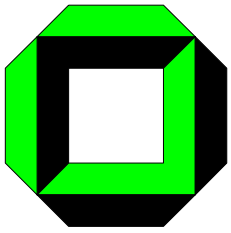
DIANEmu – A Java Based Generic Simulation Environment for Distributed Protocols[1]

Michael Klein

kleinm@ipd.uni-karlsruhe.de

Technical Report Nr. 2003-7



University of Karlsruhe
Faculty of Informatics
Institute for Program Structures and Data Organization
D-76128 Karlsruhe, Germany

[1]The work done for this report is partially sponsored by the German Research Community (DFG) in the context of the priority program (SPP) no. 1140 [1].

Abstract

DIANEmu is a simulator that has been developed within the DIANE project. It aims at simulating generic, distributed protocols on application level. Thus, its network model is fairly simple, concentrating on the higher layer: It provides many functions to facilitate the complex task of implementing distributed protocols and offers sophisticated tools to simulate the motion and behavior of users realistically. The simulator has a layered design and is completely written in Java, so own java-based protocols can be easily integrated. A database based measuring concept offers further support for protocol evaluation.

Chapter 1

Introduction

The simulation tool DIANEmu was developed within the DIANE project [2]. DIANE is funded by the priority research program (SPP) no. 1140 of the German Research Community (DFG) [1]. The project aims at developing and evaluating concepts that allow for an integrated, efficient, and effective use of resources in the form of services in ad hoc networks. In this respect, information services are especially important since they facilitate the integrated access to information that is digitally available. In order to attain these goals, DIANE proposes mechanisms for the description, discovery and execution of services. Furthermore, it takes a closer look at means of stimulating the provision of services. As scenario, the developed concepts shall be evaluated by computer science students that prepare their exam in information systems.

As an evaluation with real devices has turned out to be difficult because of lacking existing ad hoc networks, DIANE decided to evaluate their techniques within a realistic simulation environment. However, existing tools like peer-to-peer or agent platforms exposed to be unsuitable as they are based on a different paradigm. Also, specialized network simulators like GloMoSim, ns2, QualNet, or OPNet++ were not usable for DIANE's application level protocols and did not offer much support for realistic user models, but were limited to simple analytic motion models and traffic files. Moreover, the model of the lower four layers were far too detailed for the project leading to a huge amount of parameters that had to be set correctly and that influenced the results disproportionately. As another drawback, protocols that were developed for these simulators could not be used on real devices.

As a result, DIANE decided to develop its own simulation environment, DIANEmu, which, on the one hand, should be adapted to the project's need, but on the other hand be generic enough to be useful in other environments, too. In this report, DIANEmu's goals (Chapter 2), its layered architecture (Chapter 3), a more detailed and technical view to the main part of the simulator, the protocol layer (Chapter 4), and its measuring concept (Chapter 5). Finally, in Chapter 6, the reader gets further information how to obtain and use the environment himself.

Chapter 2

Goals of DIANEmu

When the simulator was designed in summer 2002, four main goals were pursued:

1 GOAL 1: Simulate ad hoc network protocols on a high level

The purpose of DIANEmu is to simulate ad hoc networks on a high level. High level means that the simulator is able to run application level protocols for ad hoc networks. Protocols for the lower ISO/OSI levels 1 to 4 (like routing or transport protocols) are not in the focus of the simulator and cannot be simulated. To support writing of these application level protocols, the simulator should

1. provide an environment that facilitates the design, implementation, and running of especially these protocols.

2. simulate users of the protocols by making it easy to code actors with their movement and their behavior (which protocol functions are called, when is a devices switched on and off etc.)

3. provide an easy yet powerful interface to the lower four layers by offering all kinds of message sending functions as well as special inter-layer functionality.

As the simulation lays stress on the application layer, the lower layers are emulated approximately only, i.e. the lower level protocols are not completely implemented, but abstract models are used to calculate the characteristics of this rather complex protocol stack. Therefore, the simulator's strength lies in proving the correctness of a protocol, not necessarily in measuring a very accurate performance.

2 GOAL 2: Protocols are easy to implement

As the main purpose of the simulator is to run user defined protocols, the second goal of DIANEmu was to provide an environment that facilitates their implementation. Therefore, the simulator should have several characteristics:

Programming Language. The simulation environment as well as the user defined protocols should use a object-oriented, robust, and common programming language to facilitate programming and enable student programmers to use the knowledge learned during their studies.

Clear interfaces. When implementing protocols, the programmer should have a clear interface to the simulation environment, which relies on a few classes only. This allows programmers to abstract from the simulator's implementation and to concentrate on a few contact points only.

Good Documentation. To enhance transparency while programming, the environment should be clearly documented. This should be done with (1) standardized comments in the program code and (2) and external reference manual.

Specialized Debugging Possibilities. Programming distributed protocols for mobile ad hoc network is a complex and error-prone activity. Therefore, the simulator should provide possibilities to step through the single events of the protocol execution. With that, conception and programming errors can be found more quickly and reliably.

Customizable Visualizer. The environment should offer the possibility to visualize the moving nodes, the sending of messages and the set up structures graphically. Therefore, on the one hand, the simulator should have built-in support for displaying standard concepts like nodes, messages etc. On the other hand, it should be possible to add custom visualizations for own data structures (like e.g. the data stored on the single nodes, the overlay structures they build etc.)

Powerful Helping Functions. The different protocols for mobile ad hoc networks often use common operation sequences like collecting messages or waiting for special messages. These operations should be offered by the environment to accelerate protocol implementation and to avoid re-coding of repeating tasks.

Powerful Sending Functions. As sending of messages is a very common activity, the environment should not limit its sending functions to simple text messages, but also allow the sending of more complex and structured messages (e.g. graphs of objects).

Transparent Multiuser Support. As the nodes in ad hoc networks act autonomously, it rather common that one node is communicating with several other nodes at the same time (using the same or different protocols). As this would result in multithreaded programming, the environment should do this task transparently. The programmer should be able to write the protocol as if only one session were active[1].

3 GOAL 3: Protocols can be transferred to real devices without change

A third and very important goal of DIANEmu was its support for protocol reuse. Protocols that are runnable in the simulation environment should be runnable on a real device without any protocol change. Obviously, this is only possible if adapted parts of the environment are transferred

[1] In some complicated cases, however, the programmer will have to add some synchronization code.

to the real device, too. Here, it simply acts as a gateway to the graphical user interface and the network access functions. This feature is very important as with it, protocol conception and implementation can be done completely in the simulation environment. Theoretically, the change to the real device should be done with one click.

4 GOAL 4: The environment is not specialized on project protocols

The fourth goal stresses the reusability of the environment itself. It should be implemented in a way that clearly separated the simulator and the protocols so that it can be used for *all* kinds of distributed protocols (not for special DIANE protocols only). Therefore, protocols (and their necessary data structures) should be implementable in separate packages, which can be loaded and used from the environment dynamically. Nonetheless, the environment should offer some specialized support for writing *ad hoc* protocols, like special helping and sending functions as well as typical visualization techniques.

Chapter 3

Layered Architecture

In the following, the architecture of the simulator is presented. The simulator has been designed in five main layers (see Figure 3.1a):

- The *connectivity layer* calculates if two devices are able to communicate directly. This is determined by the individual maximum radio range of each device as well as environmental obstacles.

- The *network layer* offers different methods to send messages to other nodes in the network: unicast, i.e. exactly to the node with a given destination address, multicast, i.e. to all members of a group of nodes, which have registered for that multicast address, anycast, i.e. to exactly one member of a group of nodes, and broadcast, i.e. to all nodes in the network that can be reached within a given number of hops. Moreover, the network layer sends crosslayer events, e.g., when a message has not been deliverable or a foreign message has been overheard.

- The *protocol layer* is the most important layer of the simulator. It offers the framework for hosting own protocols, which have to comply to a set of specifications. Protocols are realized as finite state machines that offer public functions to users and other protocols (like login, searchService, announceService an so on). They provide the offered functionality by sending messages over the message layer. The protocol layer offers a great amount of help to facilitate the implementation of distributed protocols. Examples for that are methods to automatically collect messages of a certain type, automatical distribution of incoming messages to the correct protocol state, transparent parallelization, and automatical multiplexing through a session concept. A more detailed and technical view of this layer can be found in the next chapter.

- The *user layer* models single users by running user scripts. These contain instructions concerning the user's life cycle (create, destroy), his device usage (switchOn, switchOff), his movement (goTo, wait), and his behavior (i.e. the usage of public protocol functions). To gain a realistic user model, an integrated model was developed to get these instructions [3]. For our campus scenario, each user is assigned a plan of activities in the real world (like visiting a lecture, going to the cafeteria, learning, borrowing a book and so on). This is used to derive movement and behavior in a natural manner. The parameters of the model have been compiled from surveys among students [4].

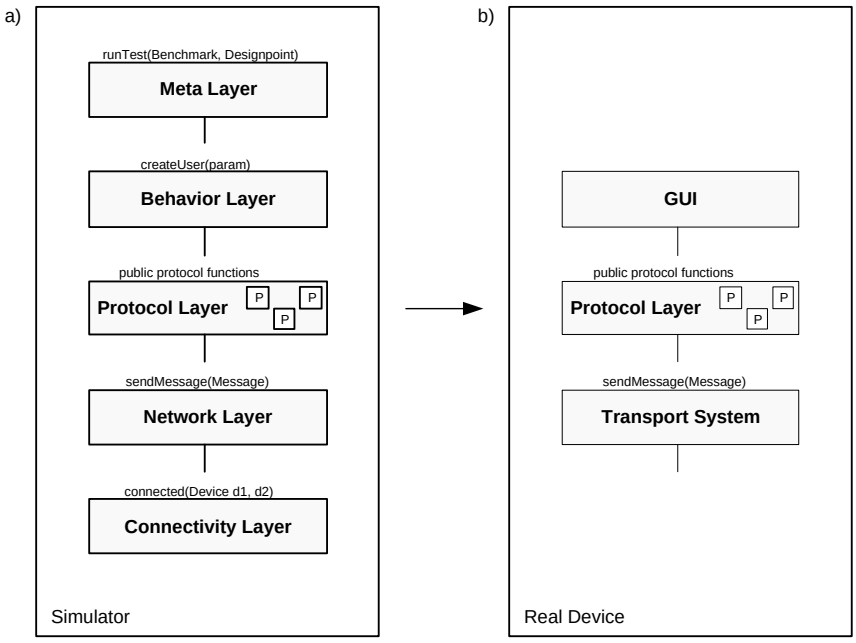


Figure 3.1: Layered architecture of DIANEmu: a) as simulator, b) on a real device.

- Finally, the *meta layer* is responsible for executing complete test cases (consisting of a benchmark and a design point, see Chapter 5). Here, the users are created as a set of scripts.

As respects Goal 3, all protocols that have been developed should be runnable on a real device without any change. This becomes possible by separating the protocol layer with all necessary protocols from the simulator and transferring it to the target device (see Figure 3.1b). Here, the public functions are made accessible by a gui or an application, the possibility to send messages is provided by an external transport system.

DIANEmu is designed as a event-based simulator, i.e. all events like arrival of a message, usage of protocol functions etc. are put into an event queue together with their time when they will occur. The event queue processes them step by step, which commonly leads to further events being enqueued. As an advantage, the simulation time is strictly separated from the real time, making measurements independent from the underlying machine. However, when transferring the protocols to a real device, the running mode must be changed to thread-based execution. Here, each device has a set of threads performing the different tasks (like handling the incoming message, the state transition etc.) autonomously.

Chapter 4

Protocol Layer

This chapter gives a detailed and more technical introduction to the main part of the simulator: the protocol layer. In order to get an overview of the basic functionality of the simulator, it is possible to skip this chapter.

1 Overview

The protocol layer is a typical component architecture: it offers the possibility to deploy own protocols as components and let them run. To assure a correct communication between protocol and protocol layer, all classes of the own protocol have to extend abstract classes and implement the a set of abstract methods. Protocols that are deployed in a correct way are able to make use of the benefits which the protocol layer is providing in a transparent manner:

- The functionality of a protocol can be split up in several states and transitions between these states. The protocol layer automatically assures the correct transition between these state when messages arrive at the node.

- The protocol layer offers several functions that are needed very often when writing distributed protocols. One example of this is the collection of messages of a certain type.

- The protocol layer offers a variable manager to organize access to inter-protocol variables. In this variable manager, each deployed protocol can store and retrieve object values with the help of a unique name.

- The protocol layer offers a timer which is often needed to detect protocol failures.

- With the help of the SessionID concept, the protocol layer transparently handles concurrent protocol access. In a nutshell, each new conversation between two or more nodes opens a new session which is identified by a unique id. Thus, all messages can be automatically assigned to the correct conversation. If necessary, the user can restrict this parallelism by explicitly setting and relinquishing locks.

Notice that each node in the network has its own instance of the protocol layer. Thus, each node also has an own instances of all loaded protocols and all of their states.

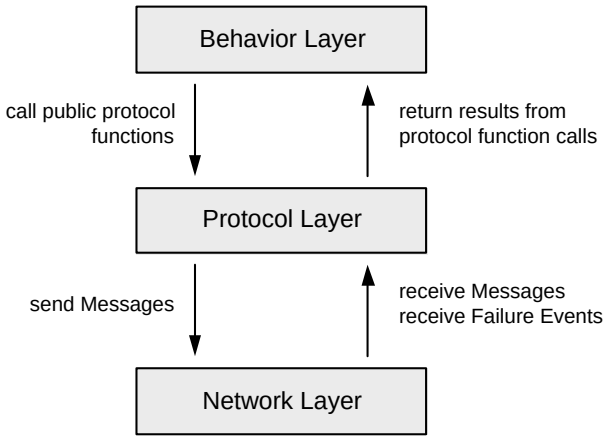


Figure 4.1: Interaction of the protocol layer with other layers.

2 Interaction with other Layers

The protocol layer interacts with two other layers: the behavior layer above it and the network layer below it (see Figure 4.1).

Generally, the interaction with the behavior layer stems from the set of users, which are maintained in this layer. Each of these users is performing some action either as simulated user by running a action script or by following a real user's inputs from a gui. Typical actions are movement (which does not directly affect the protocol layer) and calls to public functionality offered by the protocols that are loaded on the node. In the other direction, results of such function calls are returned out from the protocol layer to the user.

The network layer offers the possibility to send messages throughout the network. Generally, protocols will need to send messages to other nodes in order to fulfill their offered functionality. In the same way, messages from other nodes are received from the network layer. In case of network errors, the network layer also is able to send failure events.

The general interaction sequence looks like this (compare Figure 4.2):

1. To use a certain functionality, a user invokes a public method of a protocol. This could be `loginIntoNetwork` or `searchService`.

2. The appropriate protocol instance generates a special `Call` object containing the parameter information of the method invocation and forwards it to a special handling class: the `ProtocolHandler`.

3. The `ProtocolHandler` delivers the call to the initial state of the protocol.

4. The protocol starts running, i.e. the state of the protocol changes while messages are sent and received via the network layer.

5. If the protocol runs in a terminal state, a `Response` object is created containing the results of the protocol function. Like the call, it is forwarded to the `ProtocolHandler`.

6. The `ProtocolHandler` delivers the call to the invoked method, which unpacks it and returns the result to the waiting user.

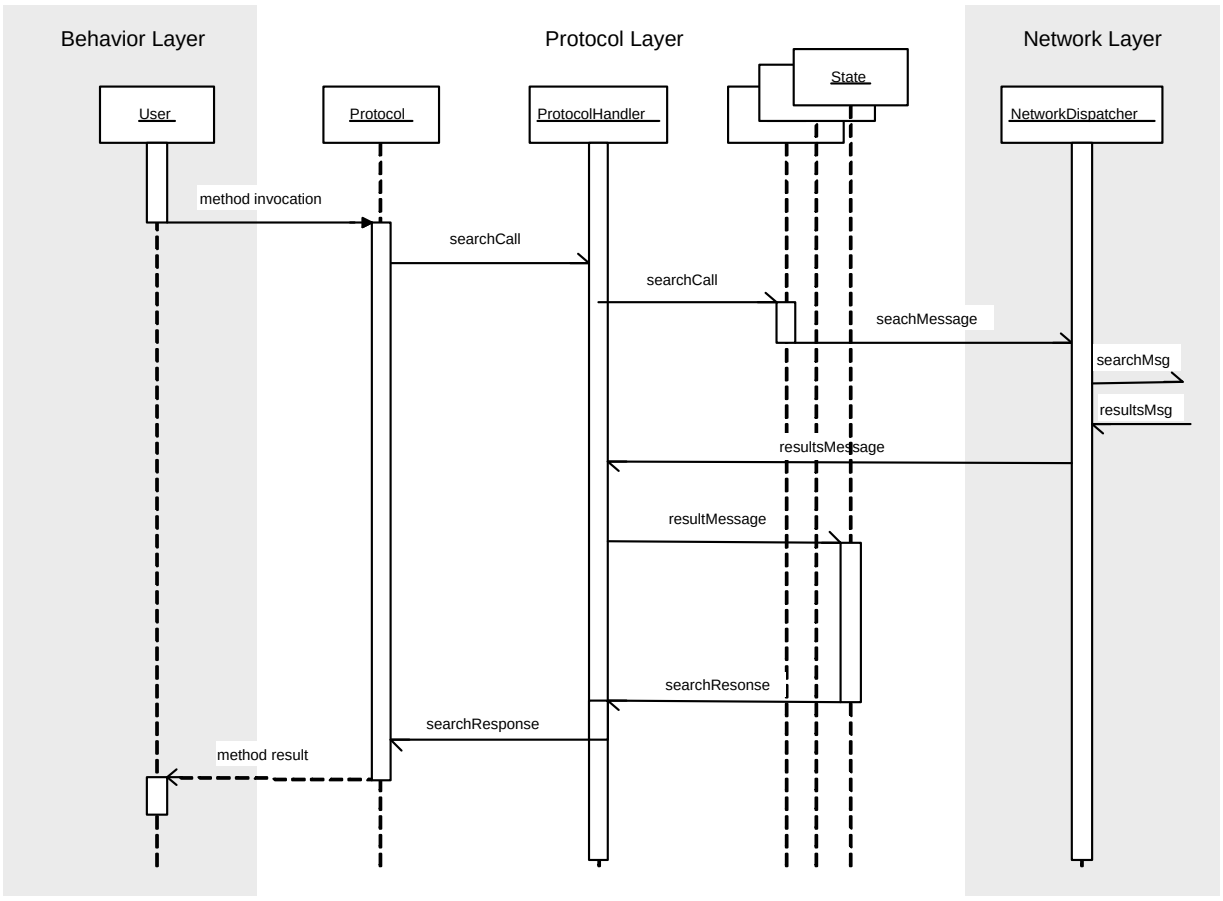


Figure 4.2: The general interaction sequence with the protocol layer.

Chapter 5

Measuring Concept

A *run* in the simulator is started from the meta layer. It is also possible to start a butch run consisting of several runs. The necessary parameter settings for a single run are separated in two classes:

- A *benchmark* captures all parameters concerning the "external world", for example the number of users, their motion model, the radio range of the devices, etc. They try to capture a realistic picture of the world and are used as a basis for comparing the protocols. Therefore, benchmarks are not altered very often, but should be fixedly given and standardized by the project manager. Typically, they have no or only a little number of parameters like the number of users.

- Other parameters are captured in a *design point*. Therefore, a design point specifies the protocol settings, i.e. which protocols should be used and how to configure them. Typical parameters are timeout values, buffer sizes, caching algorithms, and so on. Design points are not fixed, but should be altered by the developers to improve their protocols.

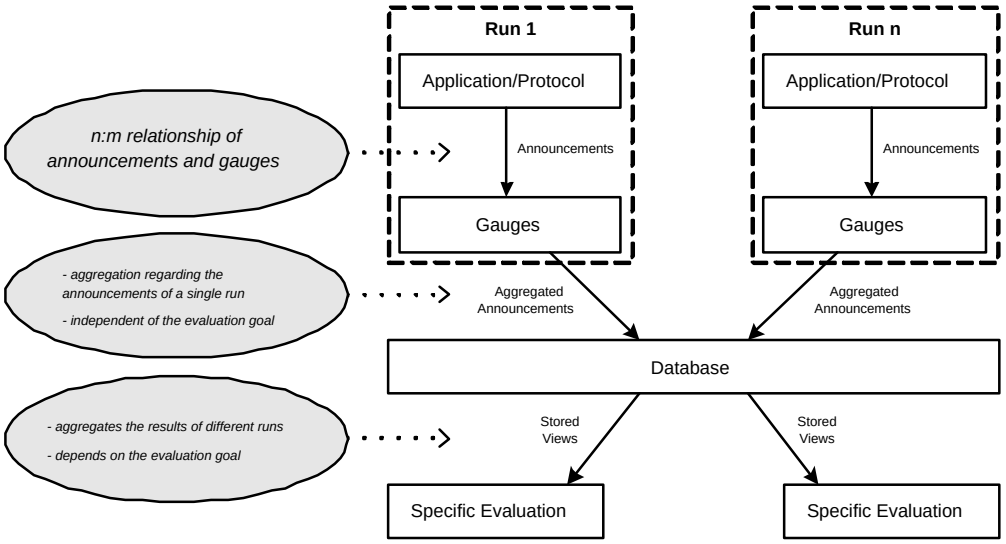


Figure 5.1: The steps of an evaluation in DIANEmu. Thanks Philipp Obreiter for this picture.

To perform measurements, the DIANEmu uses the concepts of announcements and gauges. If certain customizable events occur within the simulator (e.g., a message is sent or a service search is started), the simulator throws an announcement with details of the event. This announcement is processed by a set of gauges that have registered for this type of events. The task of gauges is to collect and aggregate these single events, storing the result in a structured manner into a relational database (see Figure 5.1). In principle, gauges are independent from the evaluation goal. Typical gauges are:

- *MessageGauge.* It protocols each message in the network together with its sender, receiver, size, hopcount and time delay. This can be used to calculate the number of messages that were used.
- *TimeDiffGauge.* It protocols the time difference between a start announcements and a finish announcement, e.g. for measuring the time delay between the invocation of a service search and return of the results.

To compare values from different runs, the database stores values from each run tagging them with a timestamp as well as with a benchmark and designpoint id. Stored views and specialized postprocessors help to aggregate the values from several runs to interesting tables, which can be exported and visualized by a chart generation tool like Excel.

Chapter 6

Technology and Download

The simulator is implemented in pure Java. Also new protocols have to be implemented in Java. This is done by extending a set of abstract classes like `Protocol`, `State`, `Message`, and so on. This allows to easily integrate other java-based modules, like external service matcher, loggers, libraries, java-based ontologies etc.

To compile the sources as well as to start the simulator or other tools, Apache's `ant` [5] was used. Ant helps to centralized the building and execution settings in one shared file, making it easier for every participant to use the environment on its own machine.

To get a quick introduction to the simulator, a manual containing a short tutorial is available. It shows how to install the environment, how to use it from the most common programming environments and gives step-by-step guide how to implement your first simple ping-pong protocol. The manual also serves as reference literature for detail questions.

The simulator as well as the manual can be obtained by sending an email to Michael Klein (`kleinm@ipd.uni-karlsruhe.de`).

Bibliography

[1] DFG (Deutsche Forschungsgemeinschaft): Schwerpunktprogramm 1140: Basissoftware für selbstorganisierende Infrastrukturen für vernetzte mobile Geräte. http://www.tm.uka.de/forschung/SPP1140/ (2003)

[2] Institute for Program Structures and Data Organization, Universität Karlsruhe: DIANE Project. http://www.ipd.uni-karlsruhe.de/DIANE/en (2003)

[3] Breyer, T., Klein, M., Obreiter, P., König-Ries, B.: Activity-based user modeling in service-oriented ad-hoc-networks. In: First Working Conference on Wireless On-Demand Network Systems (WONS 2004)., Trento, Italy (2004)

[4] Breyer, T.: Modellierung der Bewegung und des Verhaltens von Dienstnutzern in mobilen Ad-hoc-Netzen (2003) Diplomarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[5] Apache: The ant project. (http://ant.apache.org/)