

A P2P Tuple Space Implementation for Disconnected MANETs

Abdulkader Benchi, Pascale Launay, Frédéric Guidec

▶ **To cite this version:**

Abdulkader Benchi, Pascale Launay, Frédéric Guidec. A P2P Tuple Space Implementation for Disconnected MANETs. Peer-to-Peer Networking and Applications, 2015, 8 (1), pp.87-102. 10.1007/s12083-013-0224-4 . hal-00861421

HAL Id: hal-00861421
https://hal.science/hal-00861421

Submitted on 12 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A P2P Tuple Space Implementation for Disconnected MANETs

Abdulkader Benchi, Pascale Launay, Frédéric Guidec*

IRISA, Université de Bretagne-Sud

Vannes, France

{abdulkader.benchi, pascale.launay, frederic.guidec}@univ-ubs.fr

Abstract Disconnected mobile ad hoc networks (or D-MANETs) are partially or intermittently connected wireless networks, in which continuous end-to-end connectivity between mobile nodes is not guaranteed. The ability to self-form and self-manage brings great opportunities for D-MANETs, but developing distributed applications capable of running in such networks remains a major challenge. A middleware system is thus needed between network level and application level in order to ease application development, and help developers take advantage of the unique features of D-MANETs. In this paper, we introduce a peer-to-peer JavaSpaces implementation that we specifically designed for D-MANETs, and with which pre-existing or new JavaSpaces-based applications can be easily deployed in such networks.

Keywords *peer-to-peer computing; opportunistic networking; D-MANETs; coordination middleware; JavaSpaces; Future object.*

* Corresponding author. Tel: +33297017216, Fax: +33297017279

1 INTRODUCTION

A mobile ad hoc network (or MANET) is a dynamic wireless network that requires no fixed infrastructure. It is generally formed spontaneously by a collection of wireless nodes without the aid of any centralized administration. Each mobile host can communicate with its neighbors using direct pair-wise wireless links. Communications in MANETs have been enhanced over the years thanks to multi-hop forwarding protocols, such as OLSR, AODV, DYMO, DSR, etc. [1].

Yet most of these protocols rely on the assumption that the whole MANET remains continuously connected, i.e., between any pair of hosts in the MANET, there actually exists at least one temporaneous end-to-end path. Unfortunately, this assumption does not hold in real conditions; many real MANETs are, under the most favorable conditions, only partially or intermittently connected.

The sparse or irregular distribution of hosts in a MANET can, for example, induce link disruptions in the whole MANET. These disruptions may in turn split the whole MANET into a collection of distinct, continuously changing, disconnected "islands" (connected components) as shown in Fig. 1. This kind of MANET is called a Disconnected MANET (D-MANET).

The "*store, carry and forward*" approach is the foundation of Delay/Disruption-Tolerant Networking (DTN) [2]. In a D-MANET, it can help bridge the gap between non-connected parts of the network: the mobility of hosts makes it possible for messages to propagate network-wide by using mobile hosts as carriers (or *data mules*) that can move between network islands. As shown in Fig. 1, connectivity disruptions between islands 1 and 2 can for example be tolerated thanks to users moving (deliberately or by chance) between these islands. The device of a user moving from island 1 to island 2 acts as a data mule for messages addressed to hosts located in island 2. Considering that many carriers may be involved successively for the transmission of a single message, this approach provides message delivery at the price of additional transmission delays. Fig. 1 shows that the transmission of a message from island 1 to island 4 can for example involve two message carriers: first a carrier moving from island 1 to island 2, and then another carrier –or the same one– moving from island 2 to island 4.

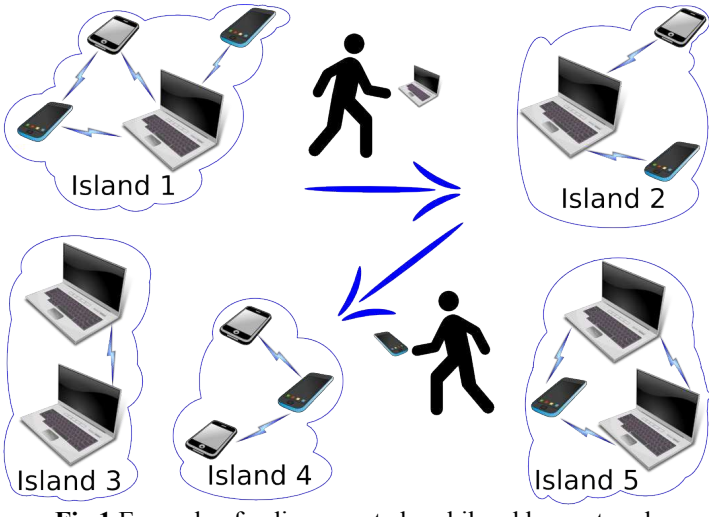


Fig.1 Example of a disconnected mobile ad hoc network

In the DTN community, some approaches make the assumption that communications between the hosts can be predicted accurately, and routing strategies can be thus devised based on contact predictions. But in most real D-MANETs, communications are not planned in advance and can hardly be predicted, especially if the physical carriers are for example human beings carrying laptops or smartphones. The term opportunistic networking is often used to denote such disruption-tolerant networks where contacts must be exploited opportunistically [3]. Each contact thus

represents an opportunity for two hosts to exchange messages. Consequently, communication protocols for D-MANETs usually provide no more than best-effort delivery. Consider again the example shown in Fig. 1, and assume a message is addressed by a host in island 2 to a host in island 3. If no human carrier ever visits island 3, then there is no chance that the message ever gets delivered in this island.

The dynamic nature of D-MANETs creates many challenges for application developers. First, as a general rule, no host in a D-MANET can be considered as stable and accessible enough to play the role of a server for all the other hosts. Consequently, applications developers should generally use a peer-to-peer model rather than a client-server one. Second, developers should take into consideration long transmission delays while writing their applications and make them as asynchronous as possible. Finally, developers should consider possible transmission failures and design their applications properly to tolerate such failures.

All these reasons yield an increasing need for a special kind of middleware system that, while coping with D-MANETs issues, provides the developers with a set of APIs that eases the development of distributed applications for D-MANETs. Already existing middleware solutions cannot satisfy the special requirement of D-MANETs for the following reasons:

- They have been mainly designed to use the client-server model.
- They support only synchronous operations.
- They tolerate neither long transmission delays nor transmission failures.

Therefore a new middleware implementation must be specifically designed to deal with the previously-mentioned constraints of D-MANETs.

The characteristics of D-MANETs favor a middleware which supports a decoupled and opportunistic style of computation: decoupled in the sense that it proceeds even in presence of disconnections, and opportunistic as it exploits connectivity whenever it becomes available. According to Mascolo et al. [4], the decoupled style of communication has been addressed by tuple-based systems which have been shown to provide many useful facilities for mobile networks, although they have not initially been designed for such environments. Hence, we argue that tuple-based systems are well suited to D-MANETs. However, traditional tuple-based implementations cannot run satisfactorily in D-MANETs and some modifications are required to make them better suited for such environments. First, the shared data space cannot be presented to mobile hosts as a centralized repository service anymore. Second, synchronous communication mechanisms supported by traditional implementations must be replaced by asynchronous mechanisms. Finally, as for the communication mechanisms, the synchronous primitives supported by traditional implementations must be also replaced by asynchronous primitives.

In the remainder of this paper, we present a peer-to-peer tuple-based implementation specifically designed for D-MANETs. It is actually an implementation of the JavaSpaces specification [5]. This implementation notably leverages on Future object [6] to extend the standard specification with asynchronous operations to cope with the dynamic nature of D-MANETs. By implementing the standard JavaSpaces specification, any pre-existing distributed application based on the JavaSpaces API can be executed in a D-MANET with no further development. Furthermore, developers can get benefits from the asynchronous operations to improve their applications to be well-suited for use in large and dynamic networks such as D-MANETs.

This paper is structured in the following way: a background of the specification of JavaSpaces and of Future object is presented briefly in Section 2. Section 3 presents our middleware architecture, along with details about its implementation. Evaluation results are presented in Section 4. Discussion is then reported in Section 5. Section 6 discusses related work. Section 7 concludes this paper and describes our plans for future work.

2 BACKGROUND

This section provides an overview of the specifications of JavaSpaces and of Future object successively.

2.1 JavaSpaces Background

The JavaSpaces technology is a Java specification of the concept of tuple space, which was originally introduced in the Linda programming language [7]. In this section, we provide a brief introduction to this concept as introduced in Linda, and to the JavaSpaces technology, which provides an implementation for Java applications. Furthermore, the shortcomings of JavaSpaces in the context of D-MANETs are also presented in this section.

2.1.1 Tuple Space

The tuple space concept has its root in the Linda parallel programming language developed at Yale University [7]. A tuple space is a shared data space acting as an associative memory used by several clients, called "*processes*", for communication and/or coordination requirements. A Linda application is viewed as a collection of processes cooperating via the flow of data structures, called "*tuples*", into and out of a *tuple space*. Each tuple is a record of typed fields containing the information to be communicated. The coordination primitives provided by Linda allow processes to insert a tuple into the tuple space (*out*) or retrieve tuples from the tuple space, either removing those tuples (*in*) or preserving the tuples in the space (*read*). For retrieving operations, tuples are selected using simple pattern-matching from a given set of parameters.

The most interesting feature in the tuple space programming model is that it provides space and time decoupling. On the one hand, *space decoupling* means that a tuple placed in a tuple space may originate from any sender process and may be delivered to any potential recipient. On the other hand, *time decoupling* means that a tuple placed in a tuple space will remain there until it is removed explicitly (potentially indefinitely), and hence the sender and receiver do not need to overlap in time. Together, these features provide an approach that is fully distributed in space and time. Such an approach is paramount in a mobile setting, where the mobile hosts involved in communication change dynamically due to their migration or connectivity patterns.

2.1.2 JavaSpaces

JavaSpaces is a Java specification of the tuple space concept, introduced as a part of the Java Jini technology. It defines a set of application programming interfaces (APIs) that extend the simple core of Linda primitives. The JavaSpaces version of Linda tuples, called "entries", are Java objects that contain public fields that act as typed fields in Linda. JavaSpaces supports also a special kind of entry, called *template*, which is used to characterize the kind of entries a process is looking for. That is, a process uses a template to find *matching entries* whose types and fields match the template. JavaSpaces provides *read*, *take* and *write* operations in order to implement Linda's *read*, *in*, and *out* operations respectively. Additionally, it provides a *notify* operation that allows processes to be informed of the existence of a matching entry. This operation notifies the processes by sending a special object called *event* containing information, to which the processes react. It is worth noting that both *read* and *take* operations are blocking operations in which the calling process is suspended while waiting the answer. JavaSpaces also provides *immediate return* versions of *read* and *take* operations, with which processes query a space and immediately return either a matching tuple or a *null* indicating that no matching entry exists. These operations, called *readIfExist* and *takeIfExist*, can be useful when a process requires an immediate answer.

As the entries of JavaSpaces are passive data, processes cannot perform operations on tuples directly. In order to modify an entry, a process must explicitly remove, update and reinsert it into the space.

2.1.3 Shortcomings of JavaSpaces Implementations

Many JavaSpaces implementations have been designed and are successfully used for stationary distributed systems running in fixed networks. Yet they do not appear to be suitable for D-MANETs.

These implementations are all based on the client-server model, which assumes the existence of stable fixed servers. Such servers must be constantly reachable and available for every hosts. This model contrasts with the extremely dynamic nature of D-MANETs, where the location of each hosts can change continuously, so that no host

can be considered as reachable and reliable enough to play the role of a server. It is therefore necessary to use a peer-to-peer model to support D-MANETs.

Furthermore, these implementations mainly support synchronous communication systems that require the process asking for a service, and the server delivering that service, to be up and running simultaneously. In a D-MANET, it is often observed that processes are not connected at the same time, because of deliberate disconnections (e.g., to save battery) or involuntary ones (e.g., no network coverage). In order to cope with this limitation, an asynchronous form of communication is necessary in order to allow interaction between processes that are not directly connected.

Finally, in the standard JavaSpaces programming model blocking operations are natural, because unless a process has data to work with it is suspended. Thus, the basic JavaSpaces operations (*read* and *take*) are synchronous (blocking) operations. According to Roman et al. [8], when JavaSpaces systems need to support mobile hosts in ad hoc networks, their operations should not be synchronous. Thus, if some needed entry is not available, processes should be able to switch to another task rather than block. Supporting the dynamic nature of ad hoc environments, as reported by Roman, requires extending the basic JavaSpaces operations with novel constructs to improve the performance of JavaSpaces and its programming flexibility. Indeed, several JavaSpaces implementations designed specifically for ad hoc networks support non-blocking operations. A good survey of these JavaSpaces implementations can notably be found in [9]. Our JavaSpaces implementation likewise extends the standard specification with support of asynchronous operations using the technique of Future object [6], as explained in the next section.

2.2 Background of Future Object

According to the Future object specification [10], a Future object acts as a *proxy* (placeholder) for the result of not-yet-performed asynchronous computation. A Future object allows client to continue computation without being blocked waiting for a not-yet-available answer.

The result of an asynchronous computation can only be retrieved using the method *get* when the computation has completed, blocking if necessary until it is ready. As a consequence, using a Future object can improve computation as that the result acquisition can be achieved concurrently with the calling process, as long as the calling process does not need to invoke methods on the returned object. This trend is illustred in Fig. 2.b, which shows the timeline of a process using Future object compared with that without Future object in Fig. 2.a. However, if the calling process needs to use the returned object, it is then automatically blocked if the result is not yet available as shown in Fig. 2.c. Cancellation is possible using the *cancel* method, so that the calling process stops waiting for the result and any resource associated with the Future object can be garbage-collected. Additional methods are provided to determine if the task completed normally or was canceled.

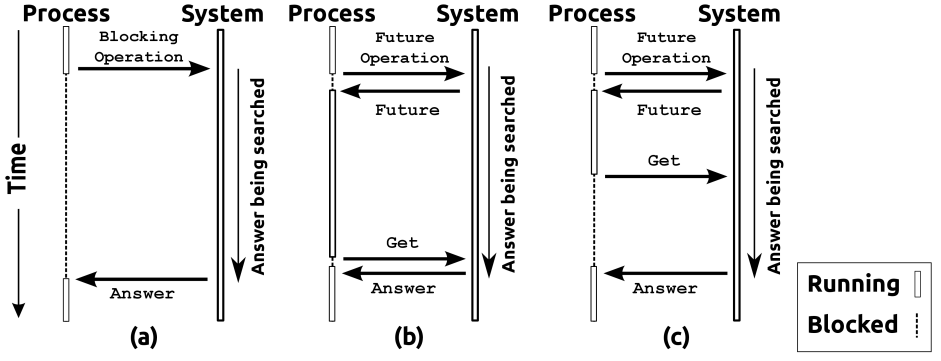


Fig.2 Examples of use of Future object

As a result, the programming model of JavaSpaces in the presence of mobility, high network dynamism and disconnections can be improved by the provision of asynchronous operations that allow non-blocking access to the space. To do so, our JavaSpaces implementation leverages on Future object to provide a pair of asynchronous

operations: *readf* (read in the future) and *takef* (take in the future).

3 JAVASPACES IMPLEMENTATION FOR OPPORTUNISTIC NETWORKS

The JavaSpaces technology has been primarily designed to provide persistent object exchange areas (spaces), through which processes coordinate actions and exchange data, and this has remained its typical usage scenario. As mentioned in Section 2.1.3, most JavaSpaces implementations are server-based systems, where centralized servers are used to manage shared spaces. Furthermore, these servers are only accessible using a set of synchronous operations, which is a consistent approach in a network where continuous connectivity between clients and servers can be assumed. As explained in the former sections, a server-based system is hardly compatible with the characteristics of D-MANETs, where no host can act as a reliable server for all the other hosts. Moreover, synchronous interactions between processes in D-MANETs should be avoided because of the long transmission delays that can be expected in such networks. A peer-to-peer, flexible and delay/disruption-tolerant JavaSpaces implementation must therefore be developed in order to provide JavaSpaces services for D-MANETs.

JION (*JavaSpaces Implementation for Opportunistic Networks*) is the JavaSpaces implementation we designed along that line. Its general architecture is shown in Fig. 3. JION is composed of two basic modules: a communication system, and the JavaSpaces system.

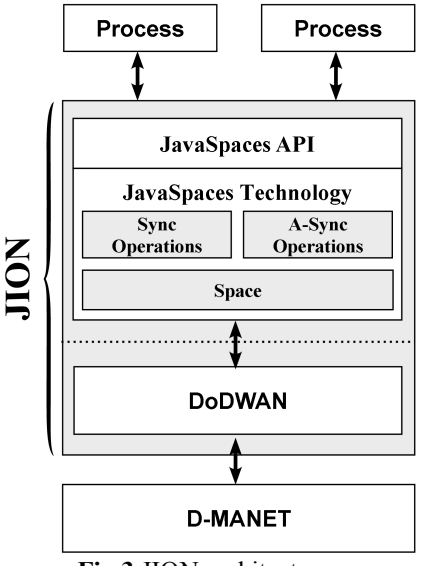


Fig.3 JION architecture

3.1 Communication Middleware

Building any application for D-MANETs requires some communication middleware system, with which hosts can collaborate in a peer-to-peer manner to ensure message transportation, while dealing with high latency and link disruptions. JION relies on a communication middleware system called DoDWAN *(Document Dissemination in mobile Wireless Ad hoc Networks) [11],* which has been designed in our laboratory and is now distributed under the terms of the GNU General Public License[1].

DoDWAN supports content-based information dissemination in D-MANETs. In content-based networking, information flows towards interested receivers rather than towards specifically set destinations. This approach notably fits the needs of applications and services dedicated to information sharing or event distribution. It can also be used for destination-driven message forwarding, though, considering that destination-driven forwarding is simply a particular case of content-driven forwarding where the only significant parameter for message processing is the

1 http://www-irisa.univ-ubs.fr/CASA/DoDWAN

identifier of the destination host (or user).

Messages in DoDWAN are composed of two parts: a descriptor and a payload. The payload is simply perceived as a byte array. The descriptor is a collection of attributes expressed as *(name, value)* tuples. These attributes can be defined freely by the developers of application services built on top of DoDWAN.

DoDWAN implements a selective version of the epidemic routing model proposed in [12]. It provides application services with a publish/subscribe API. When a message is published on a host, it is simply put in the local cache maintained on this host. Afterwards, each radio contact with another host is an opportunity for the DoDWAN system to transfer a copy of the message to that host whenever it is interested.

In order to receive messages, an application service must subscribe with DoDWAN and provide a *selection pattern* that characterizes the kind of messages it would like to receive. A selection pattern is expressed just like a message descriptor, except that the *value* field of each attribute contains a regular expression. The selection patterns specified by all local application services running on the same host define the *interest profile* of this host. DoDWAN uses this profile to determine which messages should be exchanged whenever a radio contact is established between two hosts. Details about this interaction scheme and about how it performs in real conditions can be found in [11].

As a general rule, a mobile host that defines a specific interest profile is expected to serve as a mobile carrier for all messages that match this profile. 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 setting the configuration parameters of DoDWAN accordingly.

Mobile hosts running DoDWAN only interact by exchanging control and data messages encapsulated in UDP datagrams, which can themselves be transported either in IPv4 or IPv6 packets. Large messages are segmented so that each fragment can fit in a single UDP datagram. Fragments of a large message can propagate independently in the network and be reassembled only on destination hosts.

3.2 JavaSpaces

According to the JavaSpaces specification, processes coordinate by exchanging entries through the tuple space using a simple set of operations. Entries and operations represent the basic JavaSpaces elements. This section describes the upper layer of JION and its entry module, along with the supported operations.

3.2.1 System Model

JION is a distributed peer-to-peer JavaSpaces implementation, as it is intended to be used in D-MANETs where server centralization is impractical. Each host maintains a local space, in which JION stores the entries produced locally (that is, entries produced by *write* operations initiated by local processes). Each host is thus responsible for managing its own entries. If entries were propagated all over a D-MANET and managed in a collaborative manner, this could result in *orphan entries*: to the best of our knowledge, consensus has not been solved yet in D-MANETs. Consequently, hosts in a D-MANET could not agree to remove any entry from the space, for example when a process wants to *take* it.

For the sake of illustration, imagine that an entry has been propagated in the D-MANET shown in Fig. 1. Each host now has its own copy of the given entry. When a process in island 1, for example, wants to *take* the entry, a *take* request should be sent to all the hosts in the D-MANET in order to delete the given entry from their cache. If no user ever visits island 5 for example, the copies of this entry in this island will become orphan entries.

In order to circumvent this problem, *write* operations in JION are only processed locally, while matching and fetching operations (*read*, *take* and *notify*) are processed by querying hosts over the network for the entries they own.

3.2.2 Entries and Templates

According to the JavaSpaces specification, an *entry* is an object reference characterized by its "*fields*". In the JavaSpaces terminology, entry fields refer only to the public attributes of the entry objects. In fact, entry fields are meant to act as a set of attributes characterizing an entry, and they are used while performing matching operations

when retrieving entries from the space. Templates are special entry objects. The values of their fields are used for matching operation.

As mentioned in Section 3, a DoDWAN message has two parts: a descriptor and a payload. Entries and templates are simply carried in the message as its payload, and considered as a simple byte array. Since the descriptor of DoDWAN is meant to characterize the content of the message, JION tags the message to declare (describe) its actual content in order to specify if it contains an entry or a template.

For the sake of illustration, we consider a Carpooling application which allows users to share empty seats during a ride with fellow commuters on the same route. Users can declare the ride for which a seat is available using the following class *Seat*:

```
public class Seat implements Entry{
     public String departure;
     public String arrival;
     public Date date;
     public Seat(){};
     public Seat(String departure, String arrival, Date date){
            this.departure=departure;
            this.arrival= arrival;
            this.date= date;}
}
```

Using class *Seat*, an example will be developed through this section and the next one, which aims for better understanding the basic principles of JavaSpaces' programming model. For this example, we always consider the D-MANET shown in Fig. 1.

Two seats (also called entries in JavaSpaces terminology) can be created using class *Seat* in the following manner:

```
Seat offer1 = new Seat ("Paris", "Berlin", new Date (2013, 9, 22));
Seat offer2 = new Seat ("Paris", "Madrid", new Date (2013, 8, 17));
```

These entries are characterized by three fields: *departure, arrival* and *date*; which are to be considered during matching operations. For the matching operations, it is worth taking into consideration that JION implements the same matching algorithm defined in the JavaSpaces specification.

For the sake of illustration, three requests (or templates in JavaSpaces terminology) are created as shown below:

```
Seat request1 = new Seat ("Paris", "Berlin", null);
Seat request2 = new Seat ("Paris", "Zurich", new Date (2013, 10, 17));
Entry request3 = new Entry();
```

According to the JavaSpaces specification, the template "*request1*" matches the entry "*offer1*" for the following reasons: first, the fields *"departure"* and "arrival" in the template matches exactly *by the value* in the same field of the entry. Second, the field *"date"* with null reference in the template is considered as a wildcard, hence it can match any value in the same field of the entry. Finally, the entry and the template match *by the type* (i.e., both of them have the same type since they are defined using class *Seat*). As a result, when an operation provides "*request1*" as a template, it can have the entry "*offer1*" as an answer. However, on the one hand, the template "*request2*" does not match any entry for a *value mismatching* reason. On the other hand, the template "*request3*" does not match any entry for a *type mismatching* reason. More details about the matching algorithm can be found in [5].

3.2.3 Operations

According to the JavaSpaces specification, access to entries must be done through a set of basic operations, which are: *write*, *read*, *readIfExists*, *take, takeIfExists* and *notify*. Both *read* and *take* are blocking, that is, if no matching entry is available the process performing the operation is suspended until a matching entry becomes

available. Furthermore, JION leverages on Future object to provide a pair of asynchronous operations: *readf* and *takef* .

Below is a description of the way JION supports these operations.

write: this operation stores a new entry into the local space of host for a specific period of time, called a *lease*. A lease represents the lifetime of the entry. It is worth noting that each entry is only stored on the local host and is not replicated over the D-MANET. Therefore, it is up to each host to monitor its local space and manage its own entries, and especially ensure that out-of-date entries are removed.

For our example, we suppose that a user (or a process in JavaSpaces terminology) called *P1* in island 4 has an available seat from *Paris* to *Rome* on Sunday 22/09/2013, so it writes the following entry in the space:

```
Seat offer = new Seat ("Paris","Roma", new Date (2013,9,22));
```

As mentioned above, the entry *"offer"* is only stored locally and is not disseminated over the D-MANET.

read: this synchronous operation requests the JION service to locate an entry that matches the template provided as a parameter. When a process on a host performs a *read* operation, the local space of the host is queried first in order to find a matching entry. If no matching entry is found, the process is suspended for a certain amount of time specified in the lease provided as a parameter. Meanwhile, JION disseminates the request over the D-MANET using the lease time as a deadline for the given request. Each host, when it receives this request, queries its local space to find a matching entry and forwards a copy of this entry (if any) back to the requesting process. Upon receiving the answers sent from different hosts in D-MANET, JION uses its *selection policy* and chooses an answer to the *read* operation and resumes the requesting process. According to the JavaSpaces specification, each implementation can choose its proper selection policy. For efficacy issues, the selection policy supported by JION chooses always the first available answer as an answer for the calling operation.

It is worth noting that if the time for the lease has expired and no answer has been received, JION deletes the request and resumes the calling process to inform it about this issue.

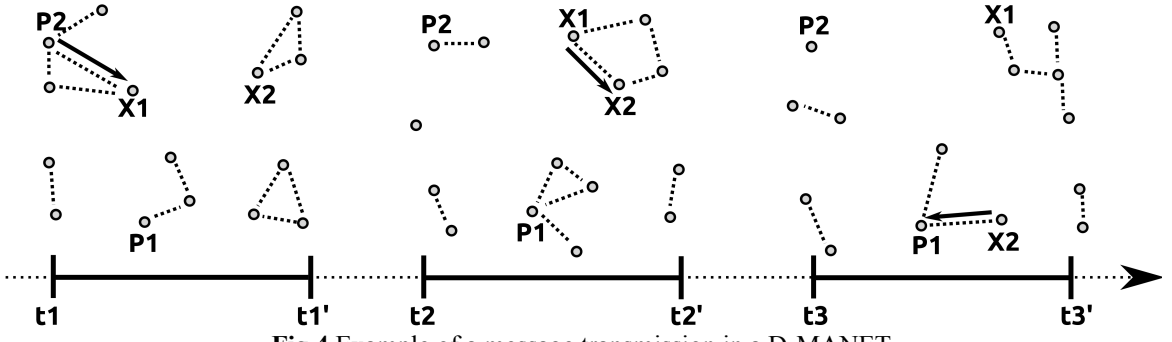


Fig.4 Example of a message transmission in a D-MANET

For our example, imagine that a user *P2* in island 1 wants to know when it is possible to go from *Paris* to *Roma*, so that it provides JION with the corresponding template which is shown here:

```
Seat request = new Seat ("Paris", "Roma", null);
```

In order to transfer this request to *P1* which has a matching entry, *P2* transmits a copy of the request to *X1* with which it has a contact during the interval (t_1 , $t_1^{'}$) as shown in Fig. 4. Then, the carrier *X1* moves and becomes *opportunistically* connected with *X2* during the interval (t_2 , $t_2^{'}$) in which it transmits a copy of the request to *X2*. In the same way, while *X2* is moving, it transmits a copy of the request to the final destination *P1* after an arbitrary disconnectivity interval ($t_2^{'}$, t_3). The process *P1*, when receiving the request, forwards a copy of *offer* back to *P2* using the mobility of hosts between islands. As a result, *P1* knows that a such ride is available on Sunday 22/9/2013.

Operation *readIfExists* is also supported by JION. According to the specification, the key feature of this operation is that it provides an immediate answer. To do so, it queries only the central server and immediately returns either a matching entry or a *null* indicating that no matching entry exists. Taking into consideration the absence of such central entity in D-MANETs, the semantic of immediate answer can only be obtained by querying locally. For this reason, the *readIfExists* in JION queries only the local space to find an entry that matches the specified template. The request is not disseminated over the D-MANET.

take: this synchronous operation basically performs the same function as *read*, except that it removes the matching entry from the space. JION first searches the local space. If no match is found, it suspends the process for the lease time provided as a parameter while searching the answer. It starts by querying all the hosts over the D-MANET in order to discover which hosts (if any) have a matching entry. The hosts send back proposals to the calling process. Upon receiving the proposals, JION uses its selection policy to select one host, from which the entry should be taken. JION then asks the chosen host to permanently remove the matching entry from its local space and hand it back to the requesting process.

If the lease expires and no answer has been received, then the operation terminates and the process is resumed to get informed about the issue.

The entire operation may take more time than the *read* operation since it consists of up to four message disseminations in the network while only two messages are required in the *read* operation.

JION also supports a *takeIfExists* operation, which performs exactly like the corresponding *readIfExists*, except that a matching entry is removed from the local space.

readf and takef : these operations are asynchronous versions of *read* and *take* operations respectively. They allow non-blocking access to the space using the technique of Future object. During the whole operation, the calling process continues executing its code without being blocked as if the operation had been effectively performed, as long as the calling process does not need the real answer. The calling process will only be blocked if it does need the actual answer, and that answer is not yet available.

When invoking *readf* or *takef* , as shown in Fig. 2.b, a new Future object is created and returned instantly as the result of the invocation. So the calling process is not blocked waiting for the answer. Meanwhile, JION behaves as if it were a normal *read* or *take* operation. Upon the completion of the operation, the answer is assigned to the Future object. It is now possible for the calling process to get the result using method *get*.

notify: this operation informs a process when entries matching a set template are written in the space.

When a process is interested about specific entries, it provides JION with the corresponding template. JION disseminates the template all over the hosts in the D-MANET. The hosts register the request in the hope that a matching entry will be written before the lifetime of the request expires. Consequently, when a matching entry is written in a host, the host sends an event object containing information about this entry to the requesting process.

For our example, we suppose that a user *P3* in island 2 is interested about specific rides. If a matching ride is created in island 1, the user moving (deliberately or by chance) from island 1 to island 2 can serve as a data mule for the transmission of the event to *P3*. However, if a matching ride is written in island 5 and no human carrier moves from island 5 to island 2, then there is no chance that the event ever gets delivered from this island.

3.2.4 Transactions

According to the JavaSpaces specification, it is possible to group multiple operations (participants) into a bundle that acts as a single atomic operation. This is done using the optional concept of transaction. Either all operations within the transaction are performed or none is. In fully-connected stable networks, a transaction is controlled by a specific manager (server), which should always be reachable by all the participants. If a participant is momentarily disconnected, the whole transaction is aborted. Considering that hosts in D-MANETs can neither rely on a reliable

server nor reach a consensus, it is not possible to ensure transactions as defined by JavaSpaces. For this reason, JION does not support the concept of transaction and each operation is considered as a singleton operation: an operation that does not consider the states of other operations while performing its tasks.

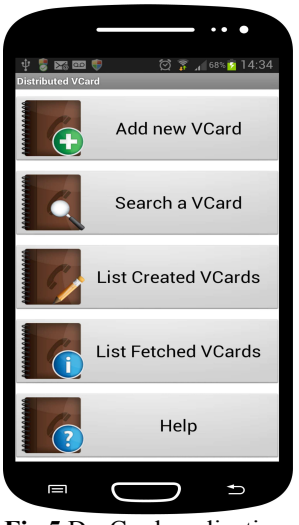


Fig.5 D-vCard application

4 EVALUATION

A D-MANET is a wireless network whose topology is continuously changing, and where radio contacts between mobile hosts do not necessarily follow any predictable pattern. Therefore, protocols and systems designed for D-MANETs are usually evaluated using network simulators. The originality of our work lies in the fact that JION has been fully implemented in Java, and can thus be tested in real-life experiments on Linux and Android platforms. JION is distributed under the terms of the GNU General Public License[2]. It has seen extensive testing to examine how well it performs in D-MANETs. While conducting these tests we strived to evaluate how easy it is for an application developer to implement a distributed application using JION. Furthermore, we have evaluated its efficiency in real conditions.

4.1 Developing Distributed Applications with JION

JION implements Sun Microsystems' JavaSpaces Technology specification, provided as a part of the Java Jini Technology [5]. Since it implements a well-known middleware specification, 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 standard JavaSpaces application, and JION will take care of its execution in a D-MANET. Indeed, any pre-existing JavaSpaces application can be deployed using JION, without any change in its source code. As a proof-of-concept, an already existing JavaSpaces application has been deployed successfully over JION without any change in its source code. The whole source code is provided in Appendix 9.1. However, developers should always take into consideration the specific constraints posed by D-MANETs where transmission delays and message delivery are usually not guaranteed. This issue is discussed further in Section 5.

For testing and evaluation purposes, we have developed a distributed vCards directory system (D-vCard) for Android platforms based on JION, as shown in Fig. 5. D-vCard is a distributed address book designed to allow owners of Android smartphones to access and share contact data based on unpredictable contacts between their smartphones. D-vCard complies with the vCard electronic business card specification, as defined in RFCs 6350 [13]. A vCard is a standard for representing the kind of information required for every contact in an address book or email application: name, address information, phone numbers, etc. It basically consists of field-value pairs separated by ":". The specification defines all possible fields and the format for their values.

2 http://www-irisa.univ-ubs.fr/CASA/JION

In the application D-vCard, creating a new vCard is simply implemented using *write* operation of JION: a vCard entry is created and stored locally on the space which acts as a local directory. Similarly, *searching*/*deleting* a vCard in/from the space is performed by creating a template having the required information as fields. This template can then be passed as a parameter to the *read*/*take* operation. Once a user gets a vCard, he/she will automatically receive notifications about any new modification on the given vCard thanks to the *notify* operation.

4.2 Experimental Evaluation

In order to evaluate the performance of JION, we conducted two different measurement campaigns: first within a single connected island, and then in a real disconnected network involving several user-carried smartphones running D-vCard.

Using the traces collected by JION during these two campaigns, we have computed several performance metrics. The campaigns along with their results are described in details in the next sections.

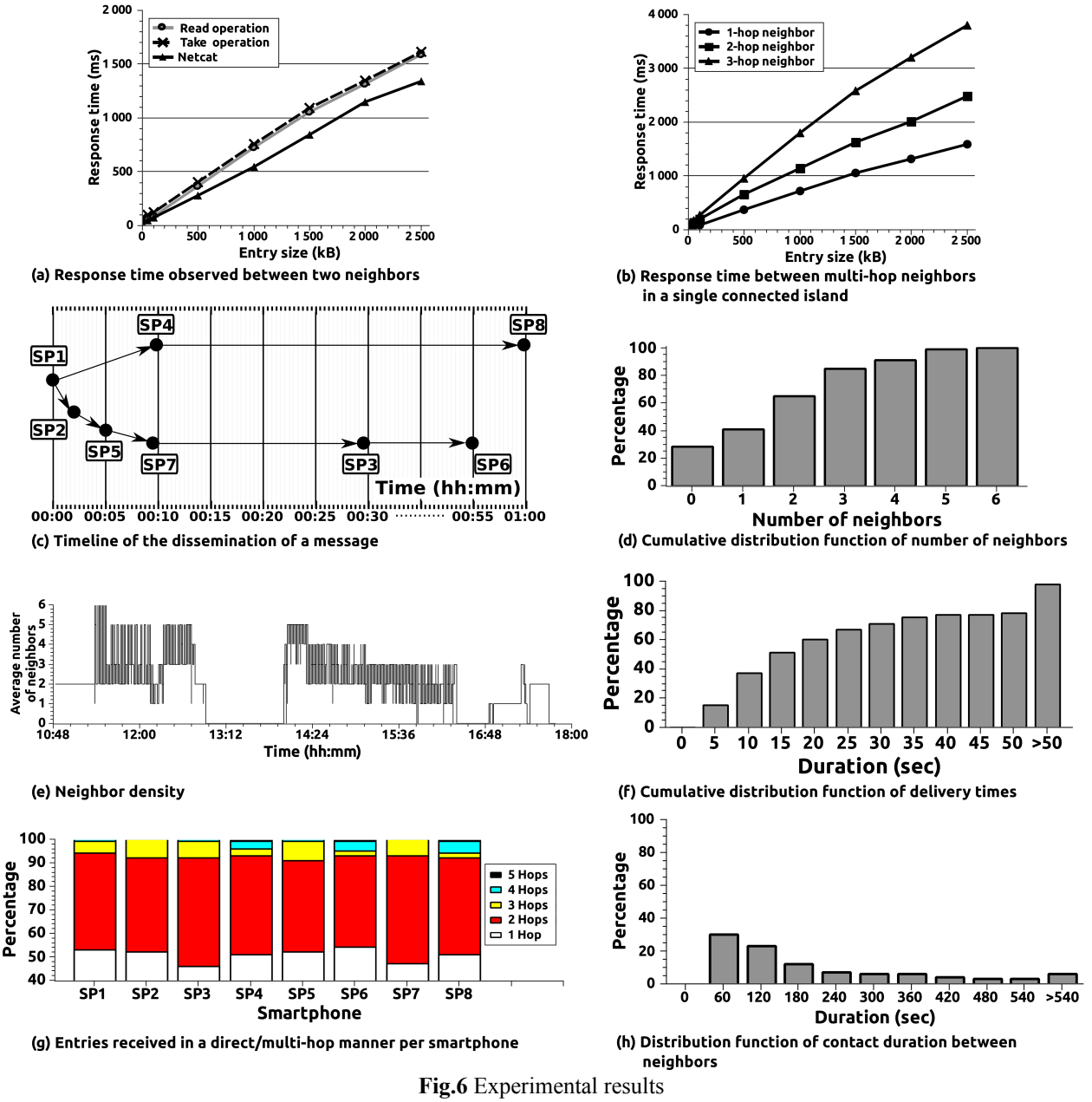


Fig.6 Experimental results

4.2.1 Efficiency of JION over a Single Connected Island

Before trying to observe how entries can propagate between several islands in a disconnected network, one can first try to measure how fast they can propagate within a single island where transmission is not hampered by connectivity disruptions. Since JION is implemented on top of the DoDWAN communication system, which itself relies on UDP transmissions, our first objective was to evaluate how our multi-layer middleware architecture performs over the underlying wireless transmission medium.

We first used two netbooks A and B, running JION over a Linux operating system. These netbooks were installed next to each other in the same room, and their built-in Wi-Fi 802.11bg chipsets were configured to operate in ad hoc mode. We actually focused on the response time, which is here defined as the time interval between the time netbook A sends a request to *read/take* entries of different sizes to netbook B and the time when it actually receives them. In order to get reference values regarding the capacity of the wireless link at application-level, we used the basic Netcat

(nc) networking utility in UDP mode, that can read and write chunks of data across network connections. 200 series of tests were conducted in this scenario. The results of these tests are presented in Fig. 6.a. *Read* and *take* operations show similar performance. However, JION shows about 20% overhead over Netcat. Considering that the communication middleware of JION (DoDWAN) implements a sophisticated opportunistic protocol in order to orchestrate communications between neighbor hosts, we consider that these results are quite reasonable.

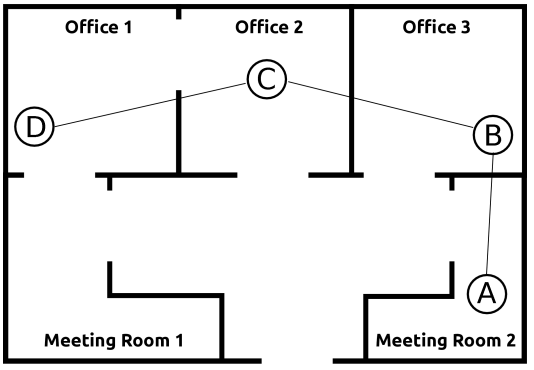


Fig.7 Multi-hop forwarding test scenario

Another real world test was conducted to investigate the behavior of JION when messages can propagate over multiple hops. The tests was carried out with four netbooks (A, B, C and D) distributed in our laboratory. Because of the effect of concrete walls on signal attenuation, the connectivity between these netbooks was such that netbook B could only communicate with A and C, while netbook D could only communicate with C as shown in Fig. 7. This test relied on *write/read* operations: a total amount of 225 entries were written on B, C, D, and host A was configured to read these entries. We measured the average time required for these entries to reach host A. The results of this test are shown in Fig. 6.b. It can be observed that the delay before host A gets an entry changes with the size of the entry and the number of transmission hops. This is because when host B serves as a relay between its neighbors 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 hosts B and D. It must also be considered that DoDWAN strives to ensure a high delivery ratio, so entries are retransmitted again and again if they are not received in the first place. Indeed, during the test all entries got received by host A, so that the delivery ratio was 100% in that case.

4.2.2 Efficiency of JION in a Real D-MANET

The previous tests demonstrated that JION performs satisfactorily in a traditional connected environment and provides a reasonable response time. However, JION becomes really beneficial when the network is not connected, as shown in Fig. 1. Consequently, after measuring how JION can perform in a single, connected island we used our D-vCard application to observe how it can perform in a real D-MANET. Eight volunteers from our laboratory were equipped with HTC smartphones running D-vCard. During this campaign, which lasted for 8 hours, the volunteers were asked to carry their smartphones whenever possible –and use D-vCard services of course– while roaming the laboratory building or its surroundings.

Fig. 6.c illustrates how one particular “notify request” message disseminated during one of the trials. This message was first published by smartphone *SP1*. After only a few minutes *SP1* established radio contact with *SP2*, which thus got a copy of the message and became a new carrier for this message. *SP1* later managed to forward the message to *SP4*, while *SP2* forwarded it to *SP5*. The message thus kept disseminating, until it reached the last smartphone *SP8*, about one hour after it was initially published. This example confirms the delay/disruption-tolerant nature of transmissions.

An important factor in D-MANETs is how often the mobile hosts meet each other. During this campaign, the cumulative distribution function (CDF) for the number of neighbors for each device is shown in Fig. 6.d. It can be

noticed that the average number of neighbors for each smartphone was about two neighbors during about 65% of the campaign duration and no neighbor at all during 25% of the campaign duration. A finer detail of smartphone density, as observed during the campaign, is provided in Fig. 6.e. It is worth noting the frequent disconnections between smartphones during the majority of the campaign duration, which makes a server-based system with synchronous operations totally impractical. A DTN-style communication is thus needed in order to provide communication services between the mobile devices.

When analyzing the traces collected by JION, a special attention was paid to the delivery time of successful operations. The delivery time is calculated as the difference between the time when an operation is started and the time when an answer is received. The delivery times are then represented in cumulative way as shown in Fig. 6.f. The cumulative delivery times were calculated in terms of time slices where a measure of 5 seconds means that the average delivery time observed was between 0 and 5 seconds; a measure of 10 seconds means it was between 0 and 10 seconds, etc. In general, the results show that nearly 80% of the entries got delivered to their destination(s) in less than 50 seconds. Yet, about 3% of the entries could not be delivered. This is the consequence of the unpredictable – yet perfectly legitimate– behavior of the users, which sometimes moved away from the laboratory or switched their devices off in order to preserve their battery budget. By doing so they prevented any further radio contact between their device and those of other users, and this of course led to message loss.

During this campaign, a total number of 3806 radio contacts were established between neighbor smartphones, with an average contact duration of 213 seconds. The contact durations are calculated in terms of time slices and presented in Fig. 6.h. As shown in the figure, the majority of radio contacts lasted less than a minute, which confirms the continuous topological changes of the network during the campaign. Hence, systems that want to rely on these contacts to transport data between smartphones, have to resort to DTN-style communication.

Another important metric that we have also calculated during this campaign is the effect of multi-hop forwarding. Fig. 6.g shows the percentage of direct/multi-hop entries. Direct-entries (shown as 1-hop in the figure) are entries that have been directly received from their original source. Multi-hop entries are entries that have been relayed by several intermediate smartphones before reaching their final destination. For example, the smartphone presented as SP4 in Fig. 6.g received its entries in the following manner: 51% were received directly from the sender (1-hop), 40% through 2 hops, 6% through 3 hops, 2% through 4 hops and finally 1% through 5 hops. It can be observed that around 50% of the entries were received in a multi-hop manner, showing that JION using DoDWAN provides a multi-hop dissemination service that performs satisfactorily in a real D-MANET.

5 DISCUSSION

The results presented in Section 4 confirm that JION is quite efficient in providing JavaSpaces services in D-MANETs. However, we also have to consider the side effects of our technique which involves the implementation of a standard specification, originally designed for “traditional” network, in D-MANETs. Based on the feedback we received from developers using JION, we observe that such technique, if not used carefully, can have both favorable and unfavorable consequences.

On the one hand, 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 standard JavaSpaces application, and JION will take care of its execution in a D-MANET. Indeed, any pre-existing JavaSpaces application can be deployed over D-MANETs easily using JION.

On the other hand, developers should however be aware of the specific constraints posed by D-MANETs, where message delivery, message ordering, and transmission delays are usually not guaranteed. Such constraints are not due to limitations in JION; they are due to the very nature of D-MANETs. Opportunistic protocols and middleware systems designed for D-MANETs can do no magic; they can support network-wide communication in a D-MANET, using mobile hosts as carriers that help bridge the gap between non-connected parts of the network. Yet, unless otherwise specified they do not control how mobile hosts move in the network, so they cannot guarantee that a message will ever reach (or reach in time) any particular host in the network. A developer working on an application

for D-MANETs should therefore assume that delivery failures and late deliveries may be more common than in-time deliveries, and design their distributed application or organize its deployment accordingly.

Erroneous preconceived ideas about JavaSpaces implementations can also play a role. For example, most JavaSpaces applications are built with the implicit assumption that network security is always ensured by JavaSpaces implementations. It is worth taking into consideration that till now there is no standard JINI security model to be used by JavaSpaces implementations. Yet, the only security available to users of all current JavaSpaces implementations is provided by their communication protocols. Similarly, the security provided by JION is based on the security system supported by DoDWAN. This system allows mobile hosts that run a distributed application to exchange messages that are encrypted with a shared symmetric key. Mobile hosts that share the same tuple space using JION can therefore benefit from this feature when writing, reading and taking entries to and from the space. One of the best ways to avoid problems caused by erroneous preconceived ideas about JavaSpaces implementations is in fact to read carefully their specification in order to design applications accordingly.

To assess the advantages of using Future object, it is worth noting that performing blocking operations, as provided in standard JavaSpaces implementations, is not preferable on distributed systems. It is impractical for an application to be blocked waiting for an answer for an unknown amount of time. In such case, it will be always considered as an *unresponsive* application. A standard way to overcome this limitation of blocking operations is often to design multithreaded applications. However, the use of Future object, from a readability point of view, is quite important since it permits to make the code more *readable,* more structured and therefore more intuitive. For the sake of illustration, in Appendix 9.2 a *non-blocking* JavaSpaces application is first written using the technique of *Threads*. Then, it is presented how to rewrite this application using Future object. Comparing the two versions of the application, one can argue that non-blocking JavaSpaces application can be easily achieved using Future object while keeping its source code human-readable and maintainable as well. (developers can easily achieve a more readable and maintainable non-blocking application using Future object rather than using Threads.)

6 RELATED WORK

In the last few years, several projects revisited Linda [7], especially in the context of mobile ad hoc environments.

Both Ara [14] and LIME [15] are coordination middleware systems implementing tuple spaces stored on hosts acting as servers. These middleware systems target mobile ad hoc networks. They provide JavaSpaces services to mobile hosts that are in communication range of the servers. Since they rely on a server-based model, they are hardly usable in real D-MANETs. Limone [16] is a lightweight alternative to LIME requiring far less overhead. Limone is based on the premise that a single round-trip message exchange is always possible, making it impractical over D-MANETs. In contrast, CAST [17] is a server-less coordination middleware for MANETs. Since it does not rely on any centralized service, this middleware suits well the dynamics of wireless open networks. CAST makes it possible to process operations even when no end-to-end route exists between the hosts involved, by implementing a source routing algorithm. This routing strategy relies on the assumption that the motion profile of each host is known. This is clearly a serious constraint, which limits the usability of CAST over the kind of D-MANETs JION targets, where the motions of hosts are neither planned nor predictable. Tuple board (TB) [18] is another server-less coordination middleware for developing collaborative applications running in ad hoc networks of mobile computing devices. Like JION, this middleware has been fully implemented and distributed. It can thus be used and tested in real conditions. However, the proposal is limited to a group of nearby connected devices: when a device leaves the network or turns off, all the tuples posted from this device are withdrawn. The frequency of link disruptions in D-MANETs makes the disconnection tolerance a vital requirement for any middleware that is meant to support D-MANETs.

It can be noticed that all these middleware systems, except LIME and Limone, are not well-suited for D-MANETs by not supporting asynchronous operations. LIME and Limone, as for traditional JavaSpaces implementations, use a basic approach where a timeout can be specified for their blocking operations to avoid locking the system indefinitely.

Furthermore, all middleware systems mentioned in this section define their own communication protocol for route discovery and maintenance. Our work is different as JION presents a two-layer architecture: the upper layer is concerned with tuple space services, while the lower layer supports opportunistic communication. As mentioned above, DoDWAN has been chosen among a few opportunistic communication protocols that are openly distributed to support communications on D-MANETs. Yet, JION could theoretically be implemented above any other communication system, such as DTN2 (a reference implementation of protocols designed by the Delay-Tolerant Networking Research Group (DTNRG) [19]) or Haggle (a *content-centric* architecture for opportunistic communication among mobile devices [20]).

7 CONCLUSION AND FUTURE WORK

In this paper, we have introduced JION (JavaSpaces Implementation for Opportunistic Networks), a peer-to-peer JavaSpaces implementation with support for Future object we designed and implemented specifically for disconnected mobile ad hoc networks (D-MANETs). Using JION, distributed applications based on the concept of tuple spaces (as defined in the JavaSpaces specification) can be deployed and executed in D-MANETs. JION is resilient to the challenges of D-MANETs and provides an effective base which eases the development of applications for D-MANETs. It has been tested on Linux and Android platforms in real conditions and is now distributed under the terms of the GNU General Public License.

Further tests are still under way in order to verify how stable JION is in different kinds of challenged environments. Future work should notably include the assessment of the portability of JION over another communication middleware other than DoDWAN (most likely by implementing it over the DTN2 reference implementation).

8 REFERENCES

[1] C. Liu and J. Kaiser, "A Survey of Mobile Ad Hoc network Routing Protocols," University of Magdeburg, 2005.

[2] K. Fall, "A Delay-Tolerant Network Architecture for Challenged Internets," in *ACM Annual Conference of the Special Interest Group on Data Communication*, New York, NY, USA, 2003, pp. 27–34.

[3] L. Pelusi, A. Passarella, and M. Conti, "Opportunistic Networking: Data Forwarding in Disconnected Mobile Ad hoc Networks," *Ieee Commun. Mag.*, vol. 44, no. 11, pp. 134–141, 2006.

[4] C. Mascolo, L. Capra, and W. Emmerich, "Mobile Computing Middleware," in *In Advanced lectures on networking*, 2002, pp. 20–58.

[5] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces(TM) Principles, Patterns, and Practice*. Prentice Hall, 1999.

[6] E. F. Walker, R. Floyd, and P. Neves, "Asynchronous remote Operation Execution in Distributed Systems," in *, 10th International Conference on Distributed Computing Systems, 1990. Proceedings*, 1990, pp. 253 –259.

[7] N. Carriero and D. Gelernter, "Linda in Context," *Commun Acm*, vol. 32, no. 4, pp. 444–458, Apr. 1989.

[8] G.-C. Roman, A. L. Murphy, and G. P. Picco, "Coordination and Mobility," in *Coordination of Internet Agents: Models, Technologies, and Applications*, 1999, pp. 254–273.

[9] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco, "Tuple Space Middleware for Wireless Networks," in *Middleware for Network Eccentric and Mobile Applications*, B. Garbinato, H. Miranda, and L. Rodrigues, Eds. Springer Press, 2009, pp. 245–264.

[10] J. S. Documentation, "Concurrency Utilities," ORACLE, 2011.

[11] J. Haillot and F. Guidec, "A Protocol for Content-Based Communication in Disconnected Mobile Ad Hoc Networks," *J. Mob. Inf. Syst.*, vol. 6, no. 2, pp. 123–154, 2010.

[12] A. Vahdat and D. Becker, "Epidemic Routing for Partially Connected Ad Hoc Networks," Duke University, Apr. 2000.

[13] Internet Engineering Task Force, "vCard Format Specification." http://tools.ietf.org/html/rfc6350.

[14] H. Peine and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents," in *Proceedings of the First International Workshop on Mobile Agents*, London, UK, 1997, pp. 50–61.

[15] A. L. Murphy, G. P. Picco, and G.-C. Roman, "LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents," *Acm Trans Softw Eng Methodol*, vol. 15, no. 3, pp. 279–328, Jul. 2006.

[16] C. Fok, G. Roman, and G. Hackmann, "A Lightweight Coordination Middleware for Mobile Computing," *Proc. 6th Int. Conf. Coord. Models Lang.*, vol. 2949, pp. 135–151, 2004.

[17] G.-C. Roman, R. Handorean, and R. Sen, "Tuple Space Coordination Across Space and Time," in *Proceedings of the 8th international conference on Coordination Models and Languages*, Berlin, Heidelberg, 2006, pp. 266–280.

[18] A. Kaminsky and C. Bondada, "Tuple Board: A New Distributed Computing Paradigm for Mobile Ad Hoc Networks," *Comput. Syst.*, pp. 5–7, 2005.

[19] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss, "Delay-Tolerant Networking Architecture," *Ietf Rfc 4838*, Apr. 2007.

[20] E. Nordström, P. Gunningberg, and C. Rohner, "A Search-based Network Architecture for Mobile Devices," Department of Information Technology, Uppsala University, 2009-003, Jan. 2009.

9 APPENDIX

9.1 Deploying an already-existing JavaSpaces application over JION

Here we demonstrate how a pre-existing JavaSpaces application can be deployed on a D-MANET using JION. The import statements are not shown and the class *Seat* is defined in Section 3.2.2. As it can be noticed we do not need to change any line in the source code to deploy it on JION.

```
1.  public class HelloWorld{
2.  public static void main (String [] args){
3.  try {
4.      JavaSpace space = JavaSpaceSingleton.getInstance();
5.          if (args[0].equals("write")){
6.                  Seat offer = new Seat ("Paris", "Madrid", new Date (2013,9,22));
7.                  space.write(offer, null, Lease.FOREVER);}
8.                   if (args[0].equals("read")){
9.                  Seat request = new Seat ();
10.                 Seat result = (Seat) space.read (request, null, Lease.FOREVER);}
11.                  if (args[0].equals("take")){
12.                 Seat request = new Seat ();
13.                 Seat result = (Seat) space.take (request, null, Lease.FOREVER);}
14.           } catch (Exception e){
15.         e.printStackTrace();}}}
```

Assuming that JION's package is installed in the home directory, this example can be compiled with the following command:

```
% javac -cp $HOME/JION.jar HelloWorld.java
```

Once the example has been successfully compiled, it can be run on a sender-side terminal in the following manner:

```
% java -cp $HOME/JION.jar HelloWorld write
```

Similarly on a receiver-side terminal for the *read* operation:

```
% java -cp $HOME/JION.jar HelloWorld read
```

Likewise, the *take* operation could be run on a receiver-side terminal in the following manner:

```
% java -cp $HOME/JION.jar HelloWorld take
```

9.2 Asynchronous deployment of a JavaSpaces application

Here we demonstrate deploying a JavaSpaces application asynchronously using *Threads*. Note that the class *Seat* is defined in Section 3.2.2.

```
1.  public class NonBlocking {
2.  public static void main(String [] args){
```

```
3.    final JavaSpace space= JavaSpaceSingleton.getInstance();
4.    new Thread(){
5.      public void run(){
6.         Seat request_1= null;
7.         try {
8.         request_1=    (Seat)    space.read(new    Seat("Paris",    "Roma",    null),    null,
    Lease.FOREVER);
9.         } catch (Exception e){
10.            e.printStackTrace();}}}.start();
11.   new Thread(){
12.     public void run(){
13.        Seat request_2= null;
14.        try {
15.        request_2= (Seat) space.read(new Seat("Paris", null, new Date(2013,9,22)), null,
    Lease.FOREVER);
16.        } catch (Exception e){
17.            e.printStackTrace();}}}.start();}}
```

The previous code can be made more readable by changing the lines, from 4 to 17, by the next two lines only, which involve using operations based on Future object.

```
   Future<Seat> request_1 = space.readf (new Seat ("Paris", "Roma", null), null, Lease.FOREVER);
   Future<Seat> request_2 = space.readf (new Seat ("Paris", null, new Date(2013,9,22)), null,
Lease.FOREVER);
```

As a result, the code will be more *readable* and easier to debug and maintain.