JANE – A Simulation Platform for Ad Hoc Network Applications

Daniel Görgen
University of Trier
Department of Computer Science
54286 Trier, Germany
goergen@uni-trier.de

Hannes Frey
University of Southern Denmark
Department of Computer Science
5230 Odense, Denmark
frey@imada.sdu.dk

Christian Hiedels
Université du Luxembourg
Faculté des Sciences, de la Technologie et de la Communication
1359 Luxembourg
christian.hiedels@uni.lu

ABSTRACT
This work describes a Java based application development, evaluation, and testing platform JANE (Java Ad hoc Network Environment) which is intended to support ad hoc network researchers in application and protocol design. Software development within this environment is expected to follow a bottom up approach. Basic functionality is implemented in elementary components which can be combined to more complex ones by using well defined interfaces. With dynamically changing network links being rather the common case than a failure situation, asynchronous communication has been selected as the main communication paradigm within this platform. Reusability of components in different execution contexts by providing an appropriate machine abstraction is a further important design decision which drove the platform development. Code written once can be executed in a pure simulation mode, in a hybrid setting with real devices being attached to a running simulation, and, finally, in a setting using real devices only. Software development following this three-tier development process paired with the platform’s rich visualization features emerged to significantly ease the burden of debugging and parameterizing in such highly dynamic and inherently distributed environments. Some ad hoc network application programs have already been implemented on top of JANE. These applications which are described in this work as well serve as a proof of concept for the platform.

1. INTRODUCTION
Future paradigms as ubiquitous and pervasive computing rely on countless mobile devices interacting among one another using wireless communication. Groups of these devices may form ad hoc networks at any time. These networks can be divided into three general classes. (1) Pure infrastructureless networks which totally rely on other mobile devices using multi-hop routing for distant interactions, (2) single-hop ad hoc infrastructure using wireless communication as the last hop into a more reliable backbone network and (3) hybrid networks mixing these two types. Hybrid networks integrate infrastructure to improve reliability of mobile multi-hop ad hoc networks. Beside “traditional” infrastructure such as WLAN access points or 3G networks, it is also possible to think of mesh networks or even single installed mobile devices to provide infrastructural duties selforganized.

In all the described scenarios the mobile and wireless network parts are characterized by limited energy resources, a broadcast based communication media, and a dynamically changing network topology due to device mobility or energy conserving sleep cycles. Software development in such an environment requires new tools and programming paradigms which address the special needs under these harsh conditions. In addition, implementing and evaluating applications and algorithms in a mobile ad hoc environment is an extensive task when using real mobile devices in field trials. A large amount of devices is needed and of course enough people to handle them. Thus, it is very hard to obtain reproducible scientific results. Using network simulators for testing and evaluating network protocols for multi-hop ad hoc networks is state of the art. But most of these simulators are not intended to implement applications on top of the simulated network. Moreover, testing applications solely in a simulated and thus idealized environment may produce misleading results or interpretations. A promising approach is the three step development process originally proposed in [21]. Applications are implemented, tested and evaluated in a simulated environment first. Second, the application is tested with real user behavior but still in a simulated environment. Finally, dedicated field trials are used as proof of concept.

The JANE environment described in this work provides ad hoc network developers a middleware which can be used to implement and test protocols and applications by following the described three-tier design paradigm. Component based design and asynchronous communication among components are forming the key ingredients of the JANE middleware. The approach is motivated by the observation
that realizing a “traditional” view of the network stack and stream oriented synchronous communication does not cope well with the extreme dynamics of an ad hoc environment. For instance, a process blocked at a stream in order to obtain data from another mobile device might frequently be tied up with error handling due to stream disconnection. Error handling, however, should be an exception and not the general case. Moreover, event based programming perfectly fits in a frequently changing environment since applications and protocols can adapt immediately when the surrounding of a device changes. Blocking in contrast requires a timeout to occur before the blocked entity can be transferred into the next state (e.g. to realize that the stream sender is no more available).

The rest of this paper is organized as follows: The first section introduces the JANE’s component-based middleware architecture. Section 2 introduces JANE’s network abstraction and a 802.11 implementation. The JANE execution modes which enable user interaction with real devices are described in section 3. Some application examples which have been realized using JANE are shortly sketched in section 4. Related work is given in section 6 and the paper is concluded with section 7.

2. THE JANE ARCHITECTURE
In JANE, everything is a service. This includes application components, basic middleware services, network layers, routing algorithms, and even hardware such as GPS receivers are represented as a JANE service. Services cannot be accessed directly. Communication with other services is only allowed using an event based communication mechanism provided by an operating system abstraction. Thus, each device runs a couple of services, locally interacting using asynchronous signals and events. Also the ad hoc network is modeled using event-driven services. JANE networks provide asynchronous message oriented communication with other devices. Message sending and receiving is mapped to the local event based interaction primitives. This model perfectly fits to the extreme event-driven environment of mobile ad hoc networks.

2.1 JANE Services
A service component consists of at least one Java object which provides methods for event handling. These can be arbitrary methods with no return value. Events can be generated by other services, by the service itself or by an operating system component. By exporting interfaces or additional event handling objects other services are enabled to trigger these methods. At basic, the main service object provides a start and a finish method triggered by the operating system. A service is able to generate new events using the operating system. For triggering itself, it can start and stop timeouts. For triggering others, it must use the operating systems service interaction component. Each service is uniquely identified, so that it can be addressed by others.

As depicted in Fig. 1, JANE distinguishes three service types: runtime service, simulation service and global service. A runtime service can be executed in all JANE environments while a simulation service has access to simulation and thus global knowledge. This includes exact device positions of all devices, exact simulation time, simulation control and visualization settings. Moreover, this service type is able to generate events for services on other devices using the same asynchronous service interaction primitives as used for local interaction. Thus, it is possible to implement network services based on generating events on neighboring devices. Execution services are also used to implement services that avoid computational expensive network load using global knowledge and direct device interaction. This is also needed, when services are evaluated and some necessary services should not influence simulation results. In contrast to the other services which are always assigned to a mobile device, global services are simulation services which are instantiated only once within the simulation. These services are mapped to each device. As simulation services, these are also used to realize functionality based on simulation and global knowledge but using a central approach. Other purposes are realizing global visualization for painting background graphics or implementing global statistics. The services, in particular runtime services, do not see any difference between these service types when interacting. During service execution it is always possible to find out the service identification of the service causing the execution of an event handler. Simulation and global services can also retrieve the hosting device of the initiating service. This enables them to reply directly to the device they received an event from.

2.2 Operating System Abstraction
The event driven core of a JANE device is hidden in an operating system abstraction (see Fig. 2). In contrast to other simulation environments, it is not possible to access the event queue or other simulation internals directly. Services are implemented on top of this operating system and thus can easily be ported to other environments. The basic duties of the JANE operating system are service management, service interaction, event execution management and timeout management. The service interaction directives are described in section 2.3 in detail.

Service management handles service starting and shutdown at runtime and at device startup. Each service is able to
start and shutdown other services. To lookup other services for interacting, a simple service discovery mechanism is provided. This is based on the Java class hierarchy and lists all identifications of services implementing a given class or interface.

Instead of adding events to the simulation event queue directly, a service is able to set and remove timeouts. This enables a service to generate events by itself, e.g. to realize periodic tasks or to timeout communication tasks. Timeouts are set using time deltas relative to the device’s local clock.

The execution manager schedules the events on a device. Here, it is possible to reorder contemporary events, e.g. by using service priorities. The current implementation uses a FIFO ordering. The execution manager also ensures that every event is executed atomic and that each service handles only one event at a time.

### 2.3 Service Interaction

**JANE** provides two types of asynchronous service interaction: (1) signaling and (2) event firing and catching.

A service **signal** addresses an event handler object using a known receiver identification and calls asynchronously a public method of the receiving object. By default a service can be addressed by signals using its service ID. Additionally, every service is able to register objects for signal receiving. The signal receiving event is executed in the context of the receiving service (also in the case of registered handler objects). To keep things simple, it is possible to export interfaces for the service and its event handler objects. Other services can request signal proxies automatically generated by the operating system using the exported interfaces. Thus, a service can send signals to another service and asynchronously execute a signal handler method by calling exactly the same method on the generated signal proxy. To reply to an asynchronous call, the sender can pass callback objects. The receiver gets a signal proxy as a method argument, which also generates asynchronous signals that are send back again. Thus, also more complex interactions can easily be implemented. An example is a message status handler for network communication where the current status of a message propagation task is signaled asynchronously.

A service can also interact with others by firing a service event. These events are not addressed to a specific receiver as a signal. Instead, other services can register event handlers which are called when such an event occurs. To specify the events which are of interest event templates are used. A fired event is matched to all registered templates implementing the same class or one of it’s superclasses. Registering a superclass template can be used to receive all derived events. An event object is matched using its attributes. A template attribute with a null value matches all possible values. Initialized attributes must be equal to the corresponding event attribute. In case of a Java **class** attribute, the **instanceof** relationship is used instead. By default, an event contains the unique identification of the sender and its Java class. Thus, a receiver is able to specify the sender by providing one of the interfaces the sender implements. It is possible to enable template matching also for every event attribute given as Java object. The template matching is restricted by a given reflection depth, which is one (only service event attributes are matched) by default. Using a higher reflection depth causes additional computational load. But a fired event is only reflected to the maximum depth of all registered templates of the same event type. The registered event handler object can implement a default handler method which is able to handle every event type. Alternatively, it can implement an event handler interface specified by the event as it is known from signals. Thus, the event can execute the correct method and pass all necessary event attributes. The event handler does not need to cast the event and it is possible to implement event handler methods for different events within one object.

### 2.4 Device Mobility

Since **JANE** is intended for a mobile environment, the simulation core must realize device mobility. At startup, it is possible to add mobility sources to the simulation. A mobility source provides enter and move events for a set of devices. A move event specifies the position of a device at a given time. A device is moved on a straight line with a constant velocity from one position to another within the time delta given by two consecutive move events. Thus, arbitrary device movement can be approximated using a sequence of lines.

Beside simple stationary scenarios where device placement follows given rules and simple random waypoint and random walk variants also more complex mobility sources have been implemented. The pathnet mobility source uses a graph given as XML file for device movement. Vertices are crossings or endpoints and contain routing probabilities to endpoints for outgoing edges. Endpoints (e.g. rooms) can be arbitrary randomized position generators and can be entered over at least one graph edge. Edges can have an arbitrary width so that devices are moved on a lane and not only on a strict line. Devices are moved on the pathnet by providing the next endpoint and the moving speed. The device’s route within the pathnet is chosen at random due to the given routing probabilities. The endpoint provides the final position.
Some mobility models also provide an interactive setting. There, the devices can also be moved by user interaction with the simulation GUI at runtime. The device movement is restricted by the underlying mobility models. Using the pathnet model, the user can only move devices between pathnet endpoints. This setting is used for interactively “playing” with the mobility scenario during testing or debugging and also for application demonstration.

At startup, devices can be grouped together. Each device group is assigned to a mobility source, so that it is possible to model different device behavior. Each group can be started with a different set of services. This feature is used to set up simulation with heterogeneous functionality and behavior very simple. Device can be stuck to fixed positions providing selforganized ad hoc infrastructure features while the rest randomly moves on the plane. Other scenarios are used to analyze applications only on a small subset of devices while the rest only act as transparent multi-hop communication routers. Different mobility patterns can also be used in combination with different simulated user behavior patterns implemented as services on a device. Besides grouping, it is of course possible to assign each device its own service set.

2.5 Visualization

For testing, debugging and demonstration JANE provides extensive visualization possibilities. Every service is able to visualize its state by providing arbitrary shapes like lines, rectangles, ellipses or collection of shapes. Shapes can be defined using simulation positions or by using device addresses as points. The shape rendering is decoupled from the simulation so that it does not influence the simulation behavior. The rendering unit can use a GUI canvas (e.g. Java2D and OpenGL [22]) for drawing and, if desired, the unit can also render to file e.g. as Postscript, PNG or XML.

Using the visualization is computational expensive and slows the simulation down. But it gives sometimes better insights as it can be achieved by evaluating trace files. For long running statistic evaluation, the visualization can completely be deactivated. This causes no additional computation load if the service shape generation is implemented appropriately by providing the shapes only on demand.

The visualization GUI can be extended to pass user interaction to a running simulation. This causes, of course, non determinism and thus non reproducible simulation runs. But for testing and also for presentation this is a very helpful feature. Such user interaction can use the same operating service abstractions like a global service. A user interaction is therefore able to generate arbitrary events for each running service on each simulated device. In combination with a mobility source which also supports user interaction, it is also possible to influence the device mobility using the visualization GUI.

3. JANE NETWORK

In JANE, also the wireless ad-hoc network is implemented as a set of services and event handling objects. A JANE network is represented at link layer. Basically, it provides asynchronous message oriented unicast and broadcast communication which addresses direct neighbors in a wireless ad hoc setting. Other services can communicate over the network by signaling a message communication task to one of the running network services. To retrieve message status events, the client is able to provide a callback handler which can be signaled by the network service. At minimum, a network signals success, failure and timeout for reliable communication and at least it signals when a message has been completely processed locally, e.g. completely put on the media.

For simulating network communication, a message must specify its size. This reduces computational load and also enables the developer to use simple message implementations which are not necessarily optimized for network communication. The developer can also piggyback additional e.g. statistical data which should not increase the message size. Additionally, messages are able to define arbitrary shapes for network visualization. Received messages are delivered as events, so that a service must explicitly register itself or an event handling object for message reception. The event receiver has also access to an enriched message header which also provides signal strength respectively distance information of the message sender.

JANE contains an implementation of the IEEE802.11 standard for ad hoc networks. The implementation is divided into three services: the medium, the physical layer and the MAC layer. A global service simulates the radio propagation model of the shared medium, e.g. a two-ray-ground model. It uses discrete slots for calculating the receive respective the interference signal strength observed on other devices which is signaled to the device’s local physical layer services. Due to all observed signal strength, the physical layer decides whether a signal is received or not. Like the ns2 [15] physical layer implementation, it also regards capture effects, so that a signal is received, when the signal to noise ratio is greater than a specific threshold. The MAC layer implements only the ad hoc part of the IEEE802.11 standard. The sending of MAC frames is mapped to turning the radio on and using a time delta calculated from the frame length and the used date rate. As mentioned above, messages in JANE always provide their size for network simulation.

JANE also provides other network services, e.g. for simulating wireless infrastructure networks. For simulating wireless ad-hoc networks, implementations with an idealized MAC-Layer can be used to reduce network computation load significantly. Since networks are realized as services, more than one network can be instantiated, e.g. for hybrid network settings. Due to the open service architecture it is possible and also simple to realize additional networks.

4. REAL USER INTERACTION

In particular for applications in mobile ad hoc networks, it is necessary to evaluate and test the application components not only with simulated, but also with real user interaction. The JANE simulation environment enables applications written once in the simulation environment to be executed on real devices as well. At this, the environment distinguishes between hybrid setting and platform core as two possible ways of executing application code on real devices.

4.1 Hybrid Setting
A simulation enables the application developer to test applications using a very large amount of (simulated) devices. Thus, it is helpful if real devices can be connected to the simulation and users are able to interact with others using a real device on a simulated network. This hybrid setting makes it also possible to integrate simulated-only devices, e.g., as transparent message routers in a multi-hop scenario or to represent passive users. Moreover, also simulated user behavior can be used to create large scenarios for tests with real user interaction.

Real devices are connected to an existing device within the simulation. The client is able to choose the device or is assigned to an arbitrary unconnected one. The external device can start arbitrary runtime services. Services on the simulated and on the connected real devices are merged transparently as if they were on the same platform (see figure 3). A service does not see any difference when interacting with another service. It is possible to start only the non runtime service within the simulation and the rest on the real device or letting everything within the simulation and just start an external application GUI instead of a service simulating user behavior. This transparency can easily be achieved due to the event-based interaction between services. Synchronous operating system calls causes the simulation to stop between two event executions until it is completely processed, so that an event stays atomic and no inconsistencies can occur. Services and registered event handlers are stored within the external operating system and the simulated operating system, respectively. Only event handler call descriptions are transmitted. These calls are appended to the simulation event queue or to a thread-driven event queue on the external device to decouple the operation of simulation and external devices. This causes, of course, non-reproducible simulation behavior. However, the hybrid scenario is used for real user behavior which is indeed non-deterministic.

The connection is achieved using Java RMI. This enables the connection of external devices also over a network. Using WLAN for connection allows to integrate also small wireless devices as PDAs which gives the user a good feeling of the real live behavior and operation of the application.

4.2 Platform Core
All implemented runtime services can be tested also in an execution environment using real devices and a wireless network interface. This is possible without any modification of the services. Thus, the application code can be completely implemented and tested within a simulated environment and can afterwards be used on a real device without any additional effort.

The execution manager of the platform uses a simple thread-driven FIFO queue for contemporary events. Timed events of the timeout manager are driven by the standard Java timer system and are queued to the execution manager. A multithreaded core is also possible but was not needed yet since only single prototype applications have been tested so far.

Since the platform does not provide network communication within the operating system, the network communication is implemented within a runtime service which maps the com-
communication events to network communication and vice versa. The implementation uses UDP unicasts and broadcasts and maps the JANE’s link layer features to these primitives. For demonstration purposes, the network connectivity can be reduced by simply discarding messages due to virtual device positions and sending ranges.

Messages are serialized using the standard Java object serialization. The network implementation also allows to hook in specialized serializer for a more efficient bit packing. Some of the standard services in particular the beaconing service provides more efficient message packing.

For applications and algorithms using device positions a service has been implemented which provides the current device position from GPS or by user interaction. Additionally, a GUI service can visualize the local running services and if beaconing is used, also the current neighborhood (see fig. 4).

5. JANE IN ACTION

We have implemented JANE to investigate applications in mobile multi-hop ad hoc networks in order to derive new self-organizing and new communication patterns in this type of network. In the past years a couple of applications have been realized. The experiences and also the application needs led to an iteration and extension process that resulted in the current version of JANE. Exemplarily, three of these JANE applications are sketched in the following sections.

5.1 UbiSettlers

UbiSettlers is a real-time strategy multi-player game, running in an ad-hoc network in combination with an infrastructure network. It is roughly inspired by a popular German board game called “The Settlers”. Every player controls an island and tries to establish an infrastructure, consisting of buildings, which are constructed using resources like Stone, Grain, Iron Ore, Wood and other. The buildings again provide bonus on the gathering of resources, thus enabling players to get more powerful structures. UbiSettlers is working just fine in single-play, but teamwork with other players is better and in fact emphasized by things like allowing only collaborating players to construct high-level buildings like a cathedral. Another way to get additional resources is trading. Each player is able to send out trade offers stating a proposal to give some own, not needed resources in return for other resources. Other players receiving such an offer can respond and establish trading with a simple handshake model. Devices not running UbiSettlers, forward messages altruistically, by using an implementation of the Lightweight Mobile Routing protocol (LMR). The graphical user interface of UbiSettlers is designed for using on PDA/MDAs in an intuitive way. Every action in the game is visualized, enabling players to react immediately. It is able to run in simulation, hybrid and platform mode of the JANE platform.

Figure 5: Screenshot of the UbiSettlers game.

5.2 NetNibbles

Basically, NetNibbles is a multi-player variant of the well-known game Nibbles (also known as Snake) designed for ad-hoc environments. Players control a never stopping Snake and collect edible objects which lengthen the body of the Snake permanently. At the same time, they try to avoid obstacles, such as walls and the bodies of other Snakes from other players. Multiple players thereby establish a logical group. Like in most games the players can achieve some score and they usually want to propagate their high score. For that purpose, a top ten high score list is introduced and maintained by a game server residing in a special device. Every time a player achieves a new high score, which is determined based upon the top ten list stored locally on each device, that high score list is updated and propagated within the logical group using ad-hoc communication. After terminating a full game session, the central game server will be updated only once. Due to the fact that different players - maybe in different groups - can achieve new high scores at the same time, the high score list might be updated at different places concurrently. The central server finally needs to conciliate the potentially conflicting different high score lists, integrating them appropriately. The game is able to run on PDA/MDAs in all three simulation modes.

Figure 6: Screenshot of the NetNibbles game.
Figure 7: Screenshot of JANE in hybrid mode running the Distributed Script application. The two GUIs on the right are connected to devices visualized on the left.

5.3 Distributed Script

Distributed Script is intended to enable students to create a script during a university lecture in a distributed manner. The lecture itself is modeled as a geographic context given by locations and times. Inside a lecture context, devices are able to communicate over a few hops using a geographic bounded version of DSR (Dynamic Source Routing) and geographic bounded broadcasting strategies. Due to the well-known geographic locations, devices located there can be addressed using position-based routing strategies as GCR (Geographic Cluster-based Routing). Thus, it is also possible for absent students to create consistent parts of a script by communicating with the lecture context. Newly created material is initially propagated using SPBM (Scalable Position-Based Multicast) to address as much interested students as possible. Missing lecture material is exchanged between neighboring devices directly, also outside a lecture context when devices pass by (so-called En-Passant communication).

As all JANE applications which have been developed including a graphically user interface also this application can be used in all three JANE modes. A hybrid mode example of JANE running the Distributed Script application is depicted in figure 7.

6. RELATED WORK

GloMoSim [10] is a network simulator which focuses on scalability. It uses the capabilities of the parallel discrete-event simulation language Parsec [9]. Implemented protocols are build to use a layered approach and standard APIs are used between different layers, which enables users to integrate new models easily. Simulation scenarios are described via text files. It has various applications, transport and routing protocols, as well as miscellaneous mobility schemes. Because of the parallel approach, GloMoSim allows to run simulations with thousands of devices, which can be visualized either during runtime or later.

QualNet [6] was developed by Scalable Network Technologies based on GloMoSim. It has a lot more network models and protocols, as well as more tools to ease creating simulation scenarios and is sold as a pure sequential and a parallel version.

Ns/2 [15] is a discrete event based network simulator written in C++ and probably the best known and widely accepted simulator for computer networks. The first version ns/1 was already developed back in 1995, and meanwhile, it is the best supported simulator available. In the beginning, it did only support stationary networks, but was enhanced with mobile ad hoc network capabilities in 1998 by the Monarch Project of the Carnegie Mellon University [19]. Now a huge number of different network protocols, as well as all popular routing algorithms are disposability, e.g. a complete implementation of IEEE 802.11 [2], as well as different projects to integrate UMTS networks [8, 7, 11]. Simulations are defined through scripts written in OTcl [5]. Ns/2 performs the simulation and stores results in a trace file, which can be analyzed and visualized with external tools, like the Network Animator “nam” [3]. New protocols are directly integrated in the source code of ns/2 and are available in simulation scripts after compiling. However, it is generally known, that it takes a long time for getting used to ns/2 [13]. Also it is worth mentioning, that ns/2 scales quite bad as soon as more than a few hundred devices are analyzed and its memory requirements are huge.

With ANSim (Ad Hoc Network Simulation) [18], a network simulator was developed at the University of Bruchsal, that purposely skips a detailed simulation of the MAC-Layer and transmitting protocols and concentrates on mobility of devices. The gained amount of processing power is used to simulate large scenarios as quick as possible. Results are visualized directly during runtime and scenario parameters can be specified in a GUI, which can be used as a scenario generator for ns/2 and GloMoSim, too.

OPNET Modeler [4] is a commercial simulation environment that allows detailed simulations of vast networks. It provides hundreds of vendor specific and generic device models. Mobile devices can be placed and moved anywhere in a 3-dimensional area. Its modeling paradigm allows defining the behavior of individual objects at a “Process Level”, which are interconnected to form devices at a “Node Level”. Devices finally are linked at a “Network Level”. Simulations
can be parallelized to use multiple processors.

A parallel network simulator for simulating very large stationary multi-protocol networks is Dartmouth Scalable Simulation Framework (DaSSF) [1], which is a C++ implementation of the Scalable Simulation Framework (SSF) API [13]. Parallel simulations with tens of thousands of devices are possible using shared and distributed memory configurations on a variety of different architectures.

The Staged Network Simulator (SNS) [21] uses a performance technique, to improve the simulation scale. It is based upon ns/2, but is able to simulate around 50 times faster on certain scenarios. Thus, it can be used to perform simulations of very large networks of up to 10000 devices. The concept is to eliminate redundant computations through function caching and reuse. SNS supports all protocols that are implemented in ns/2.

7. FURTHER AND FUTURE WORK

We have presented JANE, an application development, evaluation and testing environment for mobile ad hoc networks. It was pointed out that JANE is more than just a simulation environment but provides the user a middleware platform for applications in an event-triggered ad hoc networking environment.

On top of JANE a rich set of generic services has been implemented, which has not been introduced in this work. A communication framework provides standard position-based and topology-based routing protocols which can be chosen and adapted by the application for each communicated message. This framework can also be used to combine standard protocols to new, multimodal protocols as the Distributed Script application does by combining the position-based GCR with the topology-based DSR.

The exemplarily shown applications, are making extensively use of the capabilities of JANE. Every application is able to run in simulation, hybrid and platform mode. Although NetNibbles and UbiSettlers were mainly meant as a means to present research aspects like reconciliation of structured data and collaboration between devices, JANE allows them to be funny games on realistic hardware like PDAs. Especially the simultaneously running of more than one network model extends future research on hybrid networks. Some issues concerning the platform mode particularly in combination with more than one network protocols need to be dealt with in the near future.

A deeper integration of JANE’s network capabilities within a Linux kernel module is still work in progress. Although the single threaded platform event-queue is good enough for application prototype tests and demonstrations, a multi-threaded core is much more suitable for realizing large environments or multi-application tests. The environment is extended continuously. Beside standard position-based and topology-based unicast routing protocols currently available within the routing framework, also other protocols, for example multicast protocols, should be realized.

8. REFERENCES


