Spatio-Temporal Interaction Primitives for Delay Tolerant Systems

Mirco Musolesi
Dept. of Computer Science
University College London
Gower Street, London
WC1E 6BT, United Kingdom
m.musolesi@cs.ucl.ac.uk

Cecilia Mascolo
Dept. of Computer Science
University College London
Gower Street, London
WC1E 6BT, United Kingdom
c.mascolo@cs.ucl.ac.uk

ABSTRACT

Computing and communication devices pervasively surround our daily life and the presence of embedded systems, including tiny sensors, is increasing exponentially. However, the software and communication mechanisms used to network these devices are still the ones that we have been devised 30 years ago for standard computer systems. Pervasive systems are by orders of magnitude more dynamic than traditional systems and have often less resources (energy, memory, bandwidth). Different communication and coordination patterns are emerging for these environments, ranging from those related to *delay tolerant systems* [5], where communication happens asynchronously between devices, to location based communication, where hosts receive information only when they are in a specific location. This calls for a radical change in the communication mechanisms applied. In these environments, several concepts, not captured by the semantics of the programming interfaces of traditional systems, such as location or temporal validity of the disseminated and replicated information, are fundamental. In this paper we propose a novel set of communication primitives for this kind of systems that would allow software engineers to better exploit the potential of these environments. These primitives combine spatial and temporal concerns to cope with the dynamics and mobility of pervasive systems. This paper describes a formalisation of the primitives in Mobile UNITY and a general middleware framework to support them.

1. INTRODUCTION

The number of mobile phones in Europe is higher than the number of personal computers. Computing and communication devices pervasively surround our daily life and the presence of embedded systems, including tiny sensors, is increasing exponentially. Thanks to the progress in digital technologies, mobile computers have processing power and memory comparable to advanced and extremely expensive workstations of twenty years ago. In general, computer systems are *heterogeneous*, since they are often composed of different types of fixed and mobile devices and (intermittently) *connected*, as bandwidth may be fluctuating or absent, depending on location.

New possible scenarios are enabled by the availability of such technologies and their internetworking. One of the more promising is that of *delay tolerant networks* [5]. Delay Tolerant Networks are characterised by long delay paths and frequent (in some cases also unpredictable) disconnections and network partitioning. Possible examples may be intermittently connected mobile ad hoc networks [12], interplanetary and satellite communications [7] and mobile systems to provide transitive connectivity to isolated villages in rural areas [21, 4, 16]. In the Data Mules Project [20], for instance, the data of the sensor nodes are collected by a device (the "mule") that travels among them. Another existing solution is DakNet [16], which aims to provide intermittent connectivity to the global Internet to rural areas of India and Cambodia. People in villages access services such as email in e-kiosks: messages are collected and transported to (and from) an Internet gateway in the nearest town by buses. These are equipped with wireless technologies so that they can download and upload messages from and to the e-kiosks and the Internet gateways.

Another example is the design of software systems for space exploration [15, 7]: delays in transmitting information due to the very long distances are common in this setting. Moreover, satellites, space probes and rovers may not be directly reachable from the Earth due to their position since they may be on the non visible side of a planet or covered by other celestial bodies.

The concept of delay tolerant networks represents a very general abstraction that includes existing systems relying on fixed networks, mobile ad hoc networks and hybrid networks composed of fixed and mobile nodes. For this reason, it provides a very general scenario for the design of primitives for a very large number of systems and deployment settings.

On the other hand, communication mechanisms and algorithms currently used to network modern devices are still the ones that we have devised 30 years ago for traditional computer systems. These primitives may be still effective for a vast class of application scenarios, especially for business automation and scientific computing, however, they do not exploit the full potential of the pervasive computing scenarios, such as the one just described. In fact, classic program-

ming paradigms and, in particular, programming interfaces defined for traditional middleware systems do not capture the natural communication paradigms typical of intermittently and location based networks; some examples include the ability to send a message to a number of hosts in a location or send it to hosts currently not connected to the same portion of the network (also with the possibility of specifying their location).

The contribution of this paper is the design of novel primitives for communication in delay tolerant mobile systems that fully exploits the potential of the environments considered, by combining spatial aspects with temporal ones. These primitives allow, for example, to specify that a message has to be sent (in a synchronous or asynchronous ways) to a recipient, only when this is in a specific location; or to all the devices that are in a specific location (i.e., *geo-casting*). They also allow to express the fact that the delivery has to happen only when the sender is located in a specific point of the geographical space. Simmetrically, similar constraints and requirements in terms of space and time can be also specified by the recipients, for instance, by saying that the recipient will only want to receive a message from a certain sender, when reaching a specific location.

The primitives are embedded into a middleware framework that supports the dissemination and the persistence of the information. These aspects are key in delay tolerant networking. In fact, if a message cannot be delivered immediately as the recipient is not in the same connected portion of the network, the message might have to be stored in *intermediate buffers*. For this purpose, intelligent and reliable replication and forwarding strategies must be devised. Clearly, the reliability of the system is strictly dependent on the use of the available resources, especially in terms of memory, bandwidth and computational power. These issues are extremely relevant in the case of mobile devices. Therefore, there is a trade-off between the reliability of the system and the resource consumption. The primitives described in this paper allow the specification of the required reliability (in terms of probabilistic delivery). The middleware delivers the message by exploiting different dissemination and message forwarding strategies. These will be tuned to ensure the expected reliability, given the knowledge of the network topology that can be assumed. It is outside the scope of the paper to describe the different dissemination strategies that can be used to implement specific delivery probabilities. However, we underline that the middleware framework allows for the integration of different dissemination algorithms. The interested reader can find the details of possible dissemination models in[12, 13].

The research area of delay tolerant networks has recently received a lot of attention [5]. Existing work in the area has focussed more on the pure networking aspects rather than on the analysis and the design of programming paradigm and interfaces. To our knowledge, there are no works on these issues. With respect to the general design of spatio-temporal systems, a novel analysis has been recently presented in [18]. However, the authors propose a programming interface that is not sufficiently expressive to capture and specify the spatial and temporal constraints that characterise the interaction in delay tolerant systems, such as the reciprocal positions of senders and receivers. Moreover, the authors presents only an analysis of the programming interface without providing the design of a general software architecture that implements this API.

We provide a formalisation of the model using Mobile UNITY [19]. There are many existing formal notation for mobile computing, such as π-calculus [11], KLAIM [3], Mobile Ambients [1], to name a few. These focus on different aspects of mobile computing and provides different types of basic formal abstractions. For example, the Mobile Ambients calculus and logic are extremely useful in the definition of mobile environments with different name scopes, since they provide simple formalisms to denote the validity of namespaces in nested locations. For our formalization we have chosen Mobile UNITY as it provides powerful abstractions based on a concise set of formal structures to represent location and data sharing, that are the key concepts at the basis of our model. In Mobile UNITY location is encapsulated in a variable associated to each software component. We extended this representation of space by adding the idea of *communication region* to represent the fact that a host is not only in a particular geographical position (i.e., location) but also in a particular connected portion of the network.

The contribution of this paper can be summarised as follows:

- we propose a set of primitives that can be used to specify various spatio-temporal aspects of communication in settings where these issues are fundamental, such as in delay tolerant networked systems;
- we discuss an abstract middleware architecture that gives semantics to this programming interface;
- we propose a formalisation of the architecture by extending the syntax and the semantics of Mobile UNITY by adding the definition of communication region;
- we show how different third-part communication and dissemination protocols can be integrated in the middleware framework to support the semantics of the programming interface.

The paper is organized as follows. In Section 2 we analyse the challenges and the requirements of the design of communication primitives in delay tolerant mobile networks. In Section 3 we describe the communication primitives and the architecture of the middleware offering them as an API. Section 4 specifies the semantics of the primitives using Mobile UNITY. The proposed model is then discussed and related work is presented in Section 5. Section 6 concludes the paper, outlining our current research directions.

2. COMMUNICATION IN DELAY TOLERANT MOBILE NETWORKS: A SCENARIO

In order to define the design requirements of the primitives for delay tolerant mobile systems, we start from a realistic example, considering the case of providing communication to a village in a poor area, distant from the nearest main town and therefore connected to the global Internet by means of a bus that acts as message carrier (as in the DakNet project [16]). We suppose that the village is composed of three main locations covered by a local network that is disconnected from the Internet[1]: the residential area, the local administrative offices and the rural area near the village. We also assume that all nodes in the village are

[1]As discussed in [16], considering the user requirements in these scenarios, the connectivity to the Internet by means of a satellite link or a wired line is not convenient in most of the cases.

connected by a LAN attached to a WiMAX network that covers the rural area outside the village, providing connectivity to all the farmers. In a sense, the village can be seen as a connectivity island. When the bus is in proximity of the main town, it is connected to the Internet by a wireless gateway, whereas, when it is in the village, it is connected in a similar way to the local network.

In this setting, the ability to send messages to a location, to exploit the asynchronous communication through the use of the bus as a "store and forward" engine for messages, or to deliver messages to all hosts in a location, are examples of communication patterns difficult to express by using the synchronous and asynchronous primitives exploited by most middleware for traditional systems. The argument of this paper is that middleware should provide support for, for instance, sending a message from a host in the global Internet to a recipient in the isolated (or better, *intermittently connected*) village and viceversa. A host in the global Internet should also be able to send a message to all the hosts that are in a specific geographical region. Messages may have an expiration time, in order to express the validity of the information. In fact, a message containing weather forecast about a certain day, for instance, would not be so useful if received on the same day (or after).

Let us envisage some more specific scenarios: for example, the central meteorological office in the near big city should be able to send a weather alert to all farmers in the rural area. Let us further assume that the computer system of the offices is managed by technicians remotely from the near city: a system administrator should be able to specify his/her interest in receiving only requests from clerks of the offices in the villages that he/she manages and supervises. What is needed in this case is a **send**() primitive that allows developers to specify that a message has to be delivered to a certain recipient, in a certain area of the village, or to all the recipients that are in that area, or to a certain recipient only if he/she is in that particular area and so on. A symmetric semantics should be made available to indicate the recipients and/or their locations in the **receive**() primitive.

As far as the message reliability is concerned, a critical weather alert should be sent with high reliability, whereas an ordinary hourly update of the wheather forecast may be characterised by a lower one. Let us consider again our scenario. Let us suppose that a sensor network is deployed in the fields in the rural area outside the village, in order to measure environmental indicators such as humidity, pollution and so on. We suppose that the data are collected by means of motorbikes and helicopters. We would also like to be able to express the fact that the collection of the data will be performed only in certain locations (i.e., receival should only happen in these locations). In other words, we would like to be able to specify not only the locations of the senders and the receivers of the message, but also where the **send**() and the **receive**() should be fired.

To summarise, there is a need to provide primitives that enable and combine:

- *synchronous/asynchronous* communication (i.e., the sender and/or the receiver are or are not blocked awaiting for the successful execution of the primitives);
- *delay tolerant/non delay tolerant* communication (i.e., it is admissible or not that messages will be delivered with a delay that may not be negligible);
- *spatial/non spatial* communication (i.e., there is the support for geo-casting and location-awareness or not).

In the following sections we will present a set of primitives that allows developers to specify these dimensions and we will discuss an abstract middleware architecture that supports them.

3. SPATIO-TEMPORAL COMMUNICATION PRIMITIVES

We now present a detailed definition of the communication primitives. The primitives incorporate spatio-temporal concepts which, for instance, allow the description of the operations sketched in Section 2. The novelty of these primitives resides in their expressiveness and their flexibility, since, by using them, the software engineer can specify a wide range of requirements and constraints for the communication process. More specifically, we define a new set of primitives for sending and receiving messages. In order to meet the requirements defined in Section 2, the **send**() primitive has the following signature:

send (*m*, *recipient*, *recipientLocation*, *senderLocation*, *tExp*, *tBlock*, *reliability*)

By using this primitive, developers are not only able to define the recipients of the message *m* (in *recipient*), but also to express spatial concepts, such as the location where the message has to be delivered to (in *recipientLocation*) and the location where the effective sending has to be performed (in *senderLocation*) (i.e., the sending is performed only when the host is in the location expressed in *senderLocation*).

Moreover, the *recipient* field can assume two values, the identifier of the receiver, a list of receivers or $*$, to indicate that the message is sent to every host. Similarly, developers can specify one recipient location or a generic location (using the same symbol $*$)[2].

In order to clarify these concepts, let us consider some examples:

- **send** (*m*, *, *,...) has to be used to send a message to all the hosts, independently of their position;
- **send** (*m*, 32, *,...) indicates that the messages is to be sent to host 32; host 32 can receive it independently of its position;
- **send** (*m*, 32, *ruralArea*,...) indicates that the message is to be sent to host 32; host 32 can receive the message only when it is in location *ruralArea*;
- **send** (*m*, *, *ruralArea*,...) indicates that the message is to be sent to all hosts in location *ruralArea*.

By using the *tExp* field, developers are able to set the expiration time of the message, whereas *tBlock* defines the interval of time during which the application is blocked waiting for the correct delivery of the message. The expiration time indicates the validity of the message. The corresponding acknowledgment message will have the same expiration time.

[2]It is clearly possible to extend the syntax and the semantics of these primitives in order to specify a list of senders and receivers in different locations, by using a list of tuples with the format *(hostId,LocationId)*.

Through these timers it is possible to specify synchronicity and asynchronicity on top of the possible location based operations specified above; for example, **send** (*m*, 32, *,.., 20, 20,...) indicates that the message is to be sent to host 32, independently of its position, but that the message expiration time is set to 20 time units and that the application is blocked for 20 time units while waiting for this to be delivered and acknowledged. On the other hand, **send** (*m*, 32, *,.., 20, 0,...) will be used for the asynchronous delivery of a message with an expiration time equal to 20 time units.

Finally, the desired reliability of the delivery process can be specified as a percentage in the *reliability* field. Since, in many cases, the evolution of the deployment scenarios cannot be predicted with accuracy (i.e., it is not deterministic), the reliability value specified in the sending primitives is evaluated in probabilistic terms. The middleware driven delivery process associated to the primitives depends on the *dissemination strategy* used to replicate or forward the messages to the other hosts. If messages cannot be delivered immediately (i.e., when the recipients are not in the same connected portion of the network), they are stored in intermediate buffers that we call *Message Buffers*. The dissemination strategy is composed by set of algorithms and protocols used to transfer and disseminate the information in the system in order to deliver the information as close as possible to the recipient(s) and/or to the the locations specified in the **send**() primitive.

The description of the mechanisms to be used to guarantee the specified reliability is outside the scope of this paper and topic of other papers [13, 12]. However, let us briefly consider the case of a dissemination process based on a pure epidemic protocol [22]. If the value specified in the primitives is 100, the message might be replicated on every host, whereas if it is lower, it might be copied only on a subset of nodes. Given their semantics, epidemic protocols can be used when the two hosts are not in the same connected portion of the networks as they permit the storage of information that can be kept for a while and then distributed later. In our example, messages may be replicated on all the vehicles that acts as information between the Internet gateway and the village in order to increase the reliability of the delivery process. On the other hand, if two hosts are in the same connected portion of the network (that, in the remainder of the paper, we will call *communication regions*), in presence of a standard routing mechanisms, the message will be directly transferred to the buffer of the recipient. In our example, this is the case of messages that are exchanged by two hosts that are inside the village.

We define, in a symmetric way, the **receive**() primitive as follows:

m=**receive** (*sender*, *senderLocation*, *receiverLocation*, *tBlock*)

Similarly to the **send()** primitive, developers can specify the sender (in *sender*), its location (in *senderLocation*), the location where the receiving needs to happen (in *receiverLocation*) and the time interval during which the receiving application is blocked waiting for a message (in *tBlock*).

We will now formalise the architecture of the middleware to support these primitives, starting from the definition of the space where the computation takes place, also discussing the concepts of physical location and connectivity.

4. SPECIFICATION OF THE ARCHITECTURE

In this section, we will present the semantics of the primitives and we will specify the middleware framework which supports this programming interface, by using the Mobile UNITY [19] notation. We now define the concepts of location and connectivity then the general architecture of our middleware; finally, we provide a brief description of Mobile UNITY and of how we use it for the definition of our semantics.

4.1 Physical Location and Connectivity

The computation happens in a physical space that we indicate with Λ. Λ is subdivided into l locations λ_1, λ_2, ..., λ_l. We make the assumption that locations are not overlapping. This simplification has a sensible mapping on the physical world. In fact, real location information can be provided by a communication infrastructure; in cellular networks, for instance, the borders of each cell is not well-defined, since, in the border regions, a mobile phone receives the signals from multiple base stations. However, mobile phones register to just one location, choosing the base station that can offer the best signal strength and quality. Therefore, even if, in theory, the locations are overlapping physically, a host belongs to only one location from a logic point of view. Another example may be the Active Badge System [23], that provides the position of individuals and computers by means of networked sensors spread across buildings.

More formally, it is possible to write:

$$\Lambda = \lambda_1 \cup \lambda_2 \cup ... \cup \lambda_l \qquad \lambda_1 \cap \lambda_2 \cap ... \cap \lambda_l = \varnothing$$

For example, as shown in Figure 1, the village area is composed of three locations, $\lambda_{residential}$, λ_{office}, λ_{rural}, indicating respectively the residential, the administrative and the rural areas. A host, however, is not only contained in a geographical space, it also belongs to a logical computational space that we call *communication region*. We have introduced this concept in Section 2, and we now provide a precise definition. Communication regions are connected portions of the entire network at any specific time. More precisely, we refer to a communication region as the set of the hosts H_1, H_2, ..., H_N such that for every pair of hosts H_i,H_j with $i \neq j$ and $i, j = 1, ..., N$ it is possible to transfer the content of the Message Buffers of H_i to H_j as there is a communication path among the hosts. There are several possible examples of communication regions. An example of communication region in mobile ad hoc systems may be the set of hosts that are in the same reachability area, defined by routing mechanisms that enables multi-hop communication. Please note that if a routing protocol (such as DSDV, DSR, etc) is not present, the communication region of a host is represented by the hosts that are in the transmission range. In fixed networks, a LAN is an example of communication region. A non partitioned TCP-IP inter-networked system is another example of communication region. Instead, a network with two partitions is composed of two communication regions.

As for physical locations, we assume that a host belongs to a single communication region at a certain instant t. In general, communication regions are dynamic. In other words, we have different intermittently connected portions of the delay tolerant network composed of a continuously changing set of nodes at any point in time. In order to un-

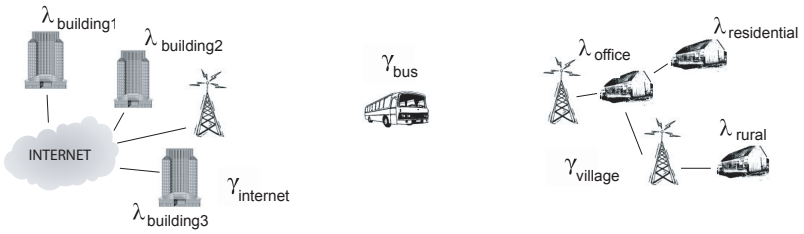

Figure 1: Example of delay tolerant networked system.

derstand this concept better, let us consider our example, once again. When the bus is connected to the Internet, we have two communication regions, the communication region $\gamma_{Internet}$, composed of all the hosts, the Internet and the bus itself, and the communication region $\gamma_{village}$, composed of all the hosts that form the isolated local network of the village. When the bus moves towards the village and it is disconnected both from the Internet and from the Local Area Network of the village, we have three communication regions: $\gamma_{Internet}$(composed of all the computers in the Internet), γ_{bus} (i.e., the bus itself), $\gamma_{village}$ (with the same composition as above). Finally, when the bus will arrive in the village and gets connected to the local network, we have two communication regions, $\gamma_{Internet}$ and $\gamma_{village}$, with the bus inside the latter.

Moreover, communication regions are disjoint and not overlapping. The union of all the g communication regions is clearly the global delay tolerant networked system that we indicate with Γ. More formally, it is possible to write:

$$\Gamma = \gamma_1 \cup \gamma_2 \cup ... \gamma_g \qquad \gamma_1 \cap \gamma_2 \cap ... \cap \gamma_g = \varnothing$$

In order to model the delivery of messages inside a communication region (i.e., without an intermediate storage in Message Buffers), we introduce the concept of *virtual channels*. A virtual channel exists between every pair of hosts that are in the same communication region. A virtual channel can be seen as an abstract representation of the idea of end-to-end communication inside a communication region.

4.2 Definition of the Entities of the System

Our system is composed of M hosts. Every host has a unique identifier. A host can be fixed (such as a gateway, a server, a base station, an infostation and so on) or mobile (such as a mobile phone, a PDA, a laptop equipped with 802.11 capabilities and so on). We assume that a host is *intermittently connected* to the other hosts present in the system by means of heterogeneous networking equipments. In other words, we consider hybrid systems composed of mobile and fixed parts and assume that hosts get in reach of each others by moving. An instance of the middleware is running on each mobile device. We indicate the instance of the middleware running on the host k with $MW(k)$. We refer to a generic application i running on $MW(k)$ with $A(i,k)$. Hosts have a variable amount of memory that we call *Message Buffer*, used to temporarily store messages.

Messages are the fundamental entities of our system. They are composed of two parts: the data (i.e., the message body) and the message headers. Let us briefly analyse the message headers. Every message is characterized by a unique *messageId*[3]. It also contains the identifier of the host and the application that has sent the message (defined, respectively, in the fields *senderId* and *applicationId*). Every message also has a finite expiration time[4].

Messages are transferred from the Message Buffer of a host H_A to that of a host H_B using *forwarding mechanisms* (i.e., the message is transferred from H_A to H_B then the message is deleted from the Message Buffer of H_A) or *replication mechanisms* (messages are copied from H_A to H_B and the copy on H_A is maintained).

The mechanisms for the delivery of the messages to a specific host or location are encapsulated in the framework. Messages should generally be forwarded (and stored) or replicated as close as possible to the recipients of the messages and, finally, into their Message Buffer, so that the recipients are able to extract and process them. The middleware framework that we are describing allows for the integration of different algorithms and protocols. There are many possible ad hoc solutions that suit best depending on the application scenarios; the definition of the possible forwarding and replication mechanisms and algorithms is outside the scope of this paper. However, we are currently investigating two possible approaches to the dissemination which could be plugged into the framework: the first is the design of gossip-style dissemination algorithms, based on epidemic techniques [14, 13]; the other is the definition of intelligent forwarding protocols based on the history of the system [12]. The underlying protocols may only partially support the general semantics of the primitives presented in Section 3. For example, it may be the case of protocols that do not support geocasting. On the other hand, there may be situations where geographical information are not available (also temporarily) so that location-aware communication is not achievable.

The *status* field determines the phase of the message delivery (i.e., whether the message is to be delivered, being delivered, to be acknowledged or acknowledged). The semantics of the transitions between the different states is presented in Section 4.4. We now briefly introduce Mobile UNITY, the notation we will use for the formalization of the semantics of the primitives.

4.3 Mobile UNITY

Mobile UNITY [19] is an extension of the UNITY nota-

[3]A unique messageId can be derived by the composition of the identifier of the host and a number generated by a counter that is incremented when the primitive is invoked.

[4]In the remainder of the paper, we assume that the clocks of the systems are synchronised, so the expiration time is a timestamp obtained by summing the current time when the message is dispatched and its time validity defined by programmers. The problem of time synchronisation in delay tolerant networks is discussed in [5].

tion proposed by Chandy and Misra for the formal study of distributed mobile systems. It has been used for the specification of different systems and applications (for instance, for the definition of fine-grained code mobility paradigms [9]).

We now introduce the main concepts of Mobile UNITY that we will use to formalise the system. A Mobile UNITY specification is composed of several programs (that are the basic units of definition), a **Components** part and an **Interactions** part. A program is associated to a location variable λ, which indicates the location of the executing program. The **declare** section of a program contains the declaration of the variables used in the program separated by the symbol []. The **initially** section initializes the variables. The **assign** section contains the program statements. A statement can be guarded by the clause following the **if**. The statements separated by [] are executed in a non deterministically sequentially interleaved way. Sequences of statements that have to be executed atomically are separated by || and enclosed between the symbols $\langle$ and $\rangle$. The **Components** section is used to specify the entities of the system. Every Mobile UNITY program contains indexes (like i and k in our case) after the name of the program (i.e., $M(k)$). This allows for the creation of multiple instances of the same program in the **Components** section. The **Interactions** section contains statements about the communication between components. In this section we specify the variables that are shared between different programs (by using the symbol =). Variables may be *transiently* shared or, in other words, they may be shared *only when* the programs are running in the same location (using the symbol $\approx$ in a guarded instruction with the keyword **when** to define the condition). The two clauses **engage** and **disengage** are used to specify the values of the variables respectively after the co-location and after the subsequent possible separation. If no **engage** value is specified, the variable remains in an inconsistent state. If no **disengage** value is specified, the variable retains the value it had before the disconnection. The semantics of the instructions is rather straightforward, since it is very similar to classic imperative languages, such as Pascal[5].

In order to model the concept of communication region, we have extended the notation of Mobile UNITY by associating a program not only to λ but also to the variable γ. Therefore, the structure of a program will be the following:

program *P(i)* **at** λ **in** γ
...
end

γ indicates, as explained, the logical location of the program in terms of connectivity. We do not explicitly indicate the dependency on time in γ. However, as for λ, it is time dependent, since a host may be located in different physical locations and connected to different logical communication regions in different instants of time. In other words a host can move from the connectivity island of the village (that we indicate with $\gamma_{village}$ in Figure 1) to the communication region composed of all the hosts of the Internet (indicated with $\gamma_{Internet}$).

[5]However, in Mobile UNITY, the notation:

$$\langle \textbf{ op } quantifiedVariables \ : \ range \ :: \ expression \rangle$$

is used to indicate that the variables from *quantifiedVariables* can have all possible values in the *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each instantation of the variables is substituted in the expressions, producing a multiset of values to which **op** is applied.

4.4 Middleware Architecture

We now give semantics to the communication primitives indicated in Section 3. We start by specifying the middleware architecture by using Mobile UNITY. The complete semantics is defined in Figure 3. Given the syntax and the parallel nature of UNITY [2] the mapping of the formal specification of our middleware architecture into a multithreaded implementation is straightforward. We now analyse the most important and interesting aspects of this formalisation.

A generic application is described as one program $A(i,k)$ and a generic instance of the middleware as another Mobile UNITY program $MW(k)$, where k is the index of the host and i the specific index of an application. The communication between the middleware and the applications running on top of it is modelled using shared buffers (Figure 2).

The middleware provides the primitives for sending (**send**()) and receiving (**receive**()), described in Section 3, to the applications. The **Components** section indicates the instances of applications and middleware running. The **Interactions** section indicates how the communication happens among the different programs representing the middleware platforms running on each hosts and the applications. We now describe the specification in more details.

Let us consider the specification of an application depicted in Figure 3 by the program $A(i,k)$. As it is possible to observe in the **assign** part, the behaviour of each application is modelled by using four statements. *Send* and *Receive* respectively represent the invocation of the **send**() and **receive**() primitives. *Idle* models the possible idle states of the application, whereas *Process* is used to represent the execution of other tasks performed between the invocation of the communication primitives. In these statements, the **send**() and **receive**() have exactly the same signature which has been described in Section 3. The boolean variable *wait* is used for synchronization purposes.

The application communicates with the middleware through a number of shared variables, which are depicted in Figure 2. The *inBuffer* and *outBuffer* channels are used for input and output between the middleware and the applications running of top of it. When an application invokes the **send**() primitive, a send request is issued and inserted in the buffer of the send requests (represented by the variable *sendRequests*) at a certain position p (Figure 2) to be picked up by the middleware; the corresponding message containing the parameters of the primitive, is inserted in the output buffer at the same position p. When the application starts, all the variables are undefined (as specified in the **Initially** section). The send requests are used to maintain at middleware level the information specified in the primitives (such as timeouts) and the delivery status of the messages.

Similarly, the receive requests are issued when a **receive**() primitive is invoked. The receive requests (represented by the variable *receiveRequests*) contain the information related to the sender and the location of the sender of the expected messages. If the blocking time of the **receive**() is zero and the host is located where the operation has to be performed, the middleware checks if there is a message matching that

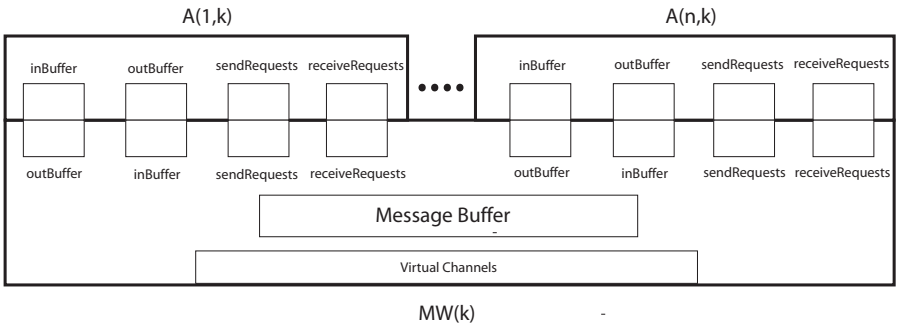

Figure 2: Abstract model of the architecture.

request. Otherwise, the receive request is valid and stored in the buffer for the interval of time declared in the *tBlock* field of the primitive. These operations are performed by the function *readMessageFromBuffer()*.

The middleware behaviour is specified in the $MW(k)$ program. The program has a number of variables: *sendRequests*, *receiveRequest*, *inBuffer*, *outBuffer* are used for the communication with the applications running on the middleware. They are represented by matrices, where each row is shared with the corresponding variables of the applications. As shown in the **Interactions** section, the item at position j of a shared array of an application $A(i,k)$ corresponds to the row element at position (i,j) of the matrix variable of the middleware $MW(k)$.

More precisely, the middleware maintains one input and one output buffer for each application. The input buffer of an application (indicated with *inBuffer*) is shared with one of the output buffers of the middleware (indicated with *outBuffer*). Simmetrically, the output buffers of each application are shared with one of the input buffer of the middleware. Each row of the *sendRequests* and the *receiveRequests* matrices are shared with the variables with the same name of the applications, following the specification contained in the **Interactions** section.

An array of boolean variables *wait* is used for the synchronisation (Figure 3). The size of this array is equal to the number of applications running on the middleware. Each item of the array corresponds to one application. *wait* is *true* when a blocking **send**() or a blocking **receive**() are invoked by the application.

The **assign** section of the middleware program performs a number of operations. The Mobile UNITY semantics imposes that each of these operations is chosen non deterministically and fired, when its guards are valid, an infinite number of times. The operations correspond to three tasks of the middleware: the *MessageChecker*, the *MessageDispatcher* and the *MessageBufferManager*. As shown in Figure 3, the *MessageChecker* performs many fundamental operations: it checks the timeouts of the send and receive requests. It handles acknowledgment messages in the Message Buffer corresponding to a pending send request. Finally, it checks if there is a message matching any receive request in the Message Buffer. The *MessageDispatcher* task puts messages not yet sent into the Message Buffer. When a message is sent, the middleware has to dispatch it, in other words, the message has to be inserted in the Message Buffer (see the *MessageDispatcher* section in the specification). The status of the send request is initially set to *TOBEDISPATCHED*. After that, the corresponding message has been dispatched, in the case of non blocking **send**(), the send request is deleted; in the case of a blocking **send**(), the *status* field of the send request is set to *TOBEACKNOLEDGED*. If the send request is generated by a blocking **send**(), it will be maintained for the blocking interval specified in the primitive call, waiting to be acknowledged. In the meanwhile, the application is blocked; as you can observe in the specification of the *MessageChecker* thread, after the acknowledgment (the *status* field is set to *ACKNOWLEDGED*) or, after the expiration of the message, the control returns to the application (in this case, the *status* field will assume the value *EXPIRED* and the variable *wait* will be set to *true*). The third task, the *MessageBufferManager*, applies the dissemination strategy and deletes the messages from the Message Buffer, if they are not valid anymore (i.e., they have reached the expiration time).

The communication between different instances of the middleware is based on the abstraction of virtual channels. These are defined by the array variable *vInChannels* and by the array variable *vOutChannels* respectively for the input and output operations. The symbol $\perp$ is used to indicate the unreachability of a certain host. A virtual channel between two instances of the middleware exists (i.e., these variables are shared and hold values different from $\perp$) when they are in the same communication region γ, as defined in the last four lines of the **Interactions** section. The **engage** and **disengage** statements are used to define the values that the channels assumes, respectively before a connection and after a disconnection (i.e., the value of the variable γ is the same or not). Let us consider again our example: a virtual channel between a computer in the village and the bus will exists (i.e., the values of the variables will not be $\perp$) when the bus will be connected to the wireless network of the village. The virtual channels can be implemented by using different mechanisms, such as sockets.

The operations performed to apply the dissemination strategy will involve the hosts that are in the same γ at a certain time t by means of the virtual channels. In other words, the messages are forwarded and exchanged among the hosts of the same communication region according to the given dissemination strategy.

We now use an example of a **send**() operation to show how the semantics describes the specific behaviour of the primitives. Let us consider the case of the communication between a space probe and and a robot, which we briefly introduce in Section 1, like PathFinder or Spirit [15], roaming on the surface of a planet, for example Mars, by means

System *Delay-Tolerant-Network-Middleware*
Program *A(i,k)* **at** λ **in** γ
declare
$sendRequests$: array of SendRequest $\cup$ $\{\perp\}$ [] $receiveRequests$: array of ReceiveRequest $\cup$ $\{\perp\}$ []
$messageBuffer$: array of Msg $\cup$ $\{\perp\}$ [] $inBuffer$: array of Msg $\cup$ $\{\perp\}$ []
$outBuffer$: array of Msg $\cup$ $\{\perp\}$ [] $wait$: boolean
initially
$sendRequests = \perp$ [] $receiveRequests = \perp$ [] $inBuffer = \perp$ [] $outBuffer = \perp$ [] $wait = \perp$
assign
$Send$: $send(recipient, recipientLocation, senderLocation, tExp, tBlock, reliability)$
if $wait = false$
[] $Receive$: $\langle receive(sender, senderLocation, receiverLocation, tBlock) \parallel readMessageFromBuffer()\rangle$ **if** $wait = false$
[] $Idle$: $\langle randomIdleInterval = random() \parallel sleep(randomIdleInterval)\rangle$ **if** $wait = false$
[] $Process$: $doSomething()$ **if** $wait = false$
end
Program *MW(k)* **at** λ **in** γ
declare
$sendRequests$: matrix of SendRequest $\cup$ $\{\perp\}$ [] $receiveRequests$: matrix of ReceiveRequest $\cup$ $\{\perp\}$ []
$inBuffer$: matrix of Msg $\cup$ $\{\perp\}$ [] $outBuffer$: matrix of Msg $\cup$ $\{\perp\}$ []
$wait$: array of boolean [] $inVChannels$: array of Integer [] $outVChannels$: array of Integer
initially
$messageBuffer = \perp$ [] $inVChannels = \perp$ [] $outVChannels = \perp$
assign
$MessageChecker$:
$\langle\langle[]r, z :: \langle sendRequests(r, z).status := EXPIRED \parallel wait(r) := false$
if $sendRequests(r, z).tBlock > tCurrent\rangle$
[] $\langle[]r, z :: \langle receiveRequests(r, z).status := EXPIRED \parallel wait(r) := false\rangle$
if $sendReceive(r, z).tBlock > tCurrent\rangle$
[] $\langle[]r, z :: \langle sendRequests(r, z).status := ACKNOWLEDGED \parallel wait(r) = false\rangle$
if $\exists v :: sendRequests(r, z).messageId = messageBuffer(v).messageId \wedge$
$messageBuffer(v).status = ACK\rangle$
[] $\langle[]r, z :: \langle addMessageToOutBuffer(messageBuffer(v), r, z) \parallel$
$receiveRequests(r, z).status := RECEIVED \parallel wait(r) := false\rangle$
if $checkMessage(receiveRequests(r, z), messageBuffer(v)) = true \wedge$
$receiveRequests(r, z).status := WAITING\rangle$
[] $\langle[]r, z :: \langle\langle addMessageToOutBuffer(messageBuffer(v), r, z) \parallel$
$receiveRequests(r, z).status := RECEIVED\rangle$
if $checkMessage(receiveRequests(r, z), messageBuffer(v)) = true$
$receiveRequests(r, z).status := EXPIRED$ **otherwise**$\rangle$
if $receiveRequests(r, z).status = NOWAIT\rangle\rangle$
$MessageDispatcher$:
$\langle\langle[]v :: \langle addToMsgBuffer(outBuffer(r, z)) \parallel inBuffer(r, z) := \perp \parallel sendRequests(r, z) := \perp\rangle$
if $sendRequests(r, z).status = TOBEDISPATCHED \wedge$
$sendRequests(r, z).senderLocation = \lambda \wedge sendRequests(r, z).tBlock = 0\rangle$
[] $\langle[]v :: \langle addMessageToMessageBuffer(outBuffer(r, z)) \parallel$
$sendRequests(r, z).status := TOBEACKNOWLEDGED\rangle$
if $sendRequests(r, z).status = TOBEDISPATCHED \wedge$
$sendRequests(r, z).senderLocation = \lambda \wedge sendRequests(r, z).tBlock > 0\rangle\rangle$
$MessageBufferManager$:
$\langle applyDisseminationStrategy()$ [] $\langle[]v :: deleteFromMsgBuffer(MessageBuffer(v))$
if $messageBuffer(v).tExp > tCurrent\rangle\rangle$

Components
$\langle[]i, k :: A(i, k), MW(k)\rangle$

Interactions
$\langle[]j :: MW(k).sendRequests(i, j) := A(i, k).sendRequests(j) \parallel$
$MW(k).receiveRequests(i, j) := A(i, k).receiveRequests(j) \parallel$
$MW(k).inBuffer(i, j) := A(i, k).outBuffer(j) \parallel$
$MW(k).outBuffer(i, j) := A(i, k).inBuffer(j) \parallel MW(k).wait(i) = A(i).wait\rangle$
[] $\langle[]z :: MW(k).inVChannels(z) \approx MW(z).outVChannels(k) \parallel$
$MW(k).inVChannels(z) \approx MW(z).outVChannels(k)$ **when** $MW(k).\gamma = MW(z).\gamma\rangle$
disengage $\langle[]z :: MW(k).inVChannels(z), \perp \parallel MW(z).outVChannels(k), \perp\rangle$,
engage $\langle[]z :: MW(k).inVChannels(z), \perp \parallel MW(z).outVChannels(k), \perp\rangle$
end
end

Figure 3: Specification of the middleware using Mobile UNITY.

of an intermediate network of satellites orbiting around it. Let us suppose that the space probe has to deliver a software update to the robot (that we call *robot1*), since a direct communication between the Earth and the robot is not possible, as the robot is roaming on the side of the planet opposite to the Earth. The software update has to be transmitted when the space probe is near the planet, otherwise the signal cannot reach the satellites. Let us also imagine that the space probe will wait for an acknowledgement from the robot and, because of the distance, it allows for a delay up to 10 minutes. Assuming that the software update is contained in a message m and the robot is located in area called Gusev Crater, the **send**() primitive will have the following form:

send(m, *robot1*, *GusevCrater*, *MarsSpace*, 10m, 10m, 100)

Let us suppose that on the space probe, on the robot and on all the intermediate satellites are running different instances of the middleware: the instance $MW(1)$ and an application that has to send the software update called $A(1,1)$ are running on the space probe, whereas the instance $MW(2)$ and an application $A(1,2)$ (listening for software updates) are running on the robot. When the **send**() is invoked, the message m is inserted in the *outBuffer* at position 1 of $MW(1)$, that corresponds to the same position of *inputBuffer* of $A(1,1)$. The corresponding send request is also inserted in the *sendRequest* shared variable used as communication channel. The status of the send request is equal to *TOBEDISPATCHED*. Then the message is inserted in the buffer by the *MessageDispatcher* task when the space probe enters the geographical area called *MarsSpace*. The status of the corresponding send request is set to *TOBEACKNOWLEDGED*. Assuming, for example, an epidemic algorithm for the dissemination process, m is then replicated in all the buffers of the satellites that are in reach of the *MessageBufferManager* . The middleware instances that are running on these intermediate satellites will check if there are receive requests for this message. The message will be replicated on all the other hosts in reach transitively and eventually transmitted to the robot and stored in the buffer of $MW(2)$. Considering a non optimised epidemic protocol, messages are replicated on all the intermediate hosts, since the desired reliability is equal to 100 [22]. Then the message is read by $A(1,2)$ by means of a generic **receive**(*,*,..). Since the status of the message is equal to *TOBEACKNOWLEDGED*, an acknowledgement message is then sent back to the space probe by exploiting the same dissemination protocol. This type of messages has the *status* field set to *ACK*. A detailed description and formalisation of the acknowledgement mechanisms have not been included in this paper for reasons of space. After the acknowledgement message is received (i.e., inserted in the buffer of $MW(1)$), it is then retrieved by the *MessageChecker* task and, if the timeout has not expired, the corresponding send request status is set to *ACKNOWLEDGED*. The expiration time field is used to delete the buffer after 10 minutes; as shown in Figure 2 this operation is performed by the *MessageBuffer* task.

In the example, the information related to the receiver location is not used for the delivery of the message, since the epidemic algorithm does not exploit geographical information for routing. The information would be useful in the case of underlying forwarding mechanisms based on the location of the hosts, such as [6]. If this was the case, in our example, the information contained in the receiver location field would be useful to decide whether to avoid the replication of the message on all the satellites, by shipping it directly to the one covering the area over the Gusev crater.

5. DISCUSSION AND RELATED WORK

In the recent years, the research community has proposed new paradigms and architectures for communication and coordination in the general area of pervasive computing. The proposed solutions have been founded on several different mechanisms and abstractions such as the sharing of tuple spaces [17] or of more complex data structures [8] and events [10], which go beyond the traditional synchronous communication mechanisms imposed by standard middleware systems. However, the pervasive computing scenario seems to offer more opportunities than those exploited for more complex communication primitives. This is even more true if we extend to study delay tolerant networked systems. The challenges posed by this scenario were firstly discussed by Fall in [5]. Some examples of existing prototypes of delay tolerant systems have been presented in Section 1, such as DakNet [16] and Data Mules [20]. However, in these works, the authors do not discuss the potential of these systems in terms of primitives and the set of operations which could be performed on such systems.

To our knowledge, our work represents the first attempt to the design and the specification of communication primitives in delay tolerant networks. An analysis of the requirements of spatio-temporal primitives have been recently presented in [18]; however, this work discusses a communication framework that can be applied only to specific scenarios (such as location-aware communication in sensor ad hoc networks). Moreover, the proposed primitives are not able to capture the aspects of the interaction in delay tolerant systems, such as, for instance, the reliability of the probabilistic delivery process and the possibility of specifying the position of the reciprocal positions of senders and receivers. In this paper, we have presented not only a more general and expressive set of primitives, but also a middleware framework supporting the abstract programming interface.

In terms of models, there are many other powerful and expressive formal notations that focus on the modelling of different aspects of mobile computing. Many process algebra and calculi have been proposed, such as KLAIM [3], starting from the π-calculus, the first model of mobile concurrency proposed by Milner [11]. However, the π-calculus has no notion of location and space. Instead, these concepts are at basis of the Mobile Ambients [1] model proposed by Cardelli and Gordon, that focusses on the definition of hierarchically structured computational domains and naming scope issues. We have decided to use Mobile UNITY, since it provides abstractions to model the concept of intermittent connectivity and transient variables sharing in a very straightforward way.

In general, the current work in delay tolerant networking propose ad hoc solutions targeting particular problems in specific deployment scenarios. Moreover, they are more focussed on networking issues rather than on the definition of application-level abstractions and programming paradigms. The contribution and the novelty of our work is that it defines a flexible and expressive set of primitives and high-level abstractions for this challenging class of networks and in general for systems for which spatio-temporal aspects are important.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a set of primitives for communication in delay tolerant networks. As part of the framework we have defined a communication middleware which supports the primitives. We have also provided a formal specification of the primitives by using the Mobile UNITY notation. We consider this work as the foundation and the starting point of our investigation on the design and the formal definition of middleware for delay tolerant systems and, in general, for systems where spatial and temporal aspects are relevant. As part of our future work, we plan to verify the formal properties of the architecture and, possibly, of various dissemination strategies by using the proof logic of Mobile UNITY.

Another interesting issue is the choice of the values of the blocking time of the primitives and expiration time of the messages. In some scenarios, it is possible to estimate such timeouts, as in the Interplanetary Internet Project [7], where the delays are strictly related to the rotation and revolution of the planets and the orbits of the satellites. The same applies for systems where the message delivery is performed by mobile carries with predefined schedule, as in the case of a bus with a timetable that can be known in advance. However, in some systems, it is not possible to make these estimations *a priori*. Therefore, we believe that it is necessary to introduce feedback mechanisms in order to provide developers with some estimation of the average delivery delays (by using mechanisms similar to the Unix command *ping*) in order to tune the timeouts of the middleware.

Moreover, the primitives presented in this paper may be used to design more complex distributed systems. Namely, the middleware architecture could be extended with the addition of a layer for publish/subscribe systems for delay tolerant networks. Subscriptions and notifications may be delivered by means of mechanisms similar to those described in this paper. We want to investigate this issue further.

We also plan to study in detail the design of intelligent dissemination and forwarding strategies, in order to enrich the semantics of replication and dissemination just sketched in this paper. In this sense, we have started investigating different epidemic models and how they relate to mobile network topologies [13].

Another interesting aspect that we would like to investigate is an extension of the idea of location. We believe that it is possible to replace the definition of location with the broader concept of *context*. In fact, in many situations, we would like to be able to express not only the geographical position, but also other concepts such as co-location (i.e., the message has to sent only if the host is co-located with a certain host).

7. REFERENCES

[1] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of FOSSACS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Verlag, 1998.

[2] M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley, 1988.

[3] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[4] A. Doria, M. Uden, and D. P. Pandey. Providing Connectivity to the Saami nomadic community. In *Proceedings of the Second International Conference on Open Collaborative Design for Sustainable Innovation*, December 2002.

[5] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of SIGCOMM'03*, August 2004.

[6] T. Imielinski and J. C. Navas. Geographic Addressing and Routing. In *Proceedings of MOBICOM'97*, September 1997.

[7] Internet Society. Interplanetary Internet Project. http://www.ipnsig.org.

[8] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A Data Sharing Middleware for Mobile Computing. *International Journal on Personal and Wireless Communication*, 21(1):77 – 103, April 2002.

[9] C. Mascolo, G. P. Picco, and G.-C. Roman. A Fine-Grained Model for Code Mobility. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of ESEC/FSE '99*, volume 1687 of *Lecture Notes on Computer Science*, pages 39–56, Toulouse, France, September 1999.

[10] R. Meier and V. Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. In *22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, pages 639–644, June 2002.

[11] R. Milner. *Communicating and the Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[12] M. Musolesi, S. Hailes, and C. Mascolo. Adaptive routing for intermittently connected mobile ad hoc networks. In *Proceedings of the 6th International Symposium on a World of Wireless, Mobile, and Multimedia Networks (WoWMoM 2005). Taormina, Italy.* IEEE press, June 2005.

[13] M. Musolesi and C. Mascolo. Epidemic-style Communication Primitives Exploiting Network Topology Information. Technical report, Department of Computer Science, University College London, April 2005. Submitted for Publication.

[14] M. Musolesi, C. Mascolo, and S. Hailes. EMMA: Epidemic Messaging Middleware for Ad hoc networks. *Journal of Personal and Ubiquitous Computing*, 2005. To Appear.

[15] NASA. Mars Exploration Program Website. http://marsweb.jpl.nasa.gov/.

[16] A. S. Pentland, R. Fletcher, and H. Hasson. Daknet: Rethinking connectivity in developing nations. *IEEE Computer*, 37(1):78–83, January 2004.

[17] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press.

[18] G.-C. Roman, O. Chipara, C. Fok, Q. Huang, and C. Lu. A Unified Specification Framework for Spatiotemporal Communication. Technical report, Washington University, Department of Computer Science and Engineering, St. Louis, Missouri, 2004.

[19] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997.

[20] R. C. Shah, S. Roy, S. Jain, and W. Brunette. Data MULEs: Modelling a Three-Tier Architecture for Sparse Sensor Networks. Technical Report IRS-TR-03-001, Intel Corporation, January 2003.

[21] I. T. Union. Connecting remote communities. *Documents of the World Summit on Information Society*, 2003. http://www.itu.int/osg/spu/wsis-themes.

[22] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical Report CS-2000-06, Department of Computer Science, Duke University, 2000.

[23] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.