Opportunistic Software Deployment in Disconnected Mobile Ad Hoc Networks

Frédéric Guidec, Nicolas Le Sommer, Yves Mahéo

▶ **To cite this version:**

Frédéric Guidec, Nicolas Le Sommer, Yves Mahéo. Opportunistic Software Deployment in Disconnected Mobile Ad Hoc Networks. International Journal of Handheld Computing Research, 2010, 1 (1), pp.24-42. 10.4018/jhcr.2010090902 . hal-00452091

HAL Id: hal-00452091
https://hal.science/hal-00452091

Submitted on 1 Feb 2010

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

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

Opportunistic Software Deployment in Disconnected Mobile Ad Hoc Networks

Frédéric Guidec, Nicolas Le Sommer and Yves Mahéo
Valoria, Université Européenne de Bretagne, France

ABSTRACT

This paper presents a middleware platform allowing the dissemination of software components on handheld devices forming a disconnected MANET. It is based on a model that exploits peer-to-peer and opportunistic interactions between neighboring devices to deploy component-based applications, without relying on any kind of infrastructure network. Each device runs a deployment manager, which strive to fill a local component repository with software components so as to be able to satisfy the deployment requests expressed by the user. To do so the deployment manager interacts with peer managers located on neighboring devices, providing its neighbors with copies of software components it owns locally, while obtaining itself from these neighbors copies of components it lacks. The platform also provides communication facilities adapted to disconnected MANETs that notably allow efficient multi-hop exchanges.

Keywords: ubiquitous computing, pervasive computing, mobile ad hoc networks, software component deployment, opportunistic networking.

INTRODUCTION

An approach to handle the complexity of modern software applications is to define these applications as assemblies of software components. Software components are independent, reusable and replaceable units of software that are meant to fulfill a well-defined function in an application [Szyperski, 1998]. An important research topic about software components aims at defining how component-based applications can be deployed on a target platform. Some works about software deployment have contributed to identify interrelated activities needed for this deployment. For example in [Carzaniga et al., 1998, Lestideau et al., 2002] it has been proposed to distinguish between activities pertaining to the provision, the delivery, the installation, the configuration, the execution, the adaptation, and the removal of software components. In this paper we mostly focus on two of these activities, namely the provision and the delivery of software components.

The originality of our work lies in the fact

that the target platforms we consider for component deployment are disconnected mobile ad hoc networks composed of lightweight mobile devices capable of wireless ad hoc communication (e.g., laptops, netbooks, mobile Internet devices, smartphones). A mobile ad hoc network (MANET) is a network that can appear and evolve spontaneously as mobile devices themselves appear, move and disappear dynamically [Perkins, 2001]. Traditionally, MANETs are considered connected, allowing a device to communicate with any other in the network temporaneously, thanks to routing by the other devices. However, in many realistic conditions, for example when devices are distributed sparsely or irregularly, a MANET can become disconnected, and get fragmented into communication islands.

For the users of laptops and handheld devices, the prospect of deploying software applications on these devices as and when needed obviously appears as an attractive one, no matter if these devices communicate in infrastructure or in ad hoc mode. Yet, the specificities of MANETs, and especially those of disconnected MANETs, lead us to reconsider the software deployment problem in this particular context.

In this paper we describe a model for software component deployment on disconnected MANETs, as well as a platform that implements this model. The paper is organized as follows. We first motivate our work by showing how infrastructure-based networks and disconnected MANETs constitute radically different environments as far as the problem of software deployment is concerned. We then present the main characteristics of a platform we designed, which provides a specific communication support for disconnected MANETs and a protocol for software deployment in such networks. Finally, the results we obtained by running our middleware platform on a mobile ad hoc network simulator are presented before discussing about related works and concluding the paper.

RATIONALE

In this section we show that deploying software components in an ad hoc network raises issues that usually do not appear in infrastructure networks. As a reminder, we first describe how software component provision and delivery are usually performed in an infrastructure-based environment. We then show that a disconnected MANET presents additional constraints that need to be addressed specifically.

Software deployment in an infrastructure network

In an infrastructure network, some stable hosts can be in charge of storing components in so-called *component repositories*, and of implementing server programs capable of delivering these components on demand. Other hosts in the network can then behave as simple clients with respect to these servers. Whenever the owner –or the administrator– of one of the client hosts initiates the deployment of a new component-based software application on this device, the problem mostly comes down to locating at least one of the servers capable of providing the components required by this application, and downloading these components so they can be installed locally. A component may actually be provided by several servers, for example in order to balance the workload

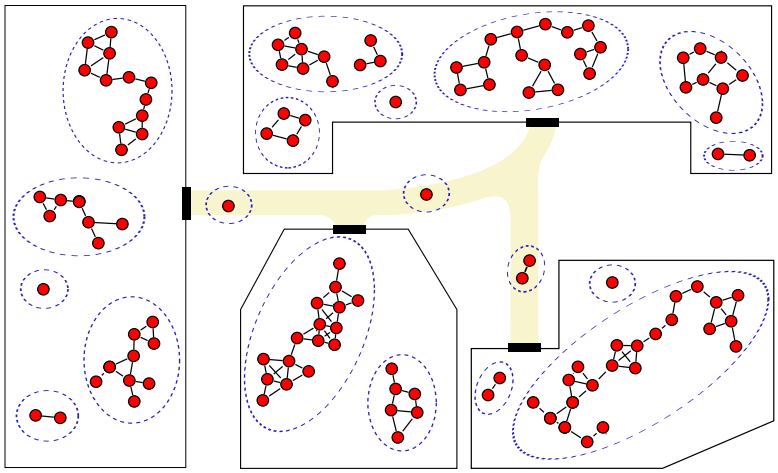


Figure 1: Illustration of a disconnected MANET

in the network, or to allow fault tolerance. In any case, once a client has identified a server that can provide a component, obtaining this component simply requires its download from the server to the client. Note that in such a context the deployment of a component on a given host can usually be considered as a "real time" operation: once a user has ordered the deployment middleware to locate and download a component, this operation can usually be performed immediately.

In the remainder of this section, we show that deploying components in an ad hoc environment can in contrast require a more lengthy process, which requires some middleware capable of enforcing a deployment strategy in the background on behalf of the user.

Disconnected mobile ad hoc networks

A MANET is formed spontaneously by a number of mobile devices communicating through radio interfaces (such as Bluetooth or Wi-Fi interfaces), without relying on any infrastructure. The devices in such a network are usually highly mobile and volatile. Device mobility results from the fact that each device is carried by a user, and users themselves move quite a lot. Device volatility is the consequence of the fact that, since the devices usually have a limited power-budget, they are frequently switched on and off by their owners.

A major characteristic of wireless ad hoc networks is that communication interfaces have a limited transmission range. Consequently any device can only communicate directly with neighboring devices. Multi-hop transmissions can be however obtained by implementing a dynamic routing algorithm on each device [Zhang, 2006, Pelusi et al., 2006]. Yet, most of the current dynamic routing algorithms cannot operate on many realistic networks that show a low density, a high volatility or a high mobility of nodes. These challenging networks are often called *disconnected* MANETs. Indeed, a disconnected MANET appears as a –possibly continuously changing–

collection of so-called “connectivity islands”. Mobile devices that belong to the same island can communicate together, using either multi-hop or single-hop transmissions. However, no temporaneous transmission is possible between devices that belong to distinct islands.

Figure 1 depicts an example of a disconnected MANET in which some laptops and handheld devices with Wi-Fi interfaces, symbolized with dots, are scattered in four buildings. The users that hold these devices can move freely inside and between the buildings. An opportunity for two devices to communicate (meaning they are within radio range of each other) is represented by a vertex between the two corresponding dots. The figure also shows the communication islands resulting from the limited radio range of the devices' interfaces.

Software deployment in a disconnected mobile ad hoc network

In a disconnected MANET, the traditional client-server deployment scheme is hardly applicable, for no device is stable and accessible enough to play the role of a server of components, maintaining a component repository and allowing client devices to access this repository whenever needed.

In the remainder of this paper, we present a model we propose in order to allow for these constraints. Basically, instead of being able to access a server whenever needed, each device must maintain a local component repository. A fully decentralized and opportunistic interaction model then makes it possible for a device to cooperate with its neighborhood, by allowing its neighbors to obtain copies of the software components available on its local repository, while itself benefiting from a similar service offered by its neighbors.

Consider the example shown in Figure 2, and assume that the owner of device *A* wishes to install on this device an application that requires components *c1*, *c2* and *c3*. In our example, *A* can obtain components *c1* and *c2* from device *B*. But as devices *C* and *E* –that both own a copy of component *c3*– are currently unreachable, *A* cannot readily obtain a copy of component *c3* from any of these devices. Yet *A* could obtain component *c3* from device *C* if this device was switched on by its user. It could also obtain this component from device *E* if *A*'s user happened to walk towards *E*, or if *E*'s user happened to walk toward *A*. A roaming device such as *D* may even serve as a benevolent carrier between *E* and *A*, transporting component c3 –and possibly other components as well– between separate islands, and thus contributing to the dissemination of software components and applications all over the network.

This example shows that when the owner of a mobile device participating in a disconnected MANET requests the deployment of a component-based application on this device, there is no guarantee that this request can be satisfied immediately, as there is no guarantee that the components required for this deployment are accessible in the neighborhood. Yet, since the structure of an ad hoc network can change continuously and unpredictably, the fact that a given component cannot be obtained at a given time does not involve that this component will remain inaccessible in the future. There is thus a need for some deployment middleware capable of ensuring the collection

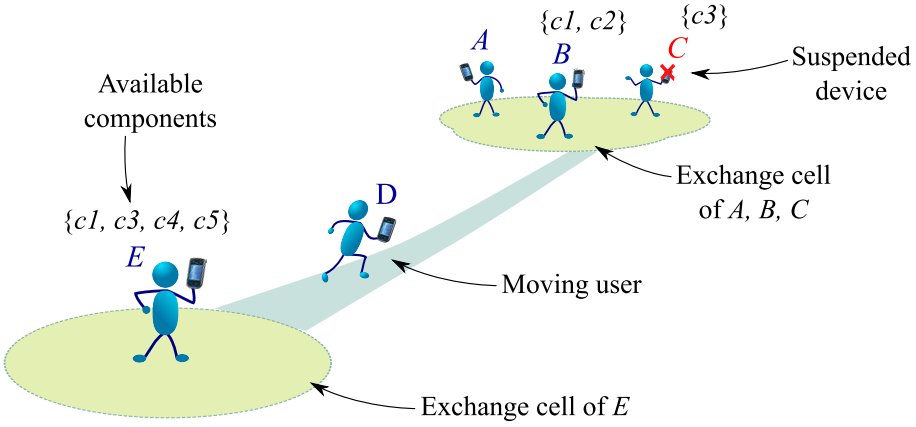


Figure 2: Illustration of software component deployment in a disconnected MANET

of missing components in the background in order to satisfy the user's needs.

DEPLOYMENT PLATFORM

In this section, we present an overview of CODEWAN (*COmponent DEployment in Wireless Ad hoc Networks*), a platform we designed in order to support the deployment of component-based software applications on disconnected MANETs. CODEWAN implements a cooperative model, whereas neighboring devices interact opportunistically in order to discover and exchange software components. Each device implements a local component repository, and a deployment manager is responsible for maintaining this repository on behalf of the user. Any component stored in the repository can be used to assemble and start an application locally. Copies of this component can also be sent on demand to neighboring devices.

Architecture of the CODEWAN platform

As shown in Figure 3, the platform is composed of three layers: an opportunistic communication layer, a component deployment layer, and an execution layer. The execution layer is meant to provide a framework for assembling and running component-based applications. The CODEWAN platform is not strongly dependent on a specific execution framework, or on a particular component model. Actually the focus in this platform is put on the dissemination of software components rather than on the assembly and execution of component-based applications. The only condition is that components in the model considered can be transmitted and stored in packages, and that the execution framework can be adapted so as to take components from the local repository maintained by the platform's deployment manager. CODEWAN currently interfaces with the JAMUS execution framework [Le Sommer and Guidec, 2002], with JULIA (a framework that implements the Frac-

tal component model [Bruneton et al., 2004]), with Cubik (a distributed component platform based on Fractal [Hoareau and Mahéo, 2008]), and with Felix[1] (a service-oriented framework for OSGi *bundles*). It could also be easily interfaced with OSGi platforms implementing a secure deployment of bundles such as Secure-Felix [Parrend and Frénot, 2007] or J2ME platforms in order to deploy MIDlets on smartphones [Muchow, 2002].

Opportunistic communication layer

The communication support in CODEWAN was designed to disseminate so-called *transfer documents* in a disconnected MANET. A transfer document combines a header that specifies the conditions required for disseminating the document in the network, and a payload. The header is expressed in XML. It indicates typically the document's source and destination, the expected propagation scope for this document, etc. The payload can be composed of application descriptors or software packages themselves.

The communication layer provides services for encapsulating transfer documents in UDP datagrams. Large documents can be fragmented and then transported in distinct, smaller transfer documents that each can fit in a single UDP datagram. The communication layer of course supports the re-assembly of such fragments after they have been received from the network.

The communication layer supports both unicast and broadcast transmissions. Radio-based transmissions are of course only possible between direct neighbors (that is, mobile devices that are within radio transmission range of each other), but in order to allow each mobile device to interact with a larger set of neighbors the communication layer also implements temporaneous multi-hop forwarding algorithms. Experiments we conducted previously show that these algorithms significantly improve the efficiency of information dissemination in disconnected MANETS [Haillot and Guidec, 2008]. Thus, a device can either broadcast a message that will reach all devices in its k-neighborhood (that is, all devices that are accessible with up to k successive transmission hops), or send a message in unicast mode to any device in this k-neighborhood. The maximum value for k is a parameter in the platform configuration.

Remember however that since the ad hoc network we consider is fragmented in a number of connectivity islands, there is very little chance that a device can ever send a message to all other devices in the network using only temporaneous multi-hop forwarding. In the best case, a message sent by a device can only reach hosts that belongs to the island it itselfs belongs to. Supporting temporaneous multi-hop forwarding between mobile devices is therefore not sufficient in the kind of network we consider. It does not prevent all mobile devices from storing packages and descriptors in their local repository, so they can bridge the gap between different islands by carrying these elements while moving in the network.

Broadcast message forwarding

Multi-hop broadcasting in a MANET is known to be a bandwidth-consuming ac-

[1] http://felix.apache.org

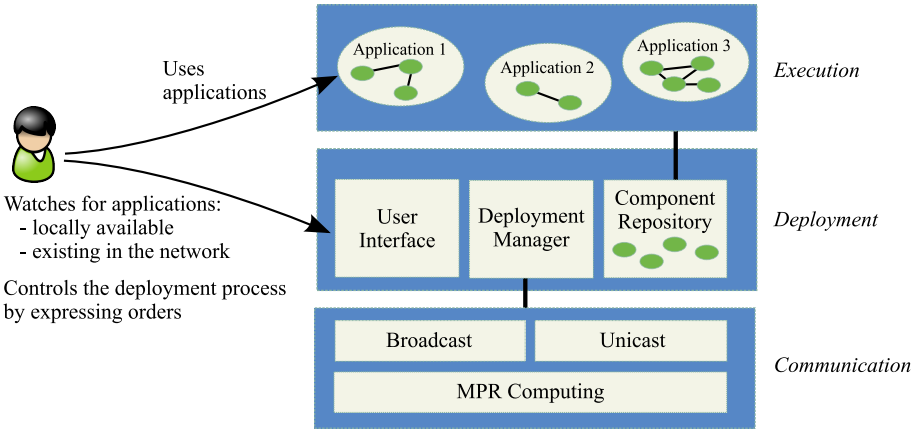


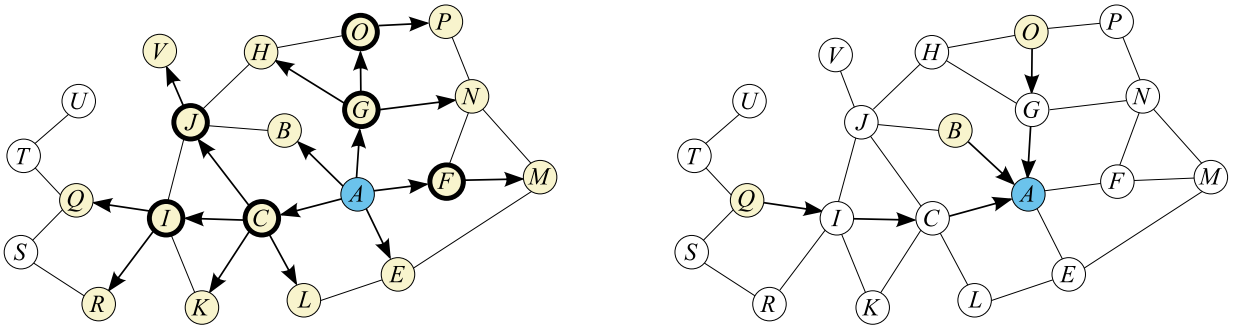
Figure 3: Architecture of the CODEWAN platform

tivity, which can occasionally lead to the so-called "broadcast storm" problem. In order to limit the overhead due to message broadcasting, we implemented a mechanism that is inspired from that used in the Optimized Link State Routing (OLSR) protocol for diffusing link-level information in the network [Clausen and Jacquet, 2003, Qayyum et al., 2002]. Basically, each node regularly selects a subset of its direct neighbors as multi-point relays (MPR), and it then relies exclusively on these MPRs for forwarding broadcast messages beyond its own radio range.The scope of a broadcast can be controlled by specifying how many hops a message is allowed to perform while being relayed by MPRs. Figure 4-a shows an example, where host *A* broadcasts a message. In this example the message is allowed to propagate up to its 3-hop neighbors, but not further.

The algorithm used by each host to construct its MPR set is not detailed in this paper for the sake of brevity. Indeed we use the same algorithm as that described in [Qayyum et al., 2002]. Basically, each host must periodically broadcast a control message in order to inform its direct (one-hop) neighbors about its presence in the network, while informing these neighbors about its own current vision of its neighborhood. By receiving such control messages, each host can identify its one-hop and two-hop neighbors, and use this information to calculate its MPR set. With the approach described in [Qayyum et al., 2002], specific control messages are broadcast periodically, that contain the information needed for calculating MPR sets. In our implementation, this information is piggy-backed in the announcements the deployment layer of the CODEWAN platform must also broadcast periodically. Thus the calculation of MPR sets does not imply sending any additional message in the network: both kinds of control information (required by both layers of the protocol) are broadcast together in the network.

Unicast message forwarding

In CODEWAN, unicast messages serve as replies to announcements diffused by compo-



(a) Host *A* broadcast a message up to its 3-hop neighbors (via multi-point relays)

(b) Hosts *B*, *O* and *Q* send unicast messages to host *A* (using source routing as a forwarding mode)

Figure 4: Illustration of the two kinds of temporaneous message forwarding supported

nent owners. Unicast messages must thus be forwarded towards the sender of a broadcast message. Source-routing is used as a means to perform this forwarding. Each broadcast message that propagates in the network encapsulates a history of the hosts by which it has been forwarded so far. Thus, whenever the receiver of a broadcast message decides to reply to this message, the path for sending this reply to its source is simply deduced from the path the former broadcast message has followed before reaching the receiver. Note that, in order to be effective, this approach requires that when a host decides to reply to a broadcast message, this reply is sent immediately after the broadcast message has been received. In such conditions, the path the broadcast message has followed downwards to reach the receiver is still valid in the network, so it can be followed upwards to the sender of the broadcast message.

Consider again the example shown in Fig. 4-a, and assume that hosts *B*, *Q*, and *O* decide to reply to the message broadcast by *A*. Figure 4-b shows how their replies can propagate upwards along the path the broadcast message has just followed downwards, each reply containing a specification of the path it must follow before reaching host *A*.

Overview of the component deployment layer

In the remainder of this paper we focus on the description of the central layer of the platform. The deployment manager is implemented in this layer, together with the component repository this manager is in charge of maintaining. The repository is a place where software components can be stored locally on a mobile device. Components stored in this repository are thus readily available for the execution framework that constitutes the upper layer of the platform. The deployment manager takes orders from the user, and interacts with peer managers that reside on neighboring devices in order to fill the local repository with components required by the user, while providing its peers with components they need in order to satisfy their own users.

Figure 5: Screenshot of the graphical interface of the deployment manager (running on an iPaq)

User interface

The deployment manager implements an interface that provides the user with a view of all the applications it is aware of. Using this interface the user can observe the status of each application. At any time a given application is either:

- *installed locally* (meaning that this application is either already running in the local execution framework, or ready to be loaded and started in this framework);
- *installable* (meaning that all the components required for running this application are available in the local repository, so the application could be installed immediately if the user requested it);
- *not installable yet* (meaning that some of the components required by this application are not present in the local repository).

Besides observing the status of each application, the user can modify this status, requesting for example that an application be started (which requires that this application be already installed locally), or that an application be uninstalled (and all its components removed from the repository). Additionally the user can initiate the deployment of an application, thus instructing the deployment manager to try to obtain any missing component for this application from neighboring devices.

CODEWAN implements a basic interface that runs in console mode. Additionally, a number of graphical interfaces have been designed in order to facilitate the interaction between the user and the deployment manager running on a mobile device. For example Figure 5 shows an interface that was designed for personal digital assistants. Note that, as a general rule, the user is only presented a view of the applications the deployment manager is aware of. The deployment manager could actually provide the user with more detailed information, for example by showing the status of any basic component that resides in the local repository. Yet we expect that in most circumstances a user should find it more convenient to consider only the status of full-featured applications, thus letting the deployment manager deal with petty details such as component retrieval and storage.

Software components, applications and packages

The deployment of component-based applications implies that components be transmitted in the network, and stored in local repositories. Before they can be loaded and executed in a runtime framework, software components are encapsulated in so-called *software packages*, that can be considered as storage and transfer envelopes for these components. Besides encapsulating the actual code of the components, software packages can additionally encapsulate some data required by a software component or application. They can also encapsulate documents describing the overall architecture of a component-based application (such as CCM component assemblies [OMG, 2002], architecture descriptors in the Fractal model [Bruneton et al., 2004], or manifests of OSGi bundles).

Package descriptors

Each software package in CODEWAN is associated a package descriptor that provides meta-information about the package. Figure 6 shows the descriptor associated with the main component of a messaging application. This descriptor provides information about this application, such as its name, version number, provider, etc. It also indicates that in order to be assembled this application requires components that can themselves be found in three other software packages. This example shows that when the components encapsulated in a particular package depend on components that are encapsulated in other packages, this information is mentioned explicitly in package descriptors. Dependencies between packages can also appear when a package contains only the description of the architecture of an application, while other packages encapsulate components that are required for assembling this application, or data that are needed for running this application.

Notice that in the example shown in Figure 6, attribute *type* indicates that package "JMessagerImpl" actually describes an application, rather than a single collection of components. This information is important for the deployment manager, as it makes it possible to present the user with a view of all known applications (through the interface discussed previously), rather than with a view of all known software packages.

Software packages

As mentioned above software packages can encapsulate software components, as well as plain data or application architecture descriptions. A software package usually encapsulates its own descriptor, but this descriptor can also be extracted from the package and processed separately whenever needed. Figure 7 shows the general structure of a software package, which in this particular case combines the package's descriptor together with a collection of software components.

The actual format we use for encapsulating the code of software components in transfer documents is not detailed in Figure 7 for the sake of readability. Indeed, the current implementation of the CODEWAN platform allows meta-information (such as application and component descriptors) to be formatted as XML documents, while the code of a component is kept in binary form. Moreover, all XML documents are systematically

```
<package-descriptor>
   <general-information
     type="application"
     name="JMessagerImpl"
     version="1.3"
     provider="Laboratoire Valoria"
     category="communication/messaging"
     summary="JMessager is a P2P messager"/>
   <dependencies>
    <required-package
      name="JMessengerUI" version="1.2"/>
    <required-package
      name="P2PAsyncDissemination"/>
    <optional-package
      name="AddressBook" version="2.0"/>
   </dependencies>
</package-descriptor>
```

Figure 6: Example of a package descriptor

```
<software-package>
   <package-escriptor>
     ...
   </package-descriptor>
   <software-component> ... </software-component>
   <software-component> ... </software-component>
</software-package>
```

Figure 7: Example of a software package

compressed using the LZ77 algorithm, as defined in [Ziv and Lempel, 1978]. This approach compensates for the verbosity of XML structures, since most descriptors in CODEWAN can be compressed with an average 12:1 compression ratio.

Communication protocol between deployment managers

Interaction between the deployment managers running on neighbor devices relies on a protocol that is inspired from the so-called Autonomous Gossiping (A/G) algorithm [Datta et al., 2004], which itself fits in the general model of epidemic routing, as proposed in [Vahdat and Becker, 2000]: transient contacts between mobile devices are exploited opportunistically by the deployment managers to exchange application descriptors and software packages according to each manager's needs. Indeed, for each application it has been ordered to install locally, a deployment manager maintains a list of the software packages it is missing, and must thus obtain from other devices (some kind of a "shopping list", actually).

Periodically, each deployment manager broadcasts an announcement in order to inform its neighbors (if any) about its identity and about its current "shopping lists" (one list for each application whose installation is in progress). This announcement can optionally include a catalog of descriptor and package identifiers, as explained below.

By sending an announcement periodically, a deployment manager informs its neighbors about its presence and about the software packages it is interested in. Conversely, by receiving similar announcements a deployment manager discovers its neighbors, and learns about the packages they are looking for. By matching its neighbors' "shopping lists" against the packages it maintains in its local repository, a deployment manager can identify packages that might be of interest to them. It can thus build a catalog containing the identifiers of these packages, and incorporate this catalog in its next announcement.

In the current implementation of our protocol, the identifiers of the application descriptors available in the local repository are also systematically included in the catalog. Thus, each deployment manager systematically proposes to provide its neighbors with the descriptors of applications they may have never heard about. This approach allows application descriptors to disseminate rapidly in the network by being passed from deployment manager to deployment manager, while packages are only passed selectively to the deployment managers that need them. As the size of a package can be relatively large, this behavior is much more frugal in terms of network load than the one based only on package transmissions.

Note that a deployment manager that discovers that it owns one or several packages a neighbor is looking for does not send this package immediately to this neighbor. Indeed, when a deployment manager broadcasts an announcement indicating that it is looking for a particular package, several of its neighbors might be able to provide this package. In order to prevent that these neighbors all send the same package simultaneously, the protocol we designed is defined in such a way that they can only *offer* to provide this package. The actual transmission of a package or application descriptor is only performed on demand.

Thus, upon receiving a catalog each deployment manager matches the identifiers it contains against its current "shopping lists" in order to identify packages that match at least one of these lists, that is, packages it is interested in. Additionally, it also parses the catalog in order to identify application descriptors that are not already available in its repository. If such elements (i.e. software packages or application descriptors) are identified, then a request for these elements is sent to the announcer, which complies by broadcasting the required elements. Again, note that a deployment manager may sometimes discover that an element it is missing can be provided by several of its neighbors. In that case this deployment manager is only allowed to request the desired element from one neighbor. If that neighbor fails to provide the element, then the requestor will get another chance of obtaining this element from another neighbor after the next round of periodic announcements. Note also that when a deployment manager is requested to provide an element, this element is broadcast on the radio channel, rather than being sent only to the requestor in unicast mode. Thus, when several neighbors of the sender are interested in the same element, they can all be satisfied by a single broadcast, rather than by a sequence of unicast transmissions.

Finally, when a deployment manager receives an element it has requested, this element is put in the local repository so it can later be proposed to other deployment managers encountered while moving in the network.

Major steps of an application's deployment

Learning about new applications

At any time the deployment manager running on a mobile device maintains in the local repository a collection of application descriptors. Some of these descriptors correspond to applications that are not installable yet, meaning that some of the packages required for assembling these applications are not available locally. The deployment manager can thus be "aware of" the existence of an application, even though this application is not installed locally.

As explained in the former section, a deployment manager learns about new applications by continuously collecting application descriptors from neighbor hosts, while providing its neighbors with the descriptors it already maintains in its repository. Neighboring deployment managers therefore continuously inform each other about the applications they are aware of.

Initiation of a new application deployment

In order to initiate the deployment of a new application on a mobile device, the user can rely on the interface of the deployment manager, and select with this interface an application that is not installed yet. This scenario however implies that the local deployment manager must already be "aware" of the existence of this application.

Alternatively the user who knows about an application the deployment manager itself has never heard about can specify explicitly the

name of this application, in which case the first task of the deployment manager will be to look for this application's descriptor in the neighborhood.

Identification of missing packages

Once the descriptor of the desired application is available, the deployment manager can examine the dependencies described in this descriptor in order to determine what other packages are needed for assembling this application.

Remember that several applications may be assembled out of the same set of components. The packages needed to assemble a new application may thus be already available locally, as they may have been collected before in order to assemble and start another application. In fact, when determining what packages are needed for assembling an application the deployment manager may actually discover that all these packages are already present in the local repository. In such a case the deployment of the application can be considered as complete.

In most cases, though, when the user asks for the deployment of a new application the deployment manager is likely to discover that a number of required packages are missing in the local repository. As mentioned before each application whose deployment is in progress is associated a list of missing packages (the so-called "shopping list). Whenever packages required to deploy a given application are identified as missing packages, their identity is appended to the corresponding "shopping list". The protocol described in the former section ensures that packages identified in a host's "shopping lists" are collected opportunistically as this host meets other hosts while moving in the network.

Processing newly received packages

Whenever the deployment manager receives an element (package or application descriptor) it has requested from a neighbor host, this element is stored in the local repository. Besides, if this element is indeed a software package, then its name is removed from the local "shopping lists". In that case the descriptor of the package must also be analyzed in order to determine if this package depends on other packages that are not available locally. If so, then these packages must also be considered as requested packages, and their names be appended to the deployment manager's "shopping lists".

Completion of an application's deployment

The deployment of an application is complete when the corresponding "shopping list" is empty, which means that all the packages required for assembling this application have been collected and are now available in the local repository. The application can then be considered as installable, and be presented as such to the user through the user interface.

The user can also decide to cancel the deployment of a particular application at any time. In that case the "shopping list" maintained by the deployment manager for this application is discarded, and the packages that have already been collected and stored in the local repository are marked as unused (unless they are indeed used by another locally installed application, and unless their names ap-

pear in another local "shopping list"). Since mobile devices are usually resource-limited, the capacity of the repository is limited. Unused packages can however be maintained by the deployment manager in the local repository as long as there remains enough space to receive and store other desired packages. Otherwise the deployment manager is entitled to remove unused packages whenever there is a need to free storage space in the repository.

EVALUATION

The model we propose for cooperative software deployment on mobile devices is inherently a probabilistic one. Indeed, when a user requests that a given application be deployed on a mobile device, there is no absolute guarantee that the deployment manager on this device will ever manage to collect the required packages. It is worth mentioning that this lack of guarantee is a consequence of the characteristics inherent to disconnected MANETs, rather than a limitation of the model itself. However the model can be adapted in order to account for these constraints. For example, in order to increase the chance that the requests of the user can be satisfied, the deployment manager in the CODEWAN platform was designed so as to exhibit a persistent behavior. Whenever it cannot obtain a number of packages from its current set of neighbors, the deployment manager simply persists and tries to obtain these packages later, after its neighborhood has changed. Device mobility and volatility thus become advantages in this process, as the neighborhood of a device is not limited to a fixed set of neighbors.

It is obviously interesting to evaluate how our model for opportunistic software deployment can perform in realistic conditions. Indeed, the CODEWAN platform has been fully implemented in Java, and it can run on any kind of mobile device featuring a Wi-Fi interface. Small-scale experiments have been conducted using up to a dozen of mobiles devices (laptops and PDAs). However the scalability of our model for software deployment can hardly be confirmed using only a few mobile devices. CODEWAN was therefore implemented in such a way that it can also interface with the MADHOC simulator, which makes it possible to simulate mobile ad hoc networks involving hundreds of mobile devices [Hogie et al., 2006]. Based on this combination we run a number of simulations in order to observe how software deployment relying on opportunistic contacts between mobile devices can perform in medium to large-scale scenarios. In this section we present some of the results we obtained.

Simulation parameters

We consider a simulation scenario in which a population of 120 users move in and between a set of four buildings, as shown in Figure 1. These buildings are located within a 120 m $\times$ 90 m area. Each user is assumed to carry a laptop running CODEWAN, and equipped with an IEEE 802.11 (Wi-Fi) interface.

The mobility of users—and therefore that of the mobile hosts they are carrying—is simulated using a variant of the random waypoint model: a user can remain motionless for a while, afterwards he/she begins to walk towards a set destination, which is selected randomly in any one of the buildings in the simu-

lation area.

In the simulation runs whose results are discussed below, we used the following mobility parameters: users are assumed to walk at speeds varying between 0.5 m/s and 2 m/s; a stay between two consecutive moves can last between 30 seconds and 3 minutes; and the amount of intra-building mobility is set to 40 % (against 60 % for inter-building mobility). Wi-Fi interfaces are assumed to have an omni-directional transmission range of 20 meters when used indoor, and 60 meters when used outdoor.

The gossiping protocol that allows neighbor deployment managers to get aware about each other and exchange descriptors and components can be adjusted by setting two parameters. The first parameter is the period with which the CODEWAN platform broadcasts an announcement. The second parameter is the maximum number of hops used in temporaneous message forwarding, which allows a mobile host to use multi-hop transmissions to reach other hosts that belong to the same connectivity island as itself. In the scenario whose results are presented below the period was set to 15 seconds, and the number of hops was set to 5 hops.

Software deployment scenario

In this scenario we focus on the deployment of a single application whose installation requires four software components. At the beginning of the simulation run, only one laptop owns a copy of the descriptor of this application. Besides, two copies of each component required for assembling this application are already stored in the local repositories of other laptops (although each of these laptops only owns one of these components).

Our first objective is to observe how long it takes for the descriptor of the application to disseminate in the whole network. In other words, we wish to evaluate the time required for the deployment manager running on each laptop to learn about a new application. Second, once all laptops have learned about the existence of the application we will assume that one or several users carrying these laptops decide to install this application locally. We will then observe how long it takes for the deployment managers running on these users' laptops to collect the required components while moving in the network.

Simulation results

Figure 8 shows the time it takes for the descriptor of a new application to disseminate network-wide. This figure was obtained by running 200 simulation runs with different seeds in the mobility algorithm, and different placements of the original application's descriptor. We measured the time required for this descriptor to reach each mobile host involved in the simulation, and calculated the average curve based on all measured values (hence the smooth aspect of the curve in Fig. 8). It can be observed that information about a new application can disseminate quite rapidly in a scenario such as that considered in this example. Indeed, once the descriptor of a new application appears somewhere in the network, about 90 % of the deployment managers running on mobile hosts are informed about this new application in less than 10 minutes, and 100 % deployment managers are aware of this application after 20 minutes. Of course these figures would be different with

a smaller or greater number of mobile devices involved in the network, or if these devices moved slower or faster, as this would have an impact on the time required for a descriptor to be carried between two buildings, for example. In any case this figure confirms that a number of laptops carried by users moving in a campus-like environment can collaborate to rapidly disseminate new information regarding a component-based application.

Once a deployment manager has learned about a new application by receiving its descriptor, information about this application can be presented to the user, which in turn can order the deployment manager to install this application locally. Let us consider again the application whose descriptor has just been disseminated in the network, and assume that a number of users decide that they would like to use this application on their laptop. Figure 9 shows the average time it takes for a deployment manager, once it has been ordered to install the application locally, to collect the four components that constitute this application. Remember that at the beginning of the simulation we assume that these components are already stored in some laptops' local repositories, and that there actually already exists two copies of each component in the network. In Fig. 9 it can be observed that when a single user is interested in the application considered (meaning the deployment manager running on its laptop must meet successively at least one laptop carrying each of the four required components), this user has to wait about 3 hours until the installation is complete and he/she can start using the application. However, when four users are interested in the same application and decide to install it on their laptop, the whole installation process takes about 70 minutes for each user. This is because whenever a deployment manager obtains a component it is missing, it becomes a carrier for this component and can therefore help in the dissemination of this component. Of course the installation process of the application considered in this scenario would be even faster if a larger number of users —and possibly all of them— decided to install this application on their laptop. Indeed, with this opportunistic deployment process relying on peer-to-peer interactions between deployment managers, the greater the popularity of a software component, the easier it is for a deployment manager to find a copy of this component, and therefore the faster the installation of applications relying on this component.

RELATED WORK

A number of protocols have been designed over the last decade in order to support communications in disconnected or partially connected MANETs. Good surveys of works conducted along this line can notably be found in [Zhang, 2006] and [Pelusi et al., 2006]. Indeed, most of these works actually aim at supporting destination-driven message forwarding, using either context-based [Leontiadis and Mascolo, 2007, Musolesi and Mascolo, 2009, Musolesi, 2004], geographic-based [Leontiadis and Mascolo, 2007], probabilistic [Lindgren et al., 2004] or history-based [Boldrini et al., 2007] heuristics to improve message delivery while reducing the network load. They therefore mainly focus on communication issues, and do not especially consider the problems inherent to the

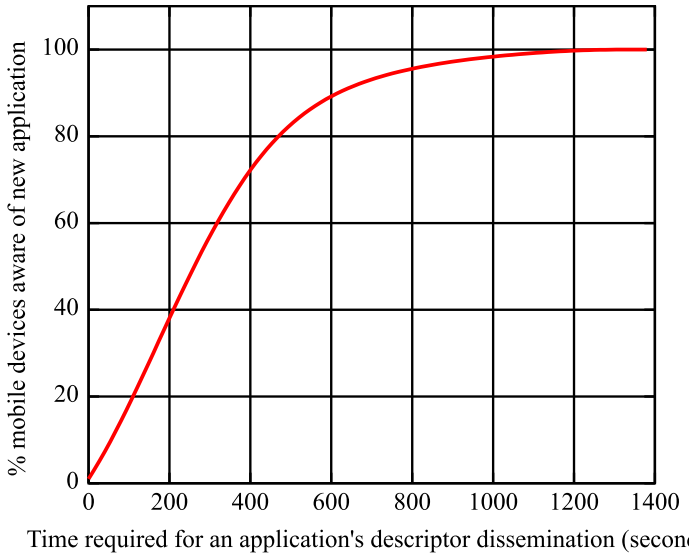


Figure 8: Average time required for a new application's descriptor to disseminate network-wide.

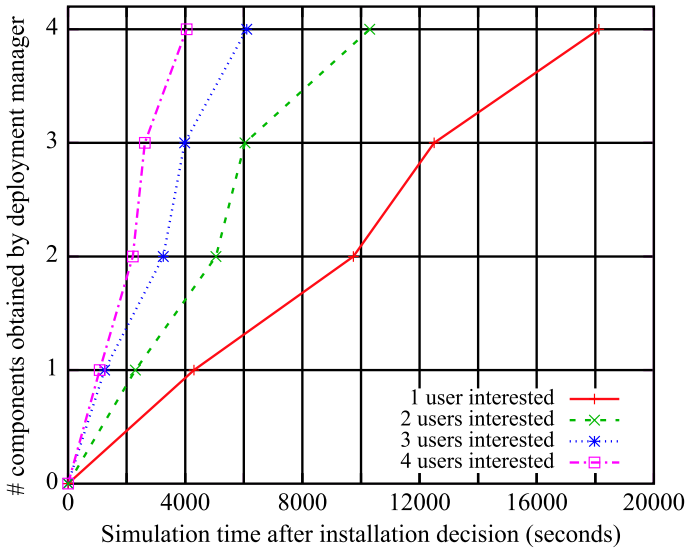


Figure 9: Time required for a deployment manager to collect the four components required for application installation depending on the number of users (and deployment managers) interested by these components

deployment of component-based applications in disconnected MANETs.

These specific problems have however been addressed in many projects and following different lines in the past few years. For example Java Web Start [Sun Microsystems, 2004] supports the deployment and the update of Java-based application programs. It relies on a client-server model: a server (or a collection of servers in the case of Maven) maintains a repository where applications can be stored, and clients can download new applications—or new versions of applications they have already downloaded—from this server. We believe that the client-server model is hardly applicable for deploying and updating software in an autonomous ad hoc network, although it usually performs satisfactorily in an infrastructure network. The approach we propose thus aims at achieving software deployment based on opportunistic peer-to-peer interactions between mobile devices, as we think that this approach is better suited to ad hoc networks.

Cooperative (or peer-to-peer) approaches to software deployment have been proposed in works related to SoftwareDock [Hall et al., 1999], [Gopalan and Znati, 2005] and OSGi [Frénot and Royon, 2005]. SoftwareDock is a framework for distributed software deployment that uses mobile agents to support the transfer of software applications. In [Frénot and Royon, 2005] it is proposed to organize nodes that participate in the deployment of software components as a hashkey-based peer-to-peer network. Both approaches are primarily meant to be applied in infrastructure networks, though, as the prime motivation is to allow load balancing and fault tolerance among the devices that provide the components. [Gopalan and Znati, 2005] proposes a novel peer-to-peer based approach for the deployment of applications and services in MANETs called POST. In POST, mobile nodes do not maintain routing information. Mobile devices are expected to register their mobility profile (their expected direction and speed of travel) and the objects they query and provide in the manager of the zone they are located in. The environment is organized in several zones identified respectively by a hashkey value. Distributed hashtables are used to store and to retrieve data information. The hash identifiers are used by the routing protocol while downloading or deploying applications in order to limit network flooding.

Closer to our work, SATIN proposes to deploy component-based, self-organized systems on mobile devices [ZACHARIADIS ET AL., 2004]. It supports the storage and the execution of components on a device, as well as component advertisement, discovery and transfer between distinct devices. Little is said in [ZACHARIADIS ET AL., 2004] about the patterns of interactions between these devices, though, or about the role—if any—of the devices' owners. The example described in the paper actually suggests that the SATIN middleware adopts a greedy behavior, and that each mobile device tries to collect any component it can discover in the network. In contrast the CODEWAN platform is more "user-oriented", as the owner of a device (or the administrator of a collection of devices) is expected to specify what applications—and consequently what components—should be deployed locally. Another major difference between SATIN and CODEWAN is that SATIN implements its own component model, while

CODEWAN is rather meant to support the dissemination of any kind of components (provided they can be stored and transported in packages) in an ad hoc network.

CONCLUSION AND FUTURE WORK

In this paper we presented the CODEWAN platform, which is dedicated to the deployment of component-based software applications on mobile devices participating in a disconnected MANET. CODEWAN implements a peer-to-peer, cooperative model for software deployment. With this model, each mobile device maintains a local repository that can accommodate a number of software components. The components stored in this repository are available for the execution layer of the platform. Neighboring devices exchange copies of the software components they own based on an opportunistic interaction scheme. A communication layer implements the underlying necessary unicast and broadcast primitives that are the basis of the interactions with neighboring devices. This implementation provides an efficient multi-hop neighborhood.

The CODEWAN platform was implemented in Java and is now fully operational. Simulation runs show that the overall performance of the platform is satisfactory. Our future work mainly aims at augmenting the platform's functionality, regarding namely the adaptiveness of the platform and its security.

Indeed, in the current implementation of the CODEWAN platform, the deployment manager running on a mobile device must be configured manually by the user of this device. For example, the user is responsible for setting the appropriate periods for broadcast announcements, or the number of hops defining the neighborhood. The user must likewise determine how much storage space must be assigned to the local repository. Future work will notably focus on the development of a strategy manager capable of adjusting the behavior of a deployment manager transparently and continuously on behalf of the user. For example the periods for announcing local packages and requesting new packages could be adjusted dynamically based on the mobility of a device, on observations of its neighborhood, or on internal events (such as the local device being suspended or resumed).

The approach we propose for deploying software applications on mobile devices relies on the assumption that the owners of these devices may find it convenient to share software components with each other using ad hoc communication. This approach obviously raises a number of legitimate concerns regarding security, as the owner of a mobile device may for example be reluctant to run on this device pieces of software obtained from unidentified sources. We believe that this problem may be solved satisfactorily by using digital signatures so as to ascertain the origin of a software component, as well as ciphering in order to limit the use of a given component to a particular community of users.

Acknowledgement: This work is supported by the French Agence Nationale de la Recherche under contract ANR-05-SSIA-0002-01.

References

[Boldrini et al., 2007] Boldrini, C., Conti, M., Iacopini, I., and Passarella, A. (2007). HiBOp: a History Based Routing Protocol for Opportunistic Networks. In *International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–12. IEEE CS Press.

[Bruneton et al., 2004] Bruneton, É., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2004). An Open Component Model and Its Support in Java. In *International Symposium on Component-Based Software Engineering*, pages 7–22. Springer.

[Carzaniga et al., 1998] Carzaniga, A., Fuggetta, A., Hall, R. S., Heimbigner, D., van der Hoek, A., and Wolf, A. L. (1998). A Characterization Framework for Software Deployment Technologies. Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado.

[Clausen and Jacquet, 2003] Clausen, T. and Jacquet, P. (2003). Optimized Link-State Routing Protocol (OLSR). IETF, RFC 3626.

[Datta et al., 2004] Datta, A., Quarteroni, S., and Aberer, K. (2004). Autonomous Gossiping: a Self-Organizing Epidemic Algorithm for Selective Information Dissemination in Mobile Ad-Hoc Networks. In *International Conference on Semantics of a Networked World*, number 3226 in LNCS, pages 126–143.

[Frénot and Royon, 2005] Frénot, S. and Royon, Y. (2005). Component Deployment Using a Peer-To-Peer Overlay. In *Working Conference on Component Deployment*, volume 3798 of *LNCS*, pages 32–35. Springer.

[Gopalan and Znati, 2005] Gopalan, A. and Znati, T. (2005). POST: A Peer-to-Peer Overlay Structure for Service and Application Deployment in MANETs. In *International Conference on Mobile Ad-hoc and Sensor Networks*, volume 3794 of *LNCS*, pages 1006–1015. Springer.

[Haillot and Guidec, 2008] Haillot, J. and Guidec, F. (2008). A Protocol for Content-Based Communication in Disconnected Mobile Ad Hoc Networks. In *International Conference on Advanced Information Networking and Applications*, pages 188–195. IEEE CS Press.

[Hall et al., 1999] Hall, R. S., Heimbigner, D., and Wolf, A. L. (1999). A cooperative approach to support software deployment using the software dock. In *International Conference on Software Engineering*, pages 174–183.

[Hoareau and Mahéo, 2008] Hoareau, D. and Mahéo, Y. (2008). Middleware Support for Ubiquitous Software Components. *Personal and Ubiquitous Computing (PUC)*, 12(2):167–178.

[Hogie et al., 2006] Hogie, L., Guinand, F., Danoy, G., Bouvry, P., and Alba, E. (2006). Simulating Realistic Mobility Models for Large Heterogeneous MANETs. In *International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 2–6. ACM Press.

[Le Sommer and Guidec, 2002] Le Sommer, N. and Guidec, F. (2002). JAMUS: Java Accommodation of Mobile Untrusted Software. In *Nord EurOpen/Usenix Conference*, pages 38–48. Multiprint.

[Leontiadis and Mascolo, 2007] Leontiadis, I. and Mascolo, C. (2007). GeOpps: Geographical Opportunistic Routing for Vehicular Networks. In *Workshop on Autonomic and Opportunistic Communications*, pages 1–6. IEEE CS Press.

[Lestideau et al., 2002] Lestideau, V., Belkhatir, N., and Cunin, P.-Y. (2002). Towards automated software component configuration and deployment. In *Process Support for Distributed Team-based Software Development Workshop*.

[Lindgren et al., 2004] Lindgren, A., Doria, A., and Schelen, O. (2004). Probabilistic Routing in Intermittently Connected Networks. In *International Workshop on Service Assurance with Partial and Intermittent Resources*, volume 3126 of *LNCS*, pages 239–254. Springer.

[Muchow, 2002] Muchow, J. (2002). *Core J2ME Technology*. Prentice Hall.

[Musolesi, 2004] Musolesi, M. (2004). Designing a Context-Aware Middleware for Asynchronous Communication in Mobile Ad Hoc Environments. In *Middleware Doctoral Symposium*, pages 304–308. ACM Press.

[Musolesi and Mascolo, 2009] Musolesi, M. and Mascolo, C. (2009). CAR: Context-Aware Adaptive Routing for Delay Tolerant Mobile Networks. *IEEE Transactions on Mobile Computing*, 8(2):246–260.

[OMG, 2002] OMG (2002). Corba components, version 3.0.

[Parrend and Frénot, 2007] Parrend, P. and Frénot, S. (2007). Supporting the Secure Deployment of OSGi Bundles. In *Workshop on Adaptive and DependAble Mission- and bUsiness-critical mobile Systems*.

[Pelusi et al., 2006] Pelusi, L., Passarella, A., and Conti, M. (2006). Opportunistic Networking: Data Forwarding in Disconnected Mobile Ad Hoc Networks. *IEEE Communications Magazine*, 4(11):134–141.

[Perkins, 2001] Perkins, C. (2001). *Ad Hoc Networking*. Addison-Wesley.

[Qayyum et al., 2002] Qayyum, A., Viennot, L., and Laouiti, A. (2002). Multipoint Relaying for Flooding Broadcast Messages in Mobile Wireless Networks. In *35th Annual Hawaii International Conference on System Sciences*, page 298. IEEE CS Press.

[Sun Microsystems, 2004] Sun Microsystems (2004). Java Web Start 1.5.0 Documentation.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

[Vahdat and Becker, 2000] Vahdat, A. and Becker, D. (2000). Epidemic Routing for Partially-Connected Ad Hoc Networks. Technical Report CS-2000-06, UCSD.

[Zachariadis et al., 2004] Zachariadis, S., Mascolo, C., and Emmerich, W. (2004).

SATIN: A Component Model for Mobile Self Organisation. In *International conference on Distributed Objetcs and Applications*, volume 3291 of *LNCS*, pages 1303–1321. Springer.

[Zhang, 2006] Zhang, Z. (2006). Routing in Intermittently Connected Mobile Ad Hoc Networks and Delay Tolerant Networks: Overview and Challenges. *IEEE Communications Surveys and Tutorials*, 8(1):24–37.

[Ziv and Lempel, 1978] Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536.