Programming Urban-area Applications

Dries Harnie
dharnie@vub.ac.be

Elisa Gonzalez Boix
egonzale@vub.ac.be

Theo D'Hondt
tjdhondt@vub.ac.be

Wolfgang De Meuter
wdmeuter@vub.ac.be

Software Languages Lab
Vrije Universiteit Brussel

ABSTRACT

The evolution of smartphones has given rise to urban-area applications: applications that communicate in a city by means of the public (moving) infrastructure, e.g. buses and trams. In this setting, applications need to communicate and discover each other using intermediaries that move around the city and transfer data between them. This requires programmers to scatter code that deals with routing messages to the correct place and dealing with network failures all over their programs. Our approach allows the programmer to specify urban-area applications in a high-level manner without the burden of directly encoding communication using intermediaries. We present this as a translation from a high-level object-oriented programming paradigm to a low-level communication mechanism.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems Distributed Applications; D.3.3 [**Software**]: Language Constructs and Features

Keywords

Tuple spaces, ambient-oriented programming

1. INTRODUCTION

There is no denying that smartphones have taken off in recent years, with significant improvements to both their computational and communication capacities. This has given rise to a huge ecosystem of mobile applications, varying from entertainment to information retrieval, on-line shopping and others. These applications typically adapt the functionality present in existing desktop or web applications to the user interface and capabilities of mobile devices. Some mobile applications are explicitly targeted towards people in a city: e.g. applications that alert the user to road works in the city or tourist guide applications. These applications are sometimes made available outside of traditional channels, for example the city of London offers tourist information applications in tourism offices spread throughout the city.

Almost all mobile applications assume that internet connectivity is always available. This is not always the case though, e.g. a user might be in an area where his device does not get a signal. Additionally, a user might not want to use mobile data connections as they can be very expensive, especially when traveling abroad. In practice, smartphone owners travel between "islands of connectivity", like their home, their office and Wi-Fi access points provided by local businesses.

When the user is not on an island of connectivity the mobile applications on their smartphone are incapable of communicating, often for hours at a time. When the device regains connectivity, programmers need to coordinate and reconcile any off-line operations with other devices. In this paper we propose to use the public transportation infrastructure to carry user communication, extending the existing islands of connectivity to the areas frequented by the public transportation. We call this kind of applications that are distributed on and focused on a city *urban-area applications*. Rather than communicating through servers on the internet, urban-area applications communicate using this public transportation backbone.

Unfortunately, current mobile middleware is not directly applicable for programming urban-area applications, as they work under different assumptions. First of all, applications primarily communicate with other applications indirectly. They will send a message using the public transportation backbone and *eventually* get an answer back. With round-trip communication times of several minutes, programmers must write applications that communicate asynchronously. Because communication happens via third parties, some messages will arrive in duplicate or not at all; developers must take care their applications still work as intended. Finally, there is no central registry of applications; instead, applications must discover other running applications in order to interact with them.

In this paper, we explore a means of programming urban-area applications that extends the popular object-oriented programming paradigm, while hiding the communication and distribution details from the programmer. This makes it possible to formulate urban-area applications using high-level communication abstractions. Our approach then translates these abstractions into low-level communication that uses store-and-forward communication strategies.

The rest of this paper is structured as follows: first we de-

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

scribe urban-area applications and the environment in which they work in section 2. Then, we discuss existing approaches to indirect communication and programming abstractions for unstable network connections in section 3. In section 4 we define criteria for a suitable low-level communication mechanism, along with a high-level language for programming urban-area applications and a translation between the two. Finally, in section 5 we present a first evaluation of our translation and we discuss a number of parameters that can be tweaked in future deployments of urban-area applications.

2. URBAN-AREA APPLICATIONS

In this section we will discuss the characteristics of urban-area applications. As mentioned in the introduction, urban-area applications are deployed in an urban setting and communicate using the available (mobile) infrastructure in the city. In this paper, we use the public transportation vehicles as extensions of the current "islands of connectivity". These vehicles, equipped with wireless communication hardware, transport communication between islands.

As a concrete example of an urban-area application deployed on an urban-area network, we present an *urban bulletin board*: an open city-wide discussion forum inspired by the popular website Craigslist. Instead of transmitting messages to a centralized server over the internet, the urban bulletin board application exchanges messages with a public transportation vehicle whenever the user is on or near it. This vehicle will then propagate it to other users. Somebody who e.g. lost her umbrella on the bus can post a message to this urban bulletin board. This message will be picked up by other public transportation vehicles and carried around as they move through town. People who are physically remote can in turn pick up this message and send out replies to the message, which will be transported back in a similar way. Implementing this as an urban-area application allows people to engage on a highly local and topical basis.

Because of the difference in network topologies, urban-area applications are more related to applications in mobile ad-hoc networks [15] and nomadic networks. In these kinds of networks, users move about constantly and occasionally connect wirelessly to either peers or (mobile) infrastructure nodes. Urban-area networks additionally allow users to forward communication on to islands of connectivity. This difference in connectivity and network topology also makes programming urban-area applications different from typical distributed applications:

Indirect communication by default
In an urban-area setting, connections to other parties will be extremely volatile. Additionally, urban-area applications will almost always communicate *indirectly* using the public transport infrastructure. Because messages need to be physically carried around by public transportation vehicles, communication between two parties can take hours.

Scoping by default
Urban-area applications are implicitly scoped to the area they are deployed. This in contrast to traditional internet applications, where messages must be explicitly scoped and users must define bounds on their interaction. Additionally, the user's relative location or connectivity in the urban area can be used to further delimit scope. For example, messages posted close to the user's position can be prioritized over other messages, changing as the user moves about.

Indirect peer-to-peer communication
Instead of using a global infrastructure like internet-hosted applications, urban-area applications will communicate locally through the public transportation infrastructure. This blurs the line between client and server applications: applications can publish data directly onto the infrastructure and similarly can *directly* read data published by others.

Large periods of disconnection
Users will often be out of range of the public transportation infrastructure or turn off their wireless connection to save power. This means that urban-area applications cannot depend on permanent connectivity: they should buffer communication locally until it can be forwarded to the public transport infrastructure. These large periods of disconnection also make it more probable that applications are working from data that is out of date.

Dealing with these properties puts an extra burden on programmers. Therefore, the goal of this paper is to hide them from programmers by means of high-level abstractions for writing urban-area applications.

3. RELATED WORK

Now that we have given an overview of what urban-area applications entail, we will discuss how existing approaches deal with these issues. In order to deal with the unstable network connection, programmers often turn to middleware suited to mobile applications. We can divide these in four categories.

First, there is *delay-tolerant networking* [3]: a low-level approach to communication for urban-area applications. This technology is typically deployed when users are physically separated and cannot communicate directly, i.e. because there is no infrastructure. For this purpose, there are *message ferries* who can physically carry messages around between users. These message ferries often do not reach their destination in one go, but instead hand over the message to another ferry which can get the message closer to the destination. There are variants where the next hop is determined by social connectedness [9] or the messages themselves [16]. Programming applications using delay-tolerant networking is done using a very low-level API that only supports direct connections. Secondly, for delay-tolerant networking to work, the programmer has to know the destination in advance. This is not possible in urban-area applications, as other running applications need to be discovered dynamically. Additionally, users of urban-area applications are highly mobile, which nullifies most of the routing optimizations used in delay-tolerant networking.

A second, already more high-level type of approach is *publish/subscribe middleware* [2]. In such systems, applications register an explicit interest in certain types of events with the middleware, whenever another application publishes such an event, the subscribers are notified asynchronously. Matching subscribers to publishers is done through *event brokers*, who notify publishers of new subscriptions and forward events to subscribers. The publish/subscribe paradigm was originally limited to static event brokers in local-area networks (LANs), but soon evolved to a more hierarchical and dynamic architecture where event brokers can be added and removed at will [2]. An important evolution for use in mobile contexts can be found in STEAM [8], where events are only valid within a certain proximity of a publisher.

Unfortunately, publish/subscribe middlewares are not suited

to one-to-one communication: programmers need to explicitly encode one-to-one communication themselves. Furthermore, the publish/subscribe hierarchy needs to be recomputed every time the network topology changes, negating the advantages of a hierarchical architecture. Finally, publish/-subscribe middlewares do not support the atomic removal of events, making it hard to clean up stale events.

A third category is *object-oriented middleware*, like Java RMI and .NET Remoting. They try to add support for distribution to the well-known object-oriented programming paradigm. However, they are not resistant to disconnections and need to know the address of the other party in advance. Jini [14] allows programmers to discover services at runtime and interact with them through proxies, allowing for a more flexible architecture. Finally, M2MI [6] enables group communication in Java by providing handles to objects that implement a given Java interface on the local network. It does not provide any delivery guarantees however, the programmer must implement this manually. None of these object-oriented middlewares support rapidly changing network topologies or the indirect communication in urban-area applications.

A final set of approaches are tuple spaces [4]. Tuple spaces contain small units of data called tuples. Applications coordinate by adding tuples to the tuple space and waiting for certain tuples to appear. In the original model, a single shared tuple space was used as a coordination mechanism for distributed computing between devices. Extensions for mobile use [11] adhere to a *federated tuple space model*: when a group of devices establish a connection, their respective tuple spaces are combined. The devices can then see other devices' tuples and e.g. put tuples in other tuple spaces. When the devices later disengage, the federated tuple space is deactivated and each device is left with its own tuple space. Variants of this model replicate tuples to improve data availability [10]. TOTA (*Tuples on the Air* [7]) additionally provides context by propagating tuples across the network according to rules embedded in the tuples.

Unfortunately, it is currently not easy to program urban-area applications using tuple spaces. First of all, the various stages of interaction that represent an interaction between running applications will result in different tuples being created. The operations in the original tuple space model are blocking and thus require one thread per communication partner. Newer tuple space implementations use event handlers for each kind of tuple, but this *inversion of control* [5] scatters the control flow of the program across several event handlers. Secondly, there is no uniform mechanism to *scope* tuples in a tuple space. There are systems where multiple tuple spaces can coexist and access to them can be controlled, but these are static and do not use semantic information. Additionally, there is usually no support for limiting the lifetime of tuples. There are exceptions, like L2imbo [1] that allow developers to specify timeouts on tuples. Scoping is essential in urban-area applications to prevent urban-area networks from collapsing under large amounts of abandoned-but-not-removed tuples. Finally, current replicated tuple spaces do not provide support for deletion of tuples in a tuple space as it can lead to inconsistency. However, programmers writing urban-area applications will sometimes need to delete tuples to indicate the end of conversations.

Summary

In summary, the existing approaches all lack some kind of functionality to properly support urban-area applications. Publish/subscribe middleware allows message to percolate throughout the urban area regardless of connectivity, but is not well suited to one-to-one communication. Object-oriented middleware offers programmers a familiar way of programming, but is ill-suited to deal with the ever changing network topology. Delay-tolerant networking can be used to deliver messages between peers who are physically separate, but is very low-level and thus hard to program. Finally, tuple spaces are also hard to program but are flexible enough to represent the communication patterns programmers envision.

4. URBAN-AREA PROGRAMMING MODEL

In this section we will describe the programming model we propose for urban-area applications. This model consists of three parts: first, each device in an urban-area setting is equipped with a low-level communication mechanism that is responsible for distributing messages across the city.

The second part of our urban-area programming model is a high-level programming paradigm that extends the object-oriented programming paradigm with abstractions for dealing with the issues presented by urban-area networks.

Finally, we will present a mapping between the high-level programming model and the low-level communication mechanism. This mapping will translate the direct communication between peers in the high-level programming model to low-level communication using intermediaries.

4.1 Urban-area tuple space

In this paper we propose tuple spaces as the low-level communication mechanism, because they are flexible enough to express all communication patterns. In an urban-area network, each device defines a tuple space to carry around not only its own, but also a number of "foreign" tuples belonging to other devices.

In the rest of this section we will define and motivate a number of features we envision for the tuple space on these devices. Currently, no tuple space supports *all* of these features. We have added some of the missing features below to an existing tuple space called TOTAM [12].

4.1.1 Replication-based model

When two tuple devices connect to each other, their tuple spaces will exchange their own tuples along with foreign tuples. Each tuple is marked with a unique ID. Prior work in the literature has shown that a replication-based model increases data availability in a highly disconnected environment. In this paper, we use replication to support indirect communication, as it enables intermediary nodes to carry information to and from their eventual destination.

4.1.2 Leasing for tuples

As long as a tuple is in circulation, it takes up crucial storage space and transmission time. Some kinds of tuples are inherently transient, for example tuples that advertise social events on a particular date or store promotions limited in time. Other tuples might be "orphaned" because their owner is no longer in the city. In order to make space and bandwidth available for newer tuples, our urban-area tuple

space requires leasing for tuples. After a certain time period has passed (a day, in our implementation), the leases are revoked and the tuples are removed from the tuple space.

4.1.3 Distributed removal of tuples

A side effect of replicating tuples across devices is that deleting tuples becomes harder: a locally deleted tuple might be reinstated by a subsequent communication with another device. Currently tuple spaces cannot distinguish between a tuple that was deleted and a tuple that has not been received yet. To make this distinction, the replicated tuple space remembers and propagates deletions of tuples by introducing *anti-tuples*. These spread like normal tuples but annihilate their corresponding tuple when they enter a tuple space.

We have extended TOTAM to allow anyone (not just the owner of the tuple) to initiate the deletion of tuples. Anti-tuples expire not long after the regular tuples, ensuring that anti-tuples do not linger.

4.1.4 Scoped tuples

In an urban-area environment, battery and network usage are important considerations. Tuples should therefore be targeted towards people more likely to propagate them to the correct destination. Highly context-sensitive data could even be limited to certain physical locations or public transport lines. For example, a message from a commuter who lost his umbrella on the bus should not spread beyond that bus line. Implementation of this feature is still ongoing.

4.1.5 Event-driven operation

As round-trip times in an urban-area application can be long, programs interacting with an urban-area tuple space cannot use synchronous operations. While it is possible to perform multiple blocking operations using threads, this is not feasible given the multitude of communication partners. Given the mobile setting we propose an event-driven operation instead, where the programmer registers callbacks that are invoked when e.g. a certain tuple is put in the tuple space. TOTAM is already event-driven. Furthermore, it allows event handlers to register new event handlers, preserving control and data flow.

4.2 Programming urban-area applications

As mentioned earlier, urban-area networks are similar to mobile ad-hoc networks. Just like an urban-area network, connections in such a network are considered extremely volatile, which requires programmers to litter their code with exception handlers. To ease development in mobile ad-hoc networks, the ambient-oriented programming paradigm [13] was proposed. It extends the object-oriented paradigm to tackle the volatile network connections inherent to mobile ad-hoc networks. In order to support urban-area applications, we propose to extend the ambient-oriented programming paradigm with support for indirect communication.

In the ambient-oriented paradigm, communication between two applications happens exclusively through *far references*: remote object references that only allow asynchronous communication and mask disconnections by default. When a far reference becomes disconnected — the device hosting the object is no longer reachable — all further messages sent to the far reference are buffered locally. When the remote device later reconnects, the messages are sent. This property makes ambient-orient programming interesting for urban-area applications. However, ambient-oriented programming relies exclusively on direct connections between devices which is almost never the case in urban-area applications.

We have implemented our extensions to the ambient-oriented programming paradigm in the AmbientTalk language [13], a distributed programming language that adheres to this paradigm. In order to illustrate how to program urban-area applications, we will use AmbientTalk to present the urban bulletin board example introduced in section 2.

In AmbientTalk, applications need to discover objects exported by other applications in order to communicate. Each application can export its services using descriptive *type tags*. When it starts, a urban-area application starts discovering any objects that are exported:

```
deftype BBoard; // urban bulletin board
whenever: BBoard discovered: { |aBoard|
  def topicsFut := aBoard<-queryFor(interestingTopics);
  when: topicsFut becomes: { |topics|
    GUI.show(topics);
  }}
```

Whenever an urban bulletin board service is discovered with the `BBoard` type tag, the `whenever:discovered:` callback is invoked with a single argument: a *far reference* to the discovered object. If this far reference disconnects and later reconnects, the discovery handler is triggered again.

Here the ← operator denotes an asynchronous message send. The ambient-oriented programming paradigm enforces that subsequent messages sent to the same far reference are processed in the same order. Note that the programmer does not have to specify explicitly how the message should be sent. The `queryFor` message returns a *future*: a placeholder for the return value. When the remote `aBoard` finishes its `queryFor` computation, the future is resolved with the return value. This in turn triggers the attached `when:becomes:` handler, in this case showing the returned topics.

Programmers can also attach arbitrary annotations to message sends:

```
aBoard<-post("lost umbrella")@For(weeks(1))
```

This posts a message to `aBoard` that has a suggested lifetime of one week. We also support a `@Near` annotation that scopes a message to a given location.

An application exports its own services using the `export:as:` construct, which makes the given object discoverable using the given type tag:

```
def myBoard := object: { def queryFor(topics) { ... }};
export: myBoard as: BBoard;
```

Messages sent to this exported object (as above) are translated into method invocations on this object and any return values are wrapped in a future.

This style of programming frees the programmer from the specifics of the networking technology used. Our proposed translation replaces the ← operation on far references by putting tuples in the urban-area tuple space, where they can be transported to their destination by public transportation vehicles. Similarly, our translation enables service discovery by exporting tuples for other applications to discover.

4.3 Translating ambient-oriented programs to tuple spaces

In this section we will explain how the ambient-oriented communication primitives we described above can be represented using operations on tuples. These tuples will reside

	Ambient-oriented code	Operations on the urban-area tuple space
EXPORT	`export: OBJ as: TAG`	*objectID* ← *generate new object ID* *mapping*$_{objectID}$ ← *OBJ* *ref* ← ⟨deviceID, objectID⟩ out⟨EXPORT, *TAG*, *ref*⟩

DISCOVERY	`whenever: TAG discovered: CLOSURE`	whenever: ⟨EXPORT, *TAG*, ?*ref*⟩ read: {
		call CLOSURE with *ref*
		}

SEND	REF←AMESSAGE(arg$_1$, arg$_2$, ...)	if *first time sending to REF* then *seqno*$_{REF}$ ← 0 else *seqno*$_{REF}$ ← *seqno*$_{REF}$ + 1 *sender* ← deviceID out⟨MESSAGE, *sender*, *REF*, *seqno*$_{REF}$, AMESSAGE, toReference⟨*arg*$_1$, *arg*$_2$, ...⟩⟩

RECEIVE	(done automatically)	whenever: ⟨MESSAGE, ?*sender*, ⟨deviceID, ?*objectID*⟩, 0, ?*message*, ?*arguments*⟩ in: {
		localObject ← *mapping*$_{objectID}$
		localObject.*message*(*arguments*)
		// *install a handler with the same sender and objectID*,
		// *but with seqno = seqno + 1*
		}

Table 1: Mapping the basic operations in ambient-oriented programs to urban-area tuple spaces

in an urban-area tuple space as defined above. Our translation can be applied both at compile-time and at run-time. Our translation uses two kinds of tuples for communication, tagged with either EXPORT or MESSAGE:

- ⟨EXPORT, *tag*, *obj*⟩
  This kind of tuples represent an object exported to the environment. The first parameter, *tag*, is a type tag that informally describes the exported object; other applications can use this type tag to discover objects they are interested in. *obj* refers to the exported object.

- ⟨MESSAGE, *sender*, *target*, *seqno*, *message*, *arguments*⟩
  This kind of tuples encode the sending of a *message* with given *arguments* to a far reference. To retain the implicit message ordering in point-to-point connections, we tag each message sent to *target* with the actor that sent it (*sender*) and a sequence number *seqno*.

Table 1 contains two rules that describe how the object discovery mechanism is encoded into tuples, and two rules that describe how message transmission and reception happens:

4.3.1 The EXPORT/DISCOVERY protocol

When an applications exports an object to the environment, it creates an EXPORT tuple that contains a type tag and a tuple that represents a far reference to the object. This tuple consists of a fresh identifier for the object and an identifier for the device it is hosted on.

When a client wants to discover objects in the environment, the procedure is reversed: it attempts to read an EXPORT tuple with the *tag* field filled in. Whenever such a tuple is found, the ?*ref* parameter is filled in by the tuple space and passed to the callback.

4.3.2 The SEND/RECEIVE protocol

Reading tuples from tuple spaces is inherently nondeterministic, there is no ordering imposed on tuples. Our translation must take care to preserve the message ordering imposed by the ambient-oriented paradigm. We tag each message with its originator and a sequence number: the first time an application sends a message to a far reference, it is tagged with sequence number zero. This number is increased for every message sent afterwards. Before a MESSAGE tuple is sent the `toReference` function replaces each object in the argument list with a far reference, similar to when an object is exported.

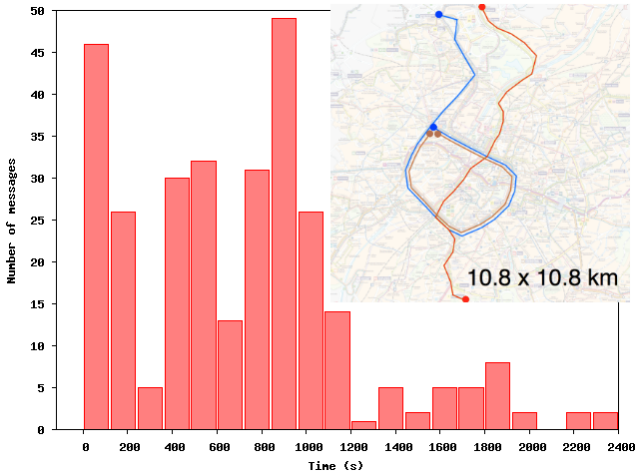

Figure 1: Histogram of delivery times

On the receiver side, each device creates an asynchronous `whenever:in:` handler that is triggered when another device sends its first message to each exported service. When such a tuple is found, it is removed from the tuple space and forwarded to the local object. Afterwards, a new `whenever:in:` handler is installed that explicitly matches the given sender, object ID and sequence number.

5. EVALUATION AND FUTURE WORK

We have tested our approach by simulating an urban-area network using thirty buses on three separate lines. At the endpoints of each line stands a person that discovers other people and then exchanges messages. Communication is carried by buses, moving from stop to stop at a speed of 30 km/h. When buses meet at a stop, they exchange their carried tuples. The result of this simulation can be seen in Figure 1: 90% of all communication is received within 20 minutes with a median delivery time of 12 minutes.

To get an estimate of how real urban-area applications will fare, we need to scale up these experiments to real-world situations. We are currently conducting experiments with real buses in Brussels with the help of the STIB, the local public transport company. We surmise that proper tuning of the individual tuple spaces will be crucial to supporting

effective urban-area network applications. In the rest of this section we will discuss some possible improvements, both for the programming model as the urban-area tuple spaces.

Prioritizing tuples: From a replicating tuple space's point of view, all tuples are equal. However, as programmers we can identify tuples that should be sent first during the short tuple transmission window, to ensure that the recipient can proceed with local computations as far as possible. A simple example of this are the MESSAGE tuples: since e.g. message #5 will not be processed before #4 arrives, our system should transmit these tuples in their sequence order. Similarly, sending anti-tuples first reduces the amount of tuples that are moved around the urban-area network. Finally, the type of application could also have an impact on the priority its tuples get. For example, tuples that are part of administrative urban-area applications could be given priority.

Shared global data structures: Since a city-wide tuple space reintroduces the idea of shared knowledge, this could be exposed to programmers. For example, a map of landmarks and tourist attractions in the city could be presented as a "shared set" which can then be discovered, imported into a local tuple space, and used off-line. Such a data structure would not be modifiable by the general public, but instead curated by a dedicated entity related to the data structure, e.g. the tourism office.

Explicit message queues: As a side effect of our translation, message queues are made explicit as tuples. We are currently investigating the possibilities of allowing programmers to reify this message queue. This could enable programmers to batch multiple messages into one single message. It could also enable "anti-messages" which — like anti-tuples — annihilate messages waiting in the message queue.

6. CONCLUSION

In this paper we presented urban-area applications: applications designed for an urban setting that communicate with each other by relaying messages over shared public infrastructure. This public infrastructure can be static, like bus stops or public internet access points, or dynamic, for example buses rolling around.

This network topology poses a number of problems for programmers who wish to use traditional mobile programming paradigms. First of all, direct connections to other devices are infrequent, so urban-area applications need to use indirect communication. Secondly, there is no central registry of active applications; instead, applications must discover other running applications. Finally, urban-area applications spend long periods of time without connectivity to other devices, so data availability is important.

To avoid addressing these problems directly in code, programmers use middleware. In this paper we identified two types of middleware: high-level extensions to the object-oriented programming paradigm and low-level middleware for propagating data across an unreliable network. The former provides a high-level approach to programming urban-area applications, but is not suited to indirect communication. By contrast, low-level middleware is hard to program but is well suited to indirect communication. Our approach combines the benefits of the two approaches by enabling programmers to write urban-area applications using object-oriented programming, while abstracting from the low-level mechanism used to provide indirect communication.

7. ACKNOWLEDGMENTS

Dries Harnie and Elisa Gonzalez Boix are supported by a grant from the Prospective Research for Brussels program of Innoviris. We would like to thank the STIB for providing us with the necessary resources, and the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] N. Davies, A. Friday, S. Wade, and G. Blair. L2imbo: a distributed systems platform for mobile computing. *Mob. Netw. Appl.*, 3(2):143–156, 1998.

[2] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Survey*, 35(2):114–131, 2003.

[3] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proc. of SIGCOMM 2003*, pages 27–34. ACM, 2003.

[4] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.

[5] P. Haller and M. Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[6] A. Kaminsky and H.-P. Bischof. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *Proc. of OOPSLA 2002*, pages 72–73, 2002.

[7] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *Proc. of PERCOM '04*, page 263, 2004.

[8] R. Meier, V. Cahill, A. Nedos, and S. Clarke. Proximity-based service discovery in mobile ad hoc networks. In *DAIS 05*, pages 115–129. Springer, 2005.

[9] A. Mtibaa, M. May, C. Diot, and M. Ammar. Peoplerank: Social opportunistic forwarding. In *Proc. of IEEE INFOCOM 2010*, pages 1–5.

[10] A. Murphy and G. Picco. Using lime to support replication for availability in mobile ad hoc networks. In *8th International Conference on Coordination Models and Languages (COORDINATION)*, LNCS, pages 194–211. Springer-Verlag, 2006.

[11] A. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proc. of ICDCS01*, pages 524–536, 2001.

[12] C. Scholliers, E. Gonzalez Boix, and W. De Meuter. Totam: Scoped tuples for the ambient. In *Proc. of CAMPUS09*, volume 19, pages 19–34. EASST, 2009.

[13] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12, 2007.

[14] J. Waldo. The Jini Architecture for Network-centric Computing. *Commun. ACM*, 42(7):76–82, 1999.

[15] Z. Zhang. Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: overview and challenges. *IEEE Communications Surveys & Tutorials*, 8(1):24–37, 2006.

[16] W. Zhao and M. Ammar. Message ferrying: proactive routing in highly-partitioned wireless ad hoc networks. In *Proc. of FTDCS 2003*, pages 308–314. IEEE.