Emulation of Radio Access Networks to Facilitate the Development of Distributed Applications

Tim Seipold
seipold@informatik.rwth-aachen.de
Department of Computer Science 4, RWTH Aachen University of Technology, Germany

***Abstract*— Emulation of Radio Access Networks is a valuable tool for the development of distributed applications incorporating mobile terminals. The network simulator *ns–2* provides mechanisms to route live network traffic through its simulation and thus emulate a network in real-time. However, it lacks certain core features for the emulation of networks with mobile terminals: roaming of terminals across base stations, dynamic addressing of nodes, and the export of these changes to the outside application to evaluate its capabilities in dealing with these consequences of terminal mobility. To enable this, a new set-up for emulation is proposed and implemented using *ns–2*. An *infrastructure* routing protocol (INFRA) is presented, as well as an additional `TAP` network object and a `TAP` agent that were required to unleash the full potential of the new routing protocol. The results of several experiments applying these new components expose the added value of the new emulation setup for the evaluation of distributed applications.**

Index Terms*— network emulation, Radio Access Networks, wireless, roaming, DHCP, *ns–2

I. INTRODUCTION

With the dawn of affordable and ubiquitously available **R**adio **A**ccess **N**etworks (RANs) the inclusion of wireless terminals in distributed applications became a logical extension of existing distributed systems. Especially in the business environment this is an attractive development, since it allows the integration of participants into the business process that have been excluded beforehand. Using RANs mobile entities can be completely incorporated into an **E**nterprise **R**esource **P**lanning **S**ystem (ERPS) or similar distributed applications and have continuous access to the up-to-date information within the system. The same way the status of the mobile entities can be kept up-to-date within the system, thus allowing better planning of the mobile entities' tasks. This is especially interesting for enterprizes, where mobile entities constitute a core domain of the business operations: cargo haul, on-site customer service etc. Fig. 1 provides a systematic overview of such a system.

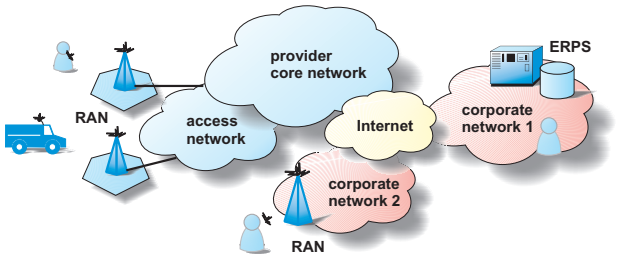


Figure 1: Overview of a distributed application incorporating mobile users connected by RANs

For like applications it is not feasible to implement a company owned RAN but buying transport services from communication providers. Covering a wide area this usually implies data transport via GPRS, UMTS etc. Those networks operate opaque to the user and the effects of the RAN are only indirectly visible to the user buying transport services. However, at the factory premises company operated RANs like WiFi or Tetra are a necessity, as they increase network performance, minimize transportation cost and make the RAN's status accessible to the user's application. Obviously, in such an environment the terminals are exposed to several issues created by the availability of several wireless networks: there may be none or several transport networks of different quality and cost available, and switching access points within a network (hand-over) or even between different networks (hand-off) vastly affects the data transport.

While some of these issues can be addressed on the **O**perating **S**ystem (OS) level, other issues need to be handled within the application. Especially decisions involving a quality-cost-tradeoff can not be made by the OS, as the required information is only available to the application. Both levels of decision need testing and tuning of parameters to reach a sufficient performance. This is, however, not as trivial as one initially supposes.

Real tests involving the effects of RANs require, beside real hardware and an existing software, real networks and real mobility. Most RANs operate on a regulated spectrum and thus forbid the implementation of a test bed, even if the necessary funding would be available. Thus, the test have to be done using the commercial providers' networks, which generates costs for the date transport, introduces cross traffic into any measurements, restricts

This paper is based on "Improving emulation of wireless access networks in *ns–2*" by Tim Seipold, which appeared in the Proceedings of the 3th IEEE Conference on Wireless Mobile Computing (ICWMC), Guadeloupe, France Caribbean, March 2007. © 2007 IEEE.

This work was supported in part by the Bundesministrium für Forschung und Bildung (BmBF) under the project code 01ISC25

access to network internal data and limits the test set-up to a given infrastructure. Further, generating the terminal mobility for real tests consumes significant resources, as all terminals need to be physically moved and generating a sufficient amount of hand-over/hand-off events takes time. Last but not least the control over and thus the reproducibility of a scenario in case of real tests is very limited. Therefore, real tests are no means that can be used during application development but only suitable for the final verification steps before deployment.

Another method of application evaluation is mathematical modelizing. Using Petri nets, queue networks or other modelizing techniques a representation of the application's behaviour is generated that is solved for certain parameters using mathematical methods. This approach's results are usually obtained rather quick, as tools can be used to solve the models. However, with increasing realism of the model its complexity and thus the time to compute a solution increases as well.

A minor drawback of this approach is the necessity of maintaining a model synchronous to the actual application. Expertise in modelizing is required, and must be continuously applied during the development cycle to update or refine the model, creating an overhead. Hence the major drawback emerges once third-party components are used for the application, as e.g. middleware systems. The internals of these components affect the application performance, but the modelizing can be based on an educated guess only, as these internals are usually hidden. This results in poor models of critical aspects of the system delivering coarse results and leads to a limited applicability of this approach as a tool in the development cycle. As mathematical modelizing neither requires the actual hardware nor the actual software, it has its value as a tool for roughly assessing the viability of a solution before any real development has been done.

Simulation as an evaluation method is rather similar to mathematical modelizing. However, instead of a mathematical model of the system, program objects with a simplified version of the real components' behaviour are combined to resemble the real system. This so-called simulation scenario is executed within a simulator and the modeled components interact the same way as the real components would. These interactions are logged with time stamps and allow a subsequent analysis of the combined system's and thus the application's behaviour.

As with mathematical modelizing, simulation lacks precision as soon as third-party components are involved, as their details can be modeled by educated guess only. On the other hand simulation requires significantly less additional expertise than mathematical modelizing, as the objects' behaviour is programmed rather than described by mathematical abstractions, and scenarios are composed the same way as the real systems are. This way simulation is much more accessible to the average developer than mathematical modelizing. Further, in most cases a simulation model can be constructed from reusable predefined components that are common between many applications, as e.g. network models (physical, link, and transport layer), routing models etc. The application behaviour itself needs still to be modeled, and it needs to be kept consistent with the actual application. This creates an overhead during the development process, and as with the mathematical modelizing the quality of the model is decisive for the quality of the results gained by this approach. Widely used network simulators with focus on wireless networks are *ns–2* [1], OPNET [2] and Glomosim [3]. There are some approaches to automatize the generation of simulation models of new protocols by using special frameworks for the implementation. This way, a suitable simulation model may be generated immediately from the real implementation of the protocol, thus reducing the overhead in the development process. However, this process limits the protocol development to the features supported by the framework, as any advanced features would nullify the automated translation into a simulation model.

Some simulation tools provide another function; instead of relying on application models to initiate action, they allow the injection of live network data from outside sources into the simulation. If data may as well be emitted from the simulation to an outside application, the simulation may be integrated into a system and emulate a component's behaviour. Using this emulation method, the existing simulation components can be used to form a scenario that can be integrate into a test environment and emulate a sub-systems behaviour in such a way that the outside system can be tested.

Applied to the domain of distributed applications, the emulated sub-system would be the transport network including the terminals' mobility, as far as RANs are involved. The applications' data is routed through the emulation that mimics the effects of the RANs allowing a throughout evaluation of the distributed system. How this can be done in detail and what peculiarities have to been considered will be presented in the next section.

II. Network emulation

The emulation of the transport network for evaluation purposes may technically be accomplished in different ways, each of which has its own merits. Since a sound understanding of these methods is required in order to improve their capabilities, a short description of their operation is presented next.

The simple real scenario involving a RAN as depicted in fig. 2a will be used as an example. A mobile terminal executes a component of the distributed application, subsequently called client, and is connected using a RAN to a wired, but basically opaque transport network which in turn connects to a machine executing another part of the distributed application, subsequently called server. The client uses a real **N**etwork **I**nterface **C**ard (NIC) for the data transport that immediately connects to the RAN on the physical layer, the terminal itself is mobile and hence generates changes in the RAN access. These changes affect the NIC in terms of addressing, routing, available

networks etc. and may trigger a reaction by the OS or the client application. In terms of network quality the wired networks are in orders of magnitude better than RANs[1], and the RAN as the weakest link dominates the connection to an extend that allows to ignore the influence of the wired networks.

The most common approach to network emulation is the black-box method as shown in fig. 2b. Client and server are connected by wired networks to a device that emulates the transport network, hence called emulator. This may be either another computer running emulation software or a dedicated emulation hardware, but in both cases the data transport through the emulator happens completely transparent to the application – besides the emulated networks' effect on the data transport in terms of bandwidth, packet loss, delay, jitter etc. As far as the evaluated application is concerned, the data link through the black box can not be discerned from the direct link using a NIC. *NIST* [4] and *dummynet* [5] are examples for like emulators. The quality of the emulator determines the correspondence of the overall results with measurements taken using a a real network.

The quality of the emulator is not only decided by the realism of its models or of the used scenario, but heavily influenced by its performance. The device must be capable of processing the data passed to it that fast, that any delay etc. is the delay that is introduced due to the current emulation scenario. This basically implies that the emulator must operate under real-time constraints. Network emulators are usually implemented as discrete-event discrete-time simulation engines with a real-time scheduler. While a simulator advances the simulation time to the start of the next event after processing an event, an emulator stalls operation until an appropriate time has passed in the real world. Depending on the time it takes to process the events and the event intensity, the emulator may not be able to process an event fast enough to complete it before the next event is due — the emulator starts to *lag* compared to the real time, because the reaction to an event happens after it would have happened in a real network. Once lag sets in, the emulation will not deliver reliable results anymore, even if it happens to catch up the lag later on. Thus, lag detection is a must, and any emulation results produced with lag occurring must be carefully considered and, depending on quality and quantity of the lag, eventually discarded.

The black box approach using a dedicated system as an emulator allows it to dimension the emulator in such a way that it performance allows operation in the absence of any lag. However, this approach is somewhat limited when it comes to scalability. A separate machine and connection is required for every terminal involved in a scenario, increasing the required hardware and network connectivity.

With the ascend of virtualization and the wide availability of sufficiently powerful computers, it became applicable to network emulation as well, as lag-free operation of the emulator became possible for an increasing number of applications and scenarios. Using a single host computer, several **V**irtual **M**achines (VMs) are created that run the client or the server application, see fig. 2c. These VMs are connected through an emulator also run on the host computer, whether in its own VM or as a simple process is of no importance. Compared to black box emulation, no external networking is involved and only the copy performance of the host machine limits the throughput through the emulator.

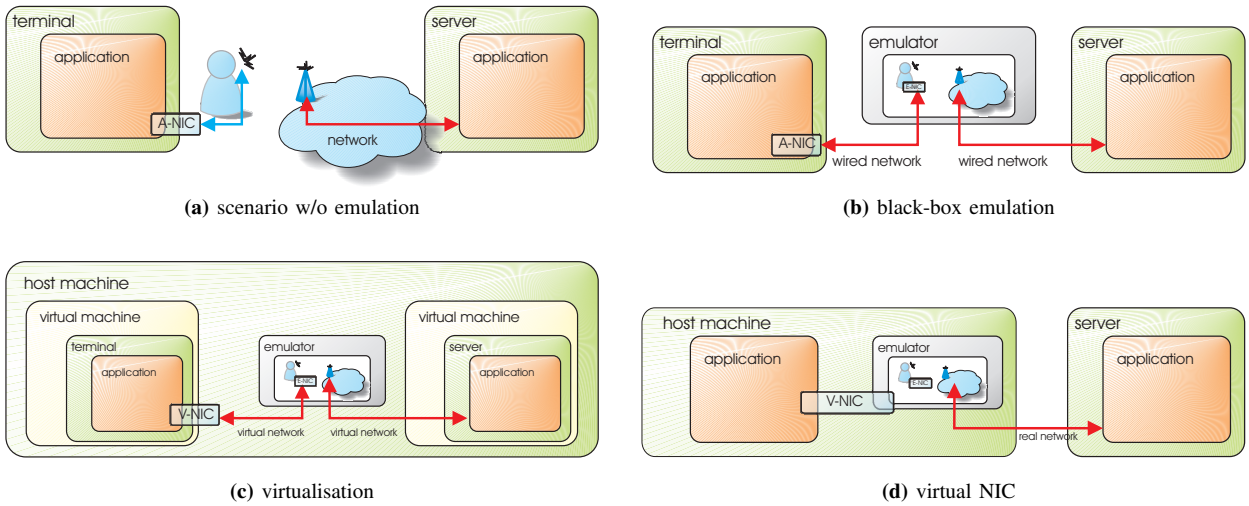
While virtualization eliminates the need for additional hardware and connection infrastructure from the black box approach, the demands on the host machine are however high, as any emulation must still be conducted in a lag-free manner. This applies to the VMs running the application components as well, it is not feasible to starve the applications of computing power in favour of the emulator, as lag here would as well negatively affect the quality of the results. However, as mobile terminals usually have very limited resources compared to desktop computers, while this limits the number of possible client VMs in a simulation scenario this approach is still applicable to many evaluation tasks.

These methods of emulation deal only with effects that the intermediate networks have on the data flow between client and server and are sufficient to investigate the applications' performance. However, the use of a RAN on the client side and running it on a mobile terminal introduces new effects that can not be evaluated with these methods. Using a RAN, the connection's quality varies in more aspects than just bandwidth, delay and error rate.

The terminal's mobility causes it to roam between **A**ccess **P**oints (APs) and RANs causing the terminal's address to change. These changes have extensive effects on the communication, as the application needs to adjust its used address according to the changes to keep the connection between client and server established. Using above methods, these effects happen within the emulator, thus are invisible to the application being evaluated. In order to evaluate applications dealing with these effects the address changes have to be exported from the emulator and made visible to the application, which could be implemented in several ways.

Black box emulation could be augmented by adding an agent on every client machine that is connected via a side channel to the black box and reproduce any effects to the NIC within the emulator on the NIC of the terminal. This approach either requires additional infrastructure for the side channel or accepts the possible interference of the side channel data with the emulation data if a shared link is used. Two qualitative issues are more severe than these quantitative issues. Firstly, this necessitates the additional agent to be installed and configured on every terminal. The agent needs to be implemented for every client OS and configured for every client system, so the big 'plug and emulate' advantage of the black box approach is significantly reduced. Further the installation of the agent

[1] i.e. bandwidth IEEE 802.11n 150Mbit/s (mode 7) vs. IEEE 802.3an/aq 10Gbit/s



(a) scenario w/o emulation

(b) black-box emulation

(c) virtualisation

(d) virtual NIC

Figure 2: Emulation methods

on the terminal may interfere with the application, thus disturb the system and distort the results. Second, due to the propagation of the effects from the emulator to the clients over the side channel, there is a delay between the incidence in the emulator and the reproduction of the incidence at the terminal. During this time, the NIC at the terminal and the corresponding endpoint in the emulator are out of sync, and all data in transport will be handled differently in the emulation than the client application initially expected it to be. This fact severely limits the applicability of an augmented black box emulation, as it hinders the applications to react immediately on the effects of an address change and may seriously distort the emulation results.

Transferring this approach to virtualization minimizes the delay between emulation and clients, as no physical network is involved in the side channel. Still, an agent is needed inside each VM to reproduce the effects happening inside the emulator at the local interface, as the information needs to moved across the VMs' border. Thus, the simple transfer of the emulation setup on a single host computer does mitigate the effects, but not solve the issue.

Considering the challenges for the emulation of RANs beyond the plain data transfer, and the use of VMs to facilitate the emulation process, the idea to use some of the techniques used to implement VMs directly for simulation purposes was born. Each VM creates virtual network devices, short V-NICs, that the host OS uses to pass network traffic to and from the VM as if it was a machine connected to a real NIC. The VM then uses some proprietary mechanism to relay the data to its guest OS, usually done by providing a virtual version of a common, real network device that can be accessed by the corresponding device drivers of the guest OS. By integrating this technique into the emulator, the emulator itself would be able create V-NICs that could be used to inject into and export data from the emulation. This can obviously be combined with virtualization or the separation of components as in the black box method. This so called V-NIC emulation method is depicted in fig. 2d, where as an example the server application is placed on a separate machine to reduce the load on the emulation host.

Every V-NIC is under the complete control of the emulator, and no additional means are necessary to convey changes made by the emulator to the NIC. All tools and interfaces the OS provides to access real network devices are available to inspect and manipulate the V-NIC. Those clients that can be run on the same OS as the emulator can be run natively on the emulation host and use the V-NIC instead of a real NIC without any difference. Obviously they are immediately exposed to any changes made to the V-NIC's status by the emulator, which warrants accurate emulation results. Further, for those clients no additional overhead for the use of a VM is required, reducing the load on the emulation host. General interference between the emulator and clients running on the same host, i.e. without the separation in individual VMs, is limited to resource consumption. Therefore there should be no difference to the virtualization method, where no VM, regardless of client or emulator, was to be starved from memory or processing power. Clients that can not be run on the same OS can still be placed inside a VM and connected to the emulator, but with the previously described limitations.

The V-NIC approach trades the flexibility of using VMs off for the improved performance and the easy and synchronous export of effects inside the emulator to the outside. The next section describes the additions to *ns–2* that have been made to facilitate the emulation of RAN for the use as a tool in the development cycle of mobility aware applications.

III. Extending *ns–2*

ns–2 is a network simulator that doubles as a network emulator. It implements a real-time event scheduler as well as components for network traffic injection from outside sources. Specifying these components as user

agents and scheduler in a simulation scenario results in an emulation scenario. Hence, simulation and emulation are nearly identical, and every issue with network simulation likewise affects the network emulation.

To simulate wired-cum-wireless scenarios, *ns–2*[2] uses a hierarchical address space, in use and notation similar to IP addresses, but with three parts instead of four, denoting *domain*, *cluster* and *node* level. The wired and wireless world is usually separated at the domain level, every AP is located as well in the wired as a wireless domain, usually defining its own wireless cluster. All mobile nodes are configured at set up time with the address of their respective AP, called *base station* in *ns–2*. Since the routing algorithm for hierarchical routing uses the first node it encounters to connect across clusters, it is not feasible to have several APs serving the same wireless cluster, as all wired nodes will try to route data to all nodes in that cluster using a single static AP. Thus, mobile nodes outside of that AP's range, but within the range of another AP of the same cluster will not be reached since all data is routed to the wrong AP. The other way round, the mobile nodes are not able to send data once they leave the range of their configured AP, since all packets are addressed to that particular AP. Using the *ns–2* components for **M**obile **I**nternet **P**rotocol (MIP) can circumvent that issue at the cost of the additional overhead that MIP introduces.

To accommodate this, a DHCP-like [6] protocol called INFRA was implemented for *ns–2*. INFRA differs slightly from DHCP, as can be seen in fig. 3a and 3b. While DHCP is usually triggered by an external source, as e.g. changes on the link layer, this was not possible in *ns–2*, as the *ns–2* channel model and IEEE 802.11 layer do not provide for SSID handling etc. Therefore appropriate adjustments had to be made for INFRA.

As DHCP, INFRA implements a client and a server mode. A server agent, located at the AP, provides addresses to clients from a given address pool. However, to compensate for the lack of link layer events, APs periodically broadcast their presence using so-called *beacon*-packets, a procedure usually implemented in the IEEE 802.11 layers. Based on the incoming beacons the client agent selects, usually based on the highest signal strength, an AP and requests an address from it. Supposed there are free addresses in its pool, the selected AP offers an address via broadcast to the client, which the client installs and acknowledges, now using a unicast packet with valid source and destination addresses. The client nodes' states during their lifetime are shown in fig. 3c.

As soon as more than a single AP is in range and multiple beacons are received, the INFRA needs to choose which AP to connect to. A simple decision algorithm based on the euclidian distance between AP and mobile terminal is used. The distance is used as a simple approximation for the signal power of the corresponding AP to simplify the calculation. INFRA however does not simply choose the nearest/strongest AP in reach, but applies a dampening algorithm, see fig. 4, in order to minimize AP hopping that may occur if APs are placed very close together. The algorithm uses two parameters for its operation. The first parameter T specifies a signal power above which a change of APs will unlikely result in increased performance, so if the current AP's signal is stronger, no change will be performed. The second parameter $G \in \mathbb{R}, G > 1$ specifies a minimum increase in signal power (and subsequently expected in performance) that must be gained by a change of APs; if the gain is lower, the current AP is retained.

The parameters T and G may be tuned, the default values where chosen as 85% of the maximum signal power and 10% required gain ($G := 1.1$). With this setting, no AP hopping occurred during the tests. This may be replaced by a more sophisticated decision procedure as required. The implementation of the exact procedure used by IEEE 802.11 is also possible, but overkill for the intended emulation purposes.

When switching between APs the client enters the roaming state indicating that it is still connected to one AP, but awaiting an address assignment from another. This state is left if either a new address is assigned, the request is declined, or a timeout occurs. In the latter cases the client stays connected to its current AP, while in the first case it gracefully returns its current lease to the old AP and assigns the new address. Obviously situations arise in which clients can not hand back their lease, e.g. by failure or simply leaving the AP's range. To cope with like situations, every AP has a watchdog timer for each lease that is reset with every activity of a terminal. Once the timer is triggered, the entry is purged from the leases' list and the address is free to be assigned to another node. This again is similar to DHCP, but instead of using explicit lease renewals this is accomplished implicitly. As default a timer of 5min is used, which is more than sufficient as the mobile terminals display usually nearly-continuous activity.

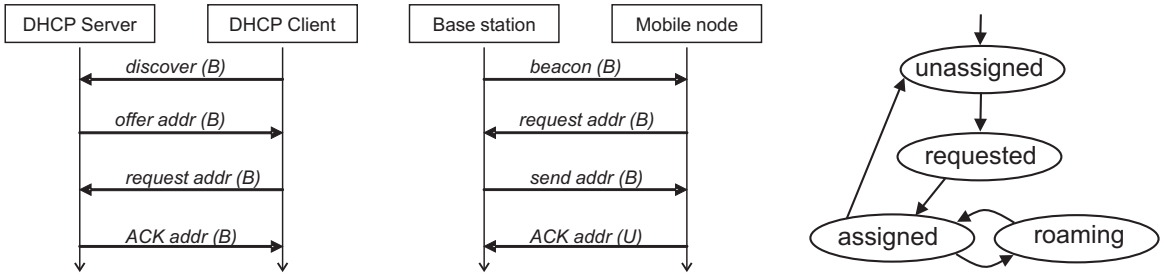
Compared to DHCP, the number and the complexity of messages is unchanged, but their order is reversed. Nevertheless, the periodically broadcasted beacons add an overhead to the channel. IEEE 802.11 sends beacons every .1s, the INFRA has a default period of .5s to reduce its overhead. DHCP operates client initiated, while INFRA operates server initiated. Without a beacon, no INFRA-client will try to request an address, as opposed to

Input: current AP, other AP considered
Output: the 'better' AP of the two

if *sigPower(current AP)* $>T$ **then**
    return *current AP* ;
if *sigPower(other AP)* $>G$ * *sigPower(current AP)* **then**
    return *other AP* ;
else
    return *current AP* ;

Figure 4: Dampening algorithm of INFRA

[2]All modifications presented in this paper where done using *ns–2* version ns-2.29.3.



(a) DHCP **(b)** INFRA **(c)** Node states in INFRA

Figure 3: Dynamic address assignment; (B = broadcast packet, U = unicast packet)

DHCP-clients. In a nutshell, the INFRA is an equivalent replacement of DHCP for simulation purposes with an identical message complexity — as long as the periodical broadcasts by every server can be neglected. Assuming the default period, a generous packet size of $500\mathrm{B}$ and a bandwidth of $10\mathrm{MB/s}$, this sums up to $\frac{1}{.5\mathrm{s}} * 500\mathrm{B} = 1000\mathrm{B/s}$ or approximately $1‰$ of the total bandwidth per AP in the region, and is thus negligible.

A notable challenge during the development was the fact that the *ns–2* packet generation mechanism creates new packets with the source address set to the node's set-up address instead of its current address. As any AP for obvious reasons does not forward packets received on the wireless interface with a source address outside its locally managed address space, all packets are dropped unless further action is taken. To solve this problem, the INFRA agent at the client rewrites the source address of every sent packet, unless its broadcast/unknown, to the node's current address. While this warrants the desired behaviour in all examined cases, this could still be a source for unexpected effects depending on the emulation scenario and especially the use of third-party components.

Any association and therefore address changes of a node are important events in terms of network emulation, as they widely effect the data transport. To forward these effects to other *ns–2*-agents, the INFRA has an event notification service based on the observer-pattern. Any component subscribing to the service is notified of address changes and may thus react accordingly. Notification on loss of connectivity could be implemented using the same mechanism but has not by now, as it was not required for our purposes and the discrimination between a transient (and thus ignorable) or permanent loss of link is non-trivial.

This mechanism is directly meshed with a new *network object* that was designed to provide V-NICs from within *ns–2*. Network objects are the components in *ns–2* responsible for the injection and export of network data into and from the emulator. Inspired by Mahrenholz et.al. [7] work that enabled *ns–2* to successfully access the V-NICs provided by virtual machines and thus introduced emulation using virtualization to *ns–2*, the inverse approach was taken here. A new network object was implemented allowing *ns–2* to create virtual network interfaces based on the *vtun*-facility of the Linux kernel [8]. These V-NICs may be used like any real NIC by an application, however instead of sending the data over a real network, the data is passed to the application creating the V-NIC — *ns–2*. Using the NIC level as an interface, the network emulation is performed completely transparent to the application, while the emulator may still control the NIC.

Currently, the `TAP` network object controls the IP settings of the V-NIC. Configured with an IP address range, it reacts to every "new association" event signalled by the INFRA on that node by cycling the V-NIC's address to the next in the range thus exposing any outside application to the challenge of dynamic addresses. Due to the internal structure of the `TAP` network object, the internal and external address change happen completely synchronous, eliminating any delay between the internal and external effect.

Within *ns–2*, so called *agents* are used to simulate the highest level of events, i.e. application or user behaviour. For emulation purposes, special network agents complement the network objects situated at the nodes. They receive data packets from the network object, wrap it for transmission across the simulated network, and forward the contents of any encapsulated packets received from the simulation to the network object. Network agents are commonly pairwise associated with one another; each agent forwards the encapsulated data to the node hosting its associated agent. Dynamic node address assignment and thus mutating addresses of the mobile nodes directly results in lost connections between the network agents, as the target's node address is stored during the set-up phase. The effects of changing addresses are the same within the simulation as in the real world.

For the purpose of emulation it is not desired to have the emulation deal with these effects, but only the outside application. Inside the emulation this problem was solved by implementing a modified `TAP` agent, that forwards the data not to the address its associated node was given during set-up, but to its current address. Solving this issue in a real network requires some dedicate protocol, however in the simulation this can be simply done using an artifice. *ns–2* nodes are identified by a network address as well as an internal node ID. A modified `TAP` agent was implemented that stores its associated node's ID instead

of its address and simply uses the ID to look up the node on the internal node list and retrieve its current address. This way, the agents will continuously forward the encapsulated packets to the appropriate node without facing any effects caused by the address changes.

For testing purposes, the ping agent was modified to address its replies not to the stored address, but to the source address of the request packet, which is a much more realistic behaviour than its current implementation. Further, this way a single ping agent can respond to requests from multiple sources. Using the ping agent in combination with INFRA the effects of dynamic addressing can be observed and evaluated in pure simulation mode, which was used during the development of INFRA.

For pure emulation purposes it would have been easier to skip the mutation of the addresses of any nodes with in the emulation but only to export the effect to the outside V-NIC interface. However, this would neither allow proper routing within the emulation to mobile nodes roaming across APs, nor would the system be reusable for simulation purposes to evaluate protocols dealing with dynamic addresses within *ns–2*. The additional effort necessary for the detailed implementation thus pays of in the long term.

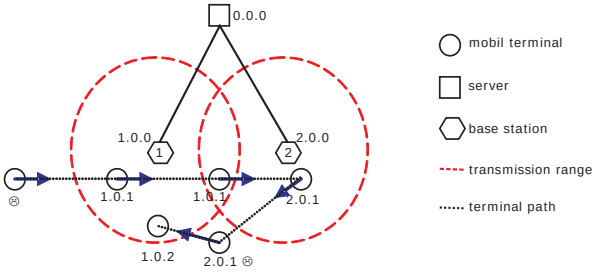


Figure 5: Example scenario

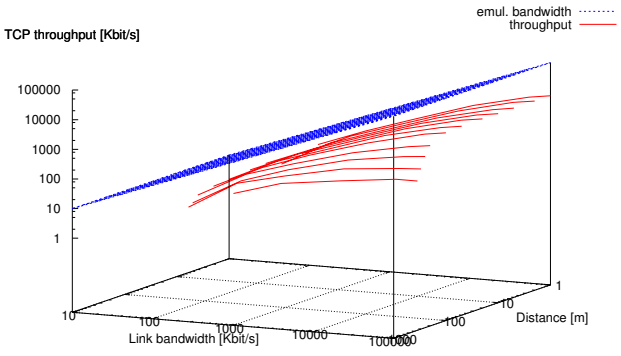


Figure 6: Wireless calibration measurements, note logarithmic axes

IV. USAGE AND EVALUATION

In order to demonstrate and evaluate the new components, a very simple scenario is created for emulation. It consists of two APs connected via wired networks to a server node and a single mobile terminal. The positioning is shown in fig. 5, the APs are arranged in such manner that their communication regions overlap. The simulated area is assumed to be plain and free of obstacles. The server's position is arbitrarily chosen for a nice arrangement, as it has no influence on the simulated network. The APs are connected to the sever over a $120\mathrm{Mbit/s}$ wired link without delay, so that this link will not influence the emulation.

The wired domain uses addresses starting with 0, and both APs are given their own domain as well in order to have the hierarchical routing of *ns–2* computing the proper routes. Within their domain, the APs have node address 0 for their wireless interface and hand out node addresses starting with 1 up to 19. The address ranges are chosen rather small to keep the hierarchical routing's routing tables at bay, as they are quadratic in the number of nodes per cluster. As *ns–2* is very sensitive to faulty addresses, it is important to correctly synchronize the setup of the hierarchical addresses with the INFRA agents, as having these hand out illegal addresses will at best cause the emulation to simply crash, or at worst silent faulty routing due to routing table overflows that unknowingly falsify the emulation results.

During the development of the *ns–2* extensions this scenario was used in combination with the modified ping agent for verifying the new components' behaviour. As in all tests, the active agent is located at the mobile terminal while the agent located at the server merely reacts to incoming requests. After these tests rendered the INFRA functional, the emulation extensions where evaluated using the same setup, but with a real ping being send from the mobile terminal to the server. The components where arranged as shown in fig 2d, with terminal and emulator being hosted on one machine and the server on another. Host and server machine are stock PCs using an AMD Athlon 64 3200+ CPU with $1\mathrm{GB}$ RAM and connected by a dedicated $100\mathrm{Mbit/s}$ Ethernet line undisturbed by cross and control traffic. The separation helps balancing the load, and simplifies the routing on the host machine, as the Linux kernel transports data packets between local NICs through the kernel. Rerouting these through the emulator is rather complicated and error-prone.

The following test were performed using *ns–2*'s IEEE 802.11 network model using two-ray-ground radio propagation model. While the stock IEEE 802.11 model is not up to date with the current extension of the standard[3], we still stick with it for its widespread use. In order to simulate higher bandwidths, the data rate, bandwidth, and basic rate parameters are simply increased with a data rate equalling the bandwidth and a basic rate of a quarter of the bandwidth. Under this assumptions, calibration measurements of the emulation where performed using bandwidths of $9.6\mathrm{kbit/s}$ (GPRS) up to $108\mathrm{Mbit/s}$ (IEEE 802.11n) in distances of $1\mathrm{m}$ up to $400\mathrm{m}$ using *iperf* [9] as a measurement tool. The measurement is the average throughput over a period of $15\mathrm{s}$. The results are shown in fig. 6 are as expected; for all bandwidths the measured

[3]There are several projects working on this problem, i.e. Mathieu Lacage (see `http://yans.inria.fr/ns-2-80211/`).

throughput is lower due to protocol overhead. For high bandwidths the difference grows, as the protocol parts using the lower basic rate and the constant synchronization times affect the results adversely. The throughput drops with higher distances, as more bit errors etc. occur. For low bandwidths at high distances the throughput is so low that iperf was not even able to conduct the TCP three-way-handshake and hence produced no measurements. During these test, the emulator reported no single occurrence of lag. This results prove that `TAP` network object and `TAP` agent operate as designed, that the performance of the *ns–2* emulation even on stock hardware is sufficient to route even current WLAN traffic and that the practical throughput of an $108\mathrm{Mbit/s}$ link is lower than the bandwidth of the dedicate link between the emulation machines.

The next function to be scrutinized is the export of emulator internal address changes to the V-NIC. In its current implementation, this process stops the emulator during the export in order to guarantee synchronous behaviour of inside and outside interfaces. For measuring a modified version of the `TAP` network object was implemented that simply times the export process and aborts the simulation. Repeated runs of this procedure resulted in an export time of $3.90 \pm 0.02\mathrm{ms}$, which is small compared to the delays introduced into by the emulation and therefore promises emulations without lag.

In order to test the INFRA three mobility patterns for the mobile terminal are used. In all cases, the terminal velocity was assumed to be $5\mathrm{m/s}$. The first pattern is a movement on the axis connecting the two APs starting at the edge of AP 1's transmission range and ending on AP 2's range. Using this pattern, the behaviour of the INFRA with a 'normal' handover was verified, including the delaying effects of the dampening algorithm. The second pattern is a linear movement on the perpendicular bisector of that axis. As the terminal is now constantly equidistant from both nodes, this stresses the dampening algorithm. It behaves as expected, once an AP is selected it is retained. At higher terminal velocity the AP may be changed depending on the interleaving of the beacons emitted by the APs. For the third mobility model, the mobile terminal moves on a path that exposes it to all four possible states of RAN availability: no access, access to AP 1 only, access to both APs, and access to AP 2 only. Afterwards, the triangular part of the path is cycled. This path combined with the (initial) address states along it is shown in fig. 5. This simple setup is useful to examine the behaviour of protocols with focus on the exact moments of changed availability. For long term evaluation, a setup using a hexagonal pattern of APs on a rectangular plane is more useful. Depending on the ration between AP spacing and AP range, as well full coverage of the area with a varying degree of overlapping of the transmission domains as incomplete coverage can be simulated. While this is a synthetic scenario, the right spacing parameters in combination with a suitable mobility generator produces quite accurate long-term results, as real world RANs expose a similar structure.

Increasingly often applications are using multiple concurrent RANs on a single mobile terminal, i.e. GPRS, UMTS and WiFi. This addition to *ns–2* is also suitable for such scenarios, however the application is not straightforward. Using multiple interfaces on a single *ns–2* node will cause *ns–2* to select an interface internally, but usually this is exactly the decision that should be made by the tested application. For such scenarios, a multiple *ns–2* nodes must be used to represent a single mobile terminal. Every node of this group has a single RAN uplink representing one of the composed terminal's interfaces. Being located on separate nodes, *ns–2*'s internal mechanisms will not interfere with the choice of the currently used RAN. Obviously all nodes of the group need to maintain the same position with in the simulation's topology and be move synchronously. That however is an easy task since it simply involves using the same movements for all nodes of a cluster. Further, the cluster sizes are expected to be in the range of single-digit numbers, as in practice the number of concurrently used RANs is limited by the device's size and power consumption. Considering that the number of emulated mobile terminals is anticipated to be reasonably low, the overhead introduced by using a group of nodes to represent a single mobile terminal will therefore be acceptable.

V. APPLICATION

The V-NIC emulation approach using the *ns–2* extensions was used to evaluate middleware performance if used from a terminal connected via a RAN. As described in section I, the comparison of middleware platforms using simulation is rather time consuming, as models for every platform and application need to be generated. Hence network emulation is especially attractive for the evaluation of middleware applications and a good example application of the enhanced *ns–2* emulation facilities.

For the evaluation, a simple middleware component called *loadserver* was created with an interface as shown in fig 7. The loadserver simply accepts requests containing an arbitrary number data bytes as parameters, delays $time\_ms$ ms to emulate processing of the request and returns a result with $resultsize$ B of dummy data. There are two modes of operation, the naive synchronous approach that blocks the caller while processing, and an asynchronous approach that requires explicit retrieval of a result by the caller using a token that has been returned on dispatch of every request. Results and tokens are provided with a local time stamp to support timing of calls on the client side. Depending on the client implementation various middleware applications can be emulated using the loadserver.

Selecting ORBit2 as ORB [10], the naive approach, and the emulation scenario depicted in fig. 5 a client was used to make 16B sized requests with no computation time and a result size increasing from 16B to 1MB. 25 passes were made and the data shown in fig. 8 collected. The overall outcome is not as interesting as the two significant

mavericks at pass (2kB/20) and (4kB/9). Both readings are several orders of magnitude higher than the readings of the other passes. This is an effect of dynamic NIC addresses on the clients side. Due to the interleaving at exactly this two points an address change occurs in such a manner, that the TCP connection between client and server has been established but is idle. During the idle period, caused by data processing on the client or the server side, the address change occurs. The NIC receives a new address and all data destined to the old address is discarded by the OS. This way the call makes no progress and stalls until a timer ends the connection, in this experiment the duration equals nearly a complete movement cycle. During this period all further changes go unnoticed, and any chances to reestablish the connection are not used.

This property of TCP has a severe effect on the performance when used over RAN. The two interruptions are below 1% of the calls, but have a combined runtime of 20% of the total runtime of this experiment. Up until Version ORBit2.17.3 of the ORB, released early in 2007, there was no parameter to influence the hold time for idle links and the user had to implement his own timeouts to be able to mitigate this problem. This obviously shows that the effects of RANs in distributed applications are far beyond the direct effects on the channel, naming bandwidth, delay, jitter, and packet error rate.

Similar experiments where conducted using JAVA RMI and web services, which was due to the emulation setup merely a question of implementing the appropriate client and server application. The network emulation had not to be changed, nor will it need to be changed for any future application. The performance of other middleware systems differs from CORBA's performance, especially with web services as XML data encoding is much less efficient than the binary representations of CORBA and JAVA RMI. However, the general behaviour is the same; the mavericks caused by address changes have a different extend due to different timeouts, but still occur.

To improve the performance of middleware applications in RANs, a sensor architecture was designed and

```
interface loadserver {
   struct structData {
     sequence<octet> buffer;
     long timestamp;
   };
   struct structToken {
     long id;
     long timestamp;
   };
   structData work( in long time_ms,
     in structData workdata, in long resultsize);

   structToken work_async( in long time_ms,
     in structData workdata, in long resultsize);

   structData result_async(in structToken token);
};
```

Figure 7: simplified loadserver interface definition

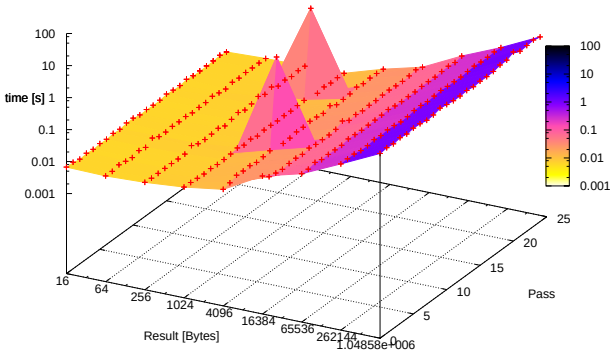


Figure 8: Results of CORBA experiment; 16B requests, 0ms processing time, 11Mbit/s RAN

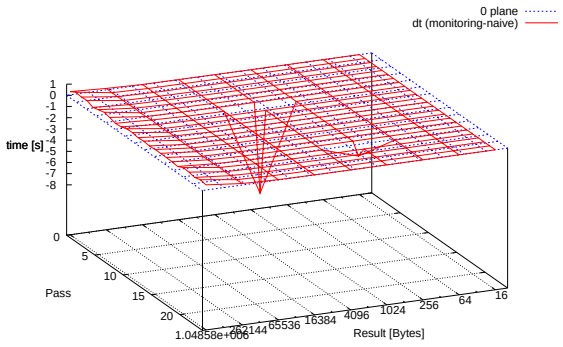


Figure 9: Comparison of naive and monitoring CORBA application

implemented that continuously monitors the NIC at the terminal and notifies the application of any significant changes. Upon i.e. a new network address the application can immediately take appropriate action, as cancel pending calls and repeat them using the new network address. Obviously, testing is vital for such applications, as the overhead for the monitoring must be balanced against the delay before changes are detected. Testing this using an identical emulation set up as for the naive CORBA application results by chance with mavericks at the same positions as the CORBA experiment, facilitating the comparison widely. Fig. 9 shows the difference between the results of the naive implementation and the implementation with monitoring. Overall, the implementation with monitoring is a little slower than the naive implementation due the monitoring etc. overhead. At the point of the address changes however, the monitoring implementation recovers much faster than the naive implementation. While the overall runtime is roughly the same for both approaches, there is obviously much less jitter in the response times of the monitoring implementation, making it much more suitable for interactive or other time sensitive applications.

The effects of call density versus terminal speed and therefore the average time spend at the same AP are also very notable. These influences can be evaluated by setting different topologies and node mobility patterns up within the emulation. All outside components do not require adjustment. This makes e.g. automatic node movement generation via e.g. CANU [11] possible and

the use of discretionary RANs in application testing. Again, this could be tested without any changes to the emulator, the setup or the application to accommodate the emulation setting. This demonstrates the power of network emulation implementing V-NICs as a tool for the development of distributed applications.

VI. Further development

The Linux kernel's vtun device is designed to behave like a virtual NIC to a wired network and was intended to facilitate network data tunneling using user space applications, e.g. openVPN featuring data encryption. Therefore only wired networks' properties are reflected by the driver and exported to tools and applications. The kernel uses the so-called **W**ireless **E**xtensions (WE) [12] for access to and managing of wireless NICs. Thus, to export wireless properties from within *ns–2* , an enriched version of the vtun interface is required, a *wtun*[4] device that implements the WE API. This way all existing tools and procedures for the management of wireless links may be used on links emulated by *ns–2*, and allow an universal use of the emulator as tool not only for application but also for OS tool development.

The vtun device driver has basically two interfaces. First, a network device interface that is used to create the virtual device and used by outside applications, and second a character device interface that is used by the application creating the virtual interface for reading and writing data. The vtun device driver simply passes the data between these two interfaces. To handle the status information that is provided by the WE, the wtun driver uses some local data for every device instance to store the complete status information. The WE interface merely accesses this information in in way that hardly differs from the access to real wireless NICs. While a real wireless NIC collects the status information handled back during its operation, the wtun device extends the vtun driver's control interface on the character side with get and set operations to all wireless status information. This is done using driver-specific `ioctl`-calls and gives the application creating the virtual device complete control over the network side's appearance to tools and applications.

The implementation of the wtun device driver is not completed by now, as functions are implemented on a need-to-have basis. The WE interface implementation is finished in all major aspects, however the complete encryption functionality has been skipped as it deems unnecessary for emulation purposes — and a virtual interface that has no physical layer that would allow eavesdropping to begin with. Notable not implemented, but desirable is the capability to monitor the signal strength of every designated node in range instead of just the AP. The WE API is evolving to make more information and functionality available, and limiting the functionality of the wtun device driver to the most commonly used functions becomes necessary.

The development of the wtun device driver progresses in parallel to the development of the *ns–2* `TAP` network object , which is to use the new facilities to export more information about the emulation status to the outside world. The WE in the `TAP` network object may be turned on or off by parameter, so the `TAP` network object may be still be used to emulate wired links without the overhead of maintaining the wireless status. There are several design issues that need to be solved for an easy and efficient export of the WE status. While returning the current signal quality, nodes in reach etc. is a rather simple task on a real device driver, getting it from the emulator is not as easy. The wtun device driver has no means to directly access the `TAP` network object and thus *ns–2* to retrieve the information. This means either a push or a pull approach needs to be implemented. An update for every packet send or received by the `TAP` network object is the straight-forward push solution, however consisting of numerous `ioctl`-calls it imposes a significant overhead and does not update the information without any network activity. Currently the `TAP` network object is implemented in a way that it periodically, every .5s by default, pushes the current virtual status into the wtun device. This causes obviously a small overhead if the status information is not needed and a delay of up to .5s between the data and the real status due to the update period. This defeats the idea of synchrony that initial drove the idea of incorporating V-NICs into the emulation. Should the tradeoff for the sake of efficiency not be acceptable for an emulation setup, it requires next to none effort to additionally trigger an update for every packet transmitted and combine it with the timer. This way, in periods with high activity the status is updated constantly providing the desired precision, while periodic updates during idle time improve efficiency.

A pull approach using a signal handler could be implemented. Triggered by a request to the V-NIC, it would pull that current information from the *ns–2* `TAP` network object and deliver it to the application hence create no overhead and always deliver the latest information. However, this is difficult to implement and it is unsure, whether signal handlers may be used within *ns–2* without interfering with other components, so for now the push approach is sufficient.

The `TAP` network object can gather the diverse information about the simulated wireless link using several methods from within the emulator. Some information, e.g. SSID, have no direct equivalent in *ns–2*, but can be modeled by other means, e.g. the channel concept within *ns–2*. Additions to the `TAP` network object translate this *ns–2* internal concepts into data compatible with the wireless extensions, in this case using the name of the channel object as SSID.

The integration of the Infra with other *ns–2* components, especially user designed extensions, has its pitfalls. Experiments using the contributed GPRS link layer [13]

[4] 'w ≡ v++'

and the UMTS link layer [14], [15] failed as these layers use the nodes' addresses to reassemble packets. Reassembly and thus packet transport fails as soon as these addresses change, rendering these extensions incompatible with INFRA. These issues may be solved after the current overhaul of *ns–2* leading to *ns–3*.

VII. CONCLUSION

Only by removing the need for real RANs and real terminal mobility test runs can become a feasible step in the development cycle of distributed applications. This can be accomplished by network emulation making it an important tool for the development of distributed applications. Terminals connected by RANs require active components to mitigate the effects of ever changing connectivity, which may be part of theOS or the distributed application itself. In order to evaluate these components, the plain classical black box approach to network emulation is no longer sufficient, as information about the emulated network's status must be made available to the outside application. This can be accomplished by the use of V-NICs created by the emulator.

The *ns–2* software was enhanced by several components. The INFRA simulates a DHCP-like dynamic address assignment, the `TAP` network object creates and manages the virtual NICs and the `TAP` agent ensures proper routing of data through the emulator in presence of dynamic addresses. `TAP` network object and INFRA combined export the internal changes to the externally visible V-NIC. Internal and external effects happen synchronously, ensuring precise and predictive behaviour of the emulator. Tests confirmed these dynamics did not affect the emulator's performance to an extend that would introduce lag into the emulation process and thus invalidate any results. However, the use of dynamic node addresses is not compatible with all *ns–2* components, especially extensions provided by other users.

Since the Linux kernel's device driver for virtual NICs, vtun, only exposes the APIs for NICs to wired networks, an extended device driver called wtun was implemented that includes the WE API as well. Using this driver and an appropriately enriched `TAP` network object inside *ns–2*, all major status information of an emulated RAN inside the emulation is accessible on the V-NIC as well. While not all information is currently available, i.e. the complete cryptography functionality was excluded, the wtun driver and the `TAP` network object can easily be extended on demand.

Several experiments involving middleware applications where presented as an example application of network emulation using V-NICs during the development cycle of distributed applications. The results show the influence of the exported RAN effects on the application. Their influences on the data transmission is major, and could not be observed without the new export facilities. Network emulation using V-NICs allows an easy evaluation of distributed applications using adaptive algorithms and complex middleware without the overhead of model construction that would be required for simulation.

REFERENCES

[1] K. R. Fall, "Network emulation in the vint/NS simulator," in *ISCC*. IEEE Computer Society, 1999, pp. 244–250. [Online]. Available: http://csdl.computer.org/comp/proceedings/iscc/1999/0250/00/02500244abs.htm

[2] X. Chang, "Network simulations with opnet," in *WSC '99: Proceedings of the 31st conference on Winter simulation*. New York, NY, USA: ACM Press, 1999, pp. 307–314.

[3] X. Zeng, R. Bagrodia, and M. Gerla, "Glomosim: A library for parallel simulation of large-scale wireless networks," in *Proceedings of the 12th Workshop on Parallel and Distributed Simulation (12th PADS'98)*, B. Unger and A. Ferscha, Eds. Banff, Alberta, Canada: ACM/Society for Simulation (SCS), San Diego, May 1998, pp. 154–161. [Online]. Available: http://doi.acm.org/10.1145/278008.278027

[4] M. Carson and D. Santay, "NIST net: a linux-based network emulation tool," *Computer Communication Review*, vol. 33, no. 3, pp. 111–126, 2003. [Online]. Available: http://doi.acm.org/10.1145/956993.957007

[5] L. Rizzo, "Dummynet: A simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, Jan. 1997. [Online]. Available: http://www.iet.unipi.it/~luigi/dummynet.ps.gz

[6] R. Droms, "Dynamic Host Configuration Protocol," RFC 2131 (Draft Standard), Mar. 1997, updated by RFCs 3396, 4361. [Online]. Available: http://www.ietf.org/rfc/rfc2131.txt

[7] D. Mahrenholz and S. Ivanov, "Adjusting the ns-2 emulation mode to a live network," in *Kommunikation in Verteilten Systemen (KiVS), 14. ITG/GI-Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005) Kaiserslautern, 28. Februar - 3. März 2005*, ser. Informatik Aktuell, P. Müller, R. Gotzhein, and J. B. Schmitt, Eds. Springer, 2005, pp. 205–217.

[8] R. Breen, "VTun," *Linux Journal*, vol. 2003, no. 112, pp. 6–6, Aug. 2003.

[9] A. Tirumala, "End-to-end bandwidth measurement using iperf," in *SC'2001 Conference CD*. Denver: ACM SIGARCH/IEEE, Nov. 2001, national Laboratory for Applied Network Research (NLANR). [Online]. Available: http://dast.nlanr.net/Projects/Iperf

[10] G. Foundation, "Orbit2 – a corba 2.4-compliant object request broker," online, 2002. [Online]. Available: http://www.gnome.org/projects/ORBit2/

[11] I. Stepanov, J. Hähner, C. Becker, J. Tian, and K. Rothermel, "A meta-model and framework for user mobility in mobile networks," in *Proceedings of the 11th International Conference on Networking 2003 (ICON 2003), Sydney, Australia, September 28 - October 1, 2003*. New York: IEEE Press, Sept. 2003, pp. 231–238. [Online]. Available: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/INPROC-2003-15/INPROC-2003-15.pdfurl:http://canu.informatik.uni-stuttgart.de/mobisim

[12] J. Tourrilhes, "Wireless tools for linux," online, Feb. 2007. [Online]. Available: http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html

[13] R. Jain, "Extending the ns2 simulator for GPRS support," 2001. [Online]. Available: http://www.it.iitb.ac.in/~sri/students/richa-thesis.pdf

[14] A. Bjrsson, "Simulation analysis of RLC/MAC for UMTS in network simulator version 2," Dec. 2003. [Online]. Available: http://www.ep.liu.se/exjobb/isy/2003/3399/

[15] F. Eng, "New modules for ns-2," 2004. [Online]. Available: http://www.control.isy.liu.se/~frida/nsmodules

Tim Seipold is currently a doctoral (∼ Ph.D.) candidate at RWTH Aachen University of Technology Aachen, where he completed his Diplom (∼ Msc.) degree in informatics 2001. He worked as research staff and project manager on several projects during 2001-2006. His research interests distributed applications and P2P technology in RANs.