# FRANC: A Lightweight Java Framework for Wireless Multihop Communication[*]

David Cavin Yoav Sasson André Schiper
{yoav.sasson,david.cavin,andre.schiper}@epfl.ch

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

Technical Report IC/2003/22

**Abstract**

Simulation and emulation are popular means for evaluating wireless Mobile Ad hoc Networks (MANETs) protocols. Since MANETs are highly dependant on their physical environment, these techniques offer only a partial understanding of factors that may influence performance. Deploying real-life MANETs is therefore an indispensable complementary step for the advancement of MANETs. This paper presents FRANC [1], a dedicated extensible Java framework for the development, deployment and evaluation of applications and algorithms for wireless mobile ad hoc networks.

## 1 Introduction

Mobile ad hoc networks (MANETs) are self-organizing mobile wireless networks that do not rely on a preexisting infrastructure to communicate. Nodes of such networks have limited transmission range, and packets may need to traverse multiple nodes before reaching their destination. Research in MANETs was initiated 30 years ago by DARPA for packet radio projects [1], but has regained popularity nowadays due to the widespread availability of portable wireless devices such as cell phones, PDAs and WiFi / Bluetooth enabled laptops.

The design, development, deployment and analysis of algorithms and applications for MANETs are complex tasks. Despite the considerable amount of research devoted to the MAC and routing levels, very little effort has been dedicated to the engineering of applications. Indeed, current MANET evaluation tools such as simulators, emulators and testbeds are mainly concerned with the lower layer of the communications stack. Real-life implementation of applications is a necessary step to fully comprehend their behavior in the complex environment of MANETs.

Software development frameworks facilitate the development of a specific category of applications by offering a context, supporting a design architecture and providing a set of tools that are likely to be used. Examples of recurrent primitives used by MANET applications and algorithms are neighbor discovery and multihop routing. A dedicated MANET application development framework would therefore not only reduce development time but would further offer a more structured implementation facilitating program verification and statistics collection.

Desired requirements for a MANET development framework are for it to run on heterogeneous devices (type, OS), offer easy configuration, statistics gathering mechanism for performance evaluation, simplicity, modularity and extensibility. The framework must also be lightweight in the sense that it has to be adapted to the limited computing, energy and bandwidth resources of devices composing MANETs.

[*]The work presented in this paper was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant 5005-67322.

[1]FRamework for Ad hoc Network Communication

Although there exists numerous application development frameworks for wired networks, very little has been done for MANETs. In this paper we present *FRANC*, the first modular and configurable framework for the development and evaluation of Java wireless multihop applications, meeting the aforementioned requirements.

The remainder of the paper is organized as follows. The next section presents the most important features of FRANC. In Section 3 we present the different applications and algorithms that have been developed with FRANC. Section 4 provides an overview of the various tools for the evaluation of MANET protocols as well as an overview of distributed application development frameworks in general. We finally conclude and describe future work in Section 5.

## 2 Feature Tour

### 2.1 Requirements

The initial motivations for the framework were to complement simulations for evaluating mobile ad-hoc algorithms with a set of recurrent building blocks. Moreover *FRANC* has been developed with the concern of meeting several requirements specific to mobile Ad-hoc networks. Firstly, *FRANC* must run on heterogeneous devices with various computing, energy and bandwidth resources and operating systems. The framework must also provide a mean to emulate seamlessly a multihop wireless communication for the needs of a small scale demonstration. It must be easily extensible with new protocols, middleware algorithms (eg. distributed algorithms) and higher-level applications. The configuration of the protocol stack and the setting up of all parameters must be easy to express.

### 2.2 Architecture

*FRANC* is composed of three different types of objects : the protocol stack, the services repository and the message pool (Figure 1).

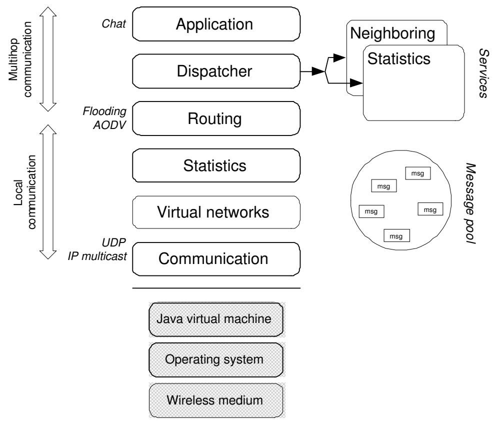


Figure 1: General Architecture

#### 2.2.1 The protocol stack

It is dynamically built at startup according to the layers' description read from an XML configuration file. We have currently implemented and successfully tested six layers. At the bottom, the *communication layer* relies on UDP/IP multicast and provides a mean for the layer above to send a broadcast message within the node's transmission range. Above there is the *virtual network* layer that emulates a multihop environment by filtering and discarding received messages (cf. Section 2.3). This feature allows to create a multihop topology even if nodes can actually physically (at the MAC layer) exchange messages. The next layer can collect various kind of statistics such as the amount of bytes or the number of messages transmitted or received. It regularly publishes the statistics collected to the *Statistic service* (cf. Section 2.2.2). Then above this layer, we have implemented two routing algorithms, a *probabilistic flooding algorithm* [2] and the *Ad-hoc On Demand Distance Vector* (AODV) [3]. From this point in the protocol stack the communication becomes multihop. Then there is the *Dispatcher* that connect the *services* to the protocol stack. It is described in more details in Section 2.2.2. Finally, on the top of the stack, we have implemented two demo applications: a *Chat* and a token based topology discovery algorithm (cf. Section 3). Both illustrate most of the features developed in the framework.

All layers must conform to an interface called `AsynchronousLayer` that exports, in particular, a `send` and an asynchronous `receive` method (cf. Figure 2). This means that the message passing mechanism between layers supposes a shared data structure (a buffer) to forward up the message asynchronously in the protocol stack. On the contrary, the `send` method is synchronous in the sense that when it returns the message is actually sent.

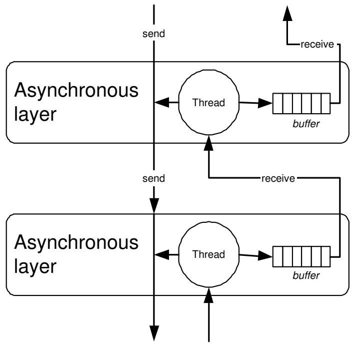


Figure 2: Asynchronous Delivery

Each `AsynchronousLayer` contains a thread and a buffer. The thread executes an infinite `while` loop (cf. Algorithm 1) that tries to read any incoming message from the buffer located in the layer below. Each time a new message is received, the layer checks, according to the type of the message, if it has to handle it. If not, the message is stored in the buffer that will be consumed by the above layer's thread. A newly received message thus traverses the protocol stack until it reaches a layer that handles it.

**Algorithm 1** : Asynchronous layers' pseudo-code

```
AsynchronousLayer(lowerLayer)
void initialize() ...
// Thread infinite loop
while true do
    Msg m = lowerLayer.buffer.read();
    switch (m.type)
        case myMSG :
            handle(m);
        default :
            buffer.add(m);
    end switch
end while
```

### 2.2.2 The services repository

The protocol stack presented in the previous section respects a strict hierarchical interaction which means that each layer has only a reference to the layer that is just below. But in some cases, several layers may need a same information (eg. neighbors) that could be maintained besides the stack and kept available for all layers interested in. These general purpose informations are called *services* and can be accessed through the *Dispatcher* layer. The *Dispatcher* acts as a message router and a service repository that takes messages out of the stack and forwards them to a particular service. Practically, whenever a layer needs an information hold by a service it asks the *Dispatcher* for a reference to the service. We have currently implemented two *services* : the neighboring service and the statistic service. The first one, by broadcasting regularly hello packets, maintains the actual list of surrounding nodes. The second one stores the

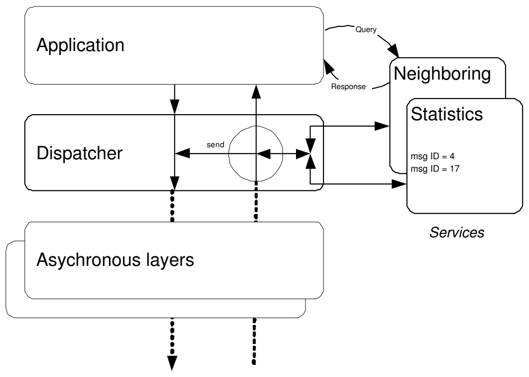


Figure 3: The Dispatcher Layer

statistics collected by the statistics layer. This latter service provides hints about the network state and can be exploited to adapt a routing policy, for example.

#### 2.2.3 The message pool

With Java, and especially with small virtual machines, the creation and the destruction (ie. garbage collection) of objects is costly in terms of time and resources. We have thus implemented a centralized shared object that exports all operations related to message handling and stores the different message factories and types. A locking mechanism also allows to reuse existing references to old messages safely preventing the simultaneous access to a same reference until the message is explicitly freed.

### 2.3 Virtual Networks

It can be rather tedious to demonstrate the correct execution of a multihop wireless application in a real environment (typically a room) because the transmission ranges are often greater than several tenth or hundreds of meters. We have thus implemented in *FRANC* a filtering layer that can emulate a virtual multihop environment. At startup, each node is configured to belong to one or several *virtual networks* and can thus only receive messages sent from the same *virtual networks*. The feature allows to demonstrate, at small scale, the execution of a multihop application. Figure 4 shows an example of three virtual networks (A, B and C). It is important to notice that the emulated multihop environment is implemented at the application level and does not affect the MAC layer. More precisely, this mechanism does not prevent messages sent by logically out of range nodes (ie. belonging to different *virtual networks*) to collide.

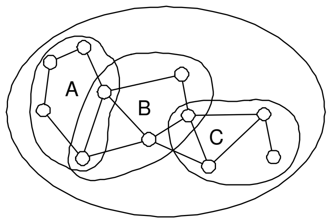


Figure 4: Virtual Networks

### 2.4 Serialization

The serialization of Java messages is an important issue with respect to performance and bandwidth usage. In order to prevent the systematic use of the costly standard Java object serialization, *FRANC* leaves the responsibility of serialization to developers that implement their own message types. Each message object that is sent through the network, must implement a `setByte(byte datas[])` and a `byte[] getBytes()` method. The developer decides what information in the message have to be serialized for a particular message type. It is not necessary to transmit everything (some fields are useless at the other side or can be deducted). Moreover, *FRANC* provides several optimized serialization methods that can transform all Java primitive types into bytes. This feature encourages the developers to encode complex data structures using simple Java types (eg. int, String, float, double, etc.) and thus reduces the amount of bytes transmitted over the network.

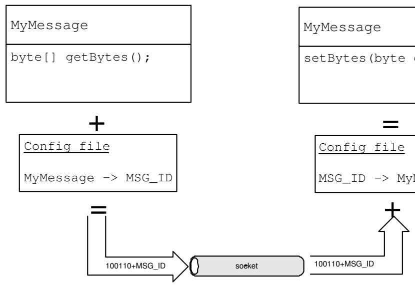


Figure 5: *FRANC* Serialization

Figure 5 depicts the serialization mechanism of *FRANC*. When a message is ready to be sent, the communication layer generates a header, with the help of the configuration file (cf. Section 2.6), that contains in particular the message ID (an integer that maps to a given message class), the source and destination nodes and a sequence number (unique for a given node). The header is then concatenated with the bytes returned by the `getBytes()` method of the message. At the other end, when the message is received, the destination node first reads and decode the headers. According to the message ID, and again with the help of the configuration file, it constructs the corresponding message (initially empty). Then the communication layer calls the newly allocated message's method `setBytes` with the remaining bytes.

This lightweight serialization procedure has several advantages. First, it gives to the developers more control over the binary representation of messages without adding more complexity. It is also more flexible because it allows two nodes to have different implementations of a same message by mapping different class names to the same message ID.

## 2.5 Statistics Gathering

Statistics gathering has two proposes. First, the dynamic collection of basic network metrics at runtime, can be used as hints for other layers (eg. routing algorithms or services). This is achieved with the help of the *statistics service* which can be accessed asynchronously by different layers. Secondly, more complex statistics can be gathered (or logged) for evaluating the performances of the framework itself or of the wireless medium (eg. throughput, latency, RTT)

## 2.6 Startup and Configuration

The startup procedure of *FRANC* goes through four different phases. First, the XML configuration file, given as a parameter, is parsed again the framework *Document Type Definition* (DTD). The DTD is a kind of grammar (rules) that defines constraints for XML document (tag names, order, etc.). The parsing step is made by Java, which rejects configurations that do not match any of the rules stated in the DTD. An sample XML configuration file is presented in Figure 6. The configuration file is made of two different sections. The first section, called `global` defines all the parameters necessary to start the framework. More precisely, the layers and services class names and exported parameters, the message types, IDs and the corresponding factories. The second section is optional and defines one or several sub-configurations that inherit all global parameters except those locally overridden. The example presented in Figure 6 launches a "faulty" node by replacing the global communication layer by a faulty communication layer, that loses or corrupts messages, for example. This mechanism is very convenient because it is possible to start different types of nodes (eg. faulty or router nodes) without having to change the XML configuration file.

After the parsing step, the second step of the startup procedure is the creation of all the layers and services. The layers are connected according to the order specified in the configuration file. Then the `initialize` method of each layer is invoked with the parameters given (for each layer) in the XML file. The last step is the invocation of the `startup` method from the lowest layer to the highest which actually starts the framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE request PUBLIC
  "-//DTD MANET Configuration 1.0//EN" "FrameworkConfig.dtd">
<global>
  <layers>
    <layer class="communicationslayers.AsynMulticast">
      <param name="port" value="20177"/>
    </layer>
    .
    <!-- Additional layers description -->
    .
  </layers>
</global>
<subconfiguration name="faulty">
  <layers>
    <layer class="communicationslayers.FAULTY_AsynMulticast">
      <param name="port" value="20177"/>
    </layer>
  </layers>
</subconfiguration>
```

Figure 6: XML sample file

## 3 Applications

This section presents two applications and a broadcast algorithm that have been developed with FRANC.

### 3.1 Chat

A multihop chat application similar to the wide-spread instant messaging programs is the first application developed using FRANC. It serves well as an illustration of a multihop application as well as for debugging purposes. Figure 7 presents a screenshot of the chat application in action. Besides the chat itself (bottom window) are windows displaying the framework's configuration and status information, neighboring nodes and message statistics.

### 3.2 Distributed Token Passing

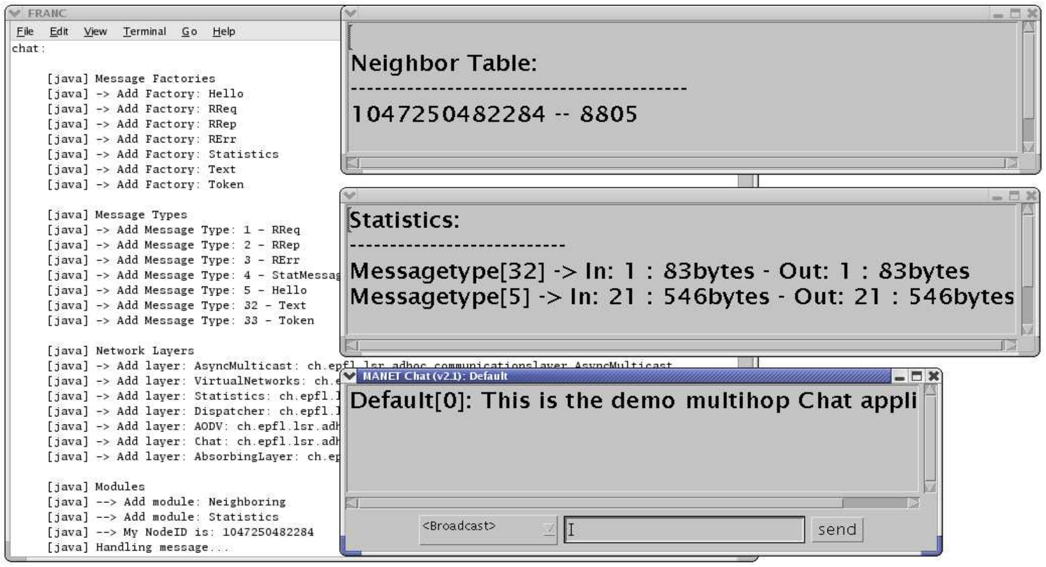
Tokens travelling from node to node, harvesting, disseminating and sharing information, gradually converging toward a desired goal is a promising communication means and policy enforcement technique adapted to the high dynamics of MANETs.

We have implemented the *LR* and *LF* distributed token passing algorithms defined in [4]. In *LR*, the next recipient of the token is chosen among the neighboring nodes based on how *recently* the nodes have had the token. For *LF*, the decision is taken based on how *frequently* the nodes have had the token.

The two token passing algorithms have been implemented as a service (cf. Section 2.2.2), since they may potentially be used by more than one layer in the stack. Indeed, token passing may be used by the routing layer for route discovery and message routing, while applications may exploit tokens to disseminate and gather various information.

### 3.3 Reliable Broadcast

The basic communication primitive of the framework is an unreliable one-hop UDP broadcast. Coupled with the fact that the 802.11b MAC in DCF mode does not enforce RTS/CTS/ACK, a significant number of packets may be lost due to packet collisions. We have implemented a first solution to the problem by developping a `reliableLayer` layer requiring an ACK to be sent by all neighbors of a one-hop broadcast. By placing `reliableLayer` directly above the Virtual Networks layer (cf. Fig. 1), reliability of all overlying layers (routing in particular) will automatically see their reliability enhanced. The reason for not placing `reliableLayer` beneath the Virtual Networks layer is because we do not not want nodes to acknowledge for messages from other virtual networks. The `reliableLayer` may be activated or deactivated through FRANC's XML configuration file (cf. Section 2.6).



```
chat:

     [java] Message Factories
     [java] -> Add Factory: Hello
     [java] -> Add Factory: RReq
     [java] -> Add Factory: RRep
     [java] -> Add Factory: RErr
     [java] -> Add Factory: Statistics
     [java] -> Add Factory: Text
     [java] -> Add Factory: Token

     [java] Message Types
     [java] -> Add Message Type: 1 - RReq
     [java] -> Add Message Type: 2 - RRep
     [java] -> Add Message Type: 3 - RErr
     [java] -> Add Message Type: 4 - StatMessag
     [java] -> Add Message Type: 5 - Hello
     [java] -> Add Message Type: 32 - Text
     [java] -> Add Message Type: 33 - Token

     [java] Network Layers
     [java] -> Add layer: AsyncMulticast: ch.epfl.lsr.adhoc.communicationslayer.AsyncMulticast
     [java] -> Add layer: VirtualNetworks: ch.e
     [java] -> Add layer: Statistics: ch.epfl.l
     [java] -> Add layer: Dispatcher: ch.epfl.l
     [java] -> Add layer: AODV: ch.epfl.lsr.adh
     [java] -> Add layer: Chat: ch.epfl.lsr.adh
     [java] -> Add layer: AbsorbingLayer: ch.ep

     [java] Modules
     [java] --> Add module: Neighboring
     [java] --> Add module: Statistics
     [java] --> My NodeID is: 1047250482284
     [java] Handling message...
```

```
Neighbor Table:
-----------------------------------------
1047250482284 -- 8805
```

```
Statistics:
--------------------------
Messagetype[32] -> In: 1 : 83bytes - Out: 1 : 83bytes
Messagetype[5] -> In: 21 : 546bytes - Out: 21 : 546bytes
```

```
Default[0]: This is the demo multihop Chat appli
```

Figure 7: Multihop Chat Application

## 4 Related Work

In this section we present the most popular tools used for the evaluation of MANET protocols as well as distributed application development frameworks in general.

### 4.1 Simulators and Emulator Testbeds

NS-2 is currently the most popular simulator within the MANET community, but suffers from poor scalability with respect to the number of nodes simulated. GloMoSim [5] is also a widespread MANET simulator, appreciated for its scalability by using Parsec's [6] parallel processing capabilities. Its development however has somewhat slowed down in favor of its Qualnet [7] commercial counterpart. Finally, OPNET Modeler [8] is a commercial simulator for which third-party MANET routing protocols [9, 10] are available.

Simulation in itself does not suffice for evaluating MANET protocols. Useful for initial study of protocols, MANET simulators are bound to make numerous assumptions due to the complexity of ad hoc networks resulting in inaccurate physical layer and environment models. Furthermore, different simulators often provide contradictory performance results for the same protocols [11].

Emulation offers more precise results than simulation since protocols are run over a live network. The most mature emulator for MANETs is the APE testbed [12]. APE comes as a small Linux distribution with integrated MANET routing protocols and performance analysis tools. APE has been tested to scale up to 30 nodes. Among other emulators are JEmu [13], EWANT [14] and EMWIN [15].

Emulating all aspects of MANET is however a more daunting effort than for wired networks, as issues such as mobility and network topology render experiments difficult and costly to reproduce.

Current MANET simulators and emulator testbeds concentrate on the lower layers of the communcation stack. Currently there is no environment for the development, deployment and evaluation of application-layer distributed algorithms and applications for MANETs. The goal of FRANC is to offer such a tool by concentrating on the application level.

### 4.2 Frameworks for Wired Networks

There exists many frameworks for the development of distributed applications within traditional wired networks. They all concentrate on implementation flexibility, but differ in their approach for application decomposition and interaction between the components. Among the frameworks similar in their intentions to FRANC are Appia [16], Cactus [17, 18] and Neko [19].

Appia [16] is a layered communication framework implemented in Java. Appia's architecture appears as a stack of layers communicating through events that may carry information or messages. An interesting characteristic of Appia is its single-threaded approach: events within the stack are all processed by one single execution thread.

In Cactus [17, 18], applications are decomposed into protocols (may be seen as modules) structured hierarchically in a stack. Protocols themselves may be composed of micro-protocols, which offer yet a finer grain decomposition of the tasks. Interaction between protocols are message-based, whereas interaction between the micro-protocols of a protocol are event-based. By default, Cactus adopts a thread-per-message approach. Java, C and C++ implementations of Cactus are available.

Neko [19] is a Java based framework offering a uniform environment for the development and performance evaluation of distributed algorithms. Applications in Neko are built as a hierarchy of layers. Layers communicate through message passing, with a thread-per-layer approach. The main characteristic of Neko is its ability to use the same algorithm implementation for simulation as well as execution within a real network.

FRANC on the other hand is a much smaller and lightweight framework specifically geared at wireless devices. The architecture of FRANC, described in Section 2.2, addresses the limitations of such devices and offers implementations of layers that provide tools for multihop wireless communication.

## 5 Summary and Future Work

In this paper we present FRANC, an extensible Java framework for the development, deployment and evaluation of applications and algorithms for wireless mobile ad hoc networks. Among the benefits of such a framework is accelerated MANET application development and facilitated algorithm verification and performance analysis. FRANC in its current state is fully functional and has been extensively tested. We are actively pursuing the development of FRANC. Development efforts are carried on two fronts, enhancing the framework itself as well as implementing MANET protocols and algorithms with help of the framework.

Regarding framework itself, we are currently working on a GUI that abstracts the complexity of editing directly the XML configuration files, as well as a set of tools to extract and analyze various performance metrics related to the framework. We would like to offer at a later stage simulation capabilities to FRANC. Simulation may be used as a first assessment of performance and reduces the difficulty of conducting numerous experiments comprising of a large number of mobile nodes. By implementing a wireless MAC layer such as 802.11b as well as simulated mobility patterns, it will be possible to evaluate the *same* implementation of algorithms simulated and real-life settings.

In terms of algorithms and applications, we are investigating several flavors of broadcast primitives offering different levels of quality of service. We may also consider evaluating typical P2P applications relevant in the context of MANETs.

We intend on making the source code of FRANC as well as its documentation available for download. Given sufficient interest, we will set up an open-source project around it.

## Acknowledgments

We would like to thank Javier Bonny and Urs Hunkeler for their contribution to the implementation of the framework, as well as Bertrand Grandgeorge and Reto Krummenacher for the development of modules and protocols.

## References

[1] J. Jubin and J. D. Tornow, "The DARPA packet radio network protocol," vol. 75, no. 1, pp. 21–32, Jan. 1987.

[2] Y. Sasson, D. Cavin, and A. Schiper, "Probabilistic broadcast for flooding in wireless mobile ad hoc networks," in *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2003)*, New Orleans, LA, Mar. 2003.

[3] C. E. Perkins, E. M. Royer, and S. R. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing, Internet Draft (draft-ietf-manet-aodv-09.txt)," Nov. 2001, work in Progress.

[4] N. Malpani, N. Vaidya, and J. L. Welch, "Distributed token circulation in mobile ad hoc networks," in *Proceedings of the 9th International Conference on Network Protocols (ICNP)*, Nov. 2001.

[5] X. Zeng, R. Bagrodia, and M. Gerla, “GloMoSim: a library for parallel simulation of large-scale wireless networks,” in *The 12th Workshop on Parallel and Distributed Simulations (PADS’98)*, Banff, Alberta, Canada, May 1998.

[6] R. A. Meyer, *PARSEC User Manual*, UCLA Parralel Computing Laboratory, http://pcl.cs.ucla.edu.

[7] *QualNet*, http://www.scalable-networks.com/products/QualNet.

[8] *OPNET Modeler*, http://www.opnet.com/products/modeler/home.html.

[9] “NIST ad hoc on-demand distance vector (AODV) routing protocol OPNET model,” http://w3.antd.nist.gov/wctg/manet/prd_aodvfiles.html.

[10] “NIST dynamic source routing (DSR) routing protocol OPNET model,” http://w3.antd.nist.gov/wctg/prd_dsrfiles.html.

[11] D. Cavin, Y. Sasson, and A. Schiper, “On the accuracy of MANET simulators,” in *Proceedings of the Workshop on Principles of Mobile Computing (POMC’02)*. ACM, Oct. 2002, pp. 38–43. [Online]. Available: http://lsewww.epfl.ch/Publications/ById/319.html

[12] H. Lundgren, D. Lundberg, E. Nordström, C. Tschudin, and J. Nielsen, “A large-scale testbed for reproducible ad hoc protocol evaluations,” in *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2002)*, Orlando, Florida, Mar. 2002.

[13] J. Flynn, H. Tewari, and D. O’Mahony, “JEmu: A real time emulation system for mobile ad hoc networks,” in *Proceedings of the Firsit Joint IEI/IEE Symposium on Telecommunications Systems Research*, Dublin, Ireland, Nov. 2001.

[14] S. Sanhani, T. X. Brown, S. Bhandare, and S. Doshi, “EWANT: The emulated wireless ad hoc network testbed,” in *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2003)*, New Orleans, LA, Mar. 2003.

[15] P. Zheng and L. M. Ni, “Emwin: emulating a mobile wireless network using a wired network,” in *Proceedings of the 5th ACM international workshop on Wireless mobile multimedia*. ACM Press, 2002, pp. 64–71.

[16] H. Miranda, A. Pinto, and L. Rodrigues, “Appia, a flexible protocol kernel supporting multiple coordinated channels,” in *Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21)*. Phoenix, Arizona, USA: IEEE Computer Society, Apr. 2001, pp. 707–710.

[17] N. T. Bhatti, “A system for constructing configurable high-level protocols. Ph.D. Dissertation, University of Arizona,” 1996.

[18] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu, “Coyote: a system for constructing fine-grain configurable communication services,” *ACM Transactions on Computer Systems*, vol. 16, no. 4, pp. 321–366, Nov. 1998. [Online]. Available: http://www.acm.org:80/pubs/citations/journals/tocs/1998-16-4/p321-bhatti/

[19] P. Urbán, X. Défago, and A. Schiper, “Neko: A single environment to simulate and prototype distributed algorithms,” *Journal of Information Science and Engineering*, vol. 18, no. 6, pp. 981–997, Nov. 2002. [Online]. Available: http://lsewww.epfl.ch/Publications/ById/307.html