarella, From Opportunistic Networks to Opportunistic Computing, IEEE Communications Magazine 48 (9) (2010) 126–139.

[14] S. Hadim, J. Al-Jaroodi, N. Mohamed, Trends in Middleware for Mobile Ad Hoc Networks, Journal of Communications 1 (4) (2006) 11–21.

[15] E. Vollset, D. Ingham, P. Ezhilchelvan, JMS on Mobile Ad Hoc Networks, in: Personal Wireless Communications, Springer-Verlag, 2003, pp. 40–52.

[16] H. Chen, L. Chen, K. Gupta, C. Savvidis, W. G. Teo, Reliable Asynchronous Middleware for Mobile Ad Hoc Networks, university College London, Dept of Computer Science, MSc DCNDS (Sep. 2008).

[17] M. Musolesi, C. Mascolo, S. Hailes, EMMA: Epidemic Messaging Middleware for Ad Hoc Networks, Personal and Ubiquitous Computing 10 (1) (2006) 28–36.