IKREmuLib: A Library for Seamless Integration of Simulation and Emulation

Marc C. Necker, Christoph M. Gauger, Sebastian Kiesel, and Ulrich Reiser

Institute of Communication Networks and Computer Engineering
University of Stuttgart, Pfaffenwaldring 47, D-70569 Stuttgart
{necker, gauger, kiesel}@ikr.uni-stuttgart.de

Abstract. The method of emulation is an attractive approach in order to evaluate the performance of communication networks. Compared to simulation, it is especially beneficial when real-world components, e.g., applications or protocols should be included. In this paper, we present the new emulation environment *IKREmuLib*, which allows for a seamless integration of both the emulation and the simulation domain by exploiting the same system model implementation. By doing so, we avoid the disadvantages of either approach while combining the benefits of both.

1 Introduction

In order to develop new or improve existing architectures and protocols for future and deployed communication networks, it is essential to evaluate their performance before they are deployed. Often, it is impossible to actually build a prototype of the complete system and make live-measurements. Likewise, it is often not feasible to test new algorithms in existing networks, since reliable network operation must be ensured. A common approach is therefore to develop a simplified model of the complete system and evaluate the performance by means of simulation.

The most commonly used concept for network simulation is the event-driven simulation [1]. The simulation may be done at different levels, i.e. at the packet level or at the flow level. In a packet level simulation, individual packets (e.g., IP packets) are generated by traffic generators and transmitted through the modeled network towards a traffic sink. In flow level simulations, a traffic flow is characterized by a time duration or a traffic volume, which occupies a set of resources for a certain period of time.

The main advantages of the simulation approach are the reproducibility of results and the possibility to easily explore a large parameter space. Moreover, it is usually easy to experiment with complex algorithms, since the simulation environment provides network status information, which may not be accessible as easily in a real system.

On the other hand, one of the problems is the effort it takes to model complex systems and protocols. Especially for the performance evaluation of access links, it is important to have fine-grained traffic models. This applies especially to sources with significant user interaction, like for example web traffic using the Hyper Text Transfer Protocol (HTTP). When looking at such higher layer protocols, the parameter space for their configuration is usually very large. Moreover, there are often diverse but important details, which may range from variations in the protocol implementation to differences in the actual traffic generation. This makes it difficult to model such traffic sources for simulation purposes. It may therefore be advantageous to include a real world component in the simulation environment.

Integrating a real world component, such as code pieces from a web browser, into a simulation is a very complicated task. An alternative and often very beneficial approach is to use the method of emulation, which basically combines the simulation domain with real world components [2]. In principle, with emulation, a simulation is enriched by an interface, which enables the simulation to communicate with real network components [3], turning it into an emulation. Additionally, some of the protocol components involved in the communication with these real network elements or the operation of the elements itself may require the emulation to be performed in real-time. One example of an emulation environment is the emulation facility of the Berkeley network simulator ns-2 [4].

To a certain degree, the method of emulation makes it unnecessary to model critical parts of the overall system. This not only simplifies the whole performance evaluation process, it also increases the credibility of the results. In a pure simulation environment, the quality and credibility of results may be a problem due to simplifications or flaws, e.g. in the traffic models, such as inadequate or inprecise models, or even unsuitable random number generators. This problem was studied in [5], where it was found that about 70% of all publications which base their results on simulations lack credibility.

The main problem with network emulation is the inherent timing error. In an event-driven environment each event is scheduled for one particular infinitely short time instant, and several events may be scheduled for the same time instant. Since the execution of an event is not completed in zero time, the emulation usually lags behind in its execution, resulting in a timing error. The effect of such timing errors was studied in [6] for certain scenarios and ns-2.

In this paper, we present an integrated simulation and emulation environment *IKREmuLib*, based on the event-driven simulation library *IKRSimLib* [7,8]. The main goal of our architecture is to allow a quick transition from simulation to emulation and vice versa with the same model and without any code modification. We achieve this by a strictly modular design of the emulation feature on top of the existing simulation library.

The design philosophy of the IKREmuLib is to intercept incoming IP packets, filter them according to a set of rules, and then pass them to the system model, where they are processed according to the system behavior, i.e., they are delayed or dropped. Finally, packets which are not dropped are sent out to the respective network interface. We achieve high timing accuracy by splitting the system into two threads, one of which being responsible for the model execution, the other one being responsible for capturing and time-stamping the IP packets.

The remainder of the paper is structured as follows. In section 2 we give an overview of the architecture and all relevant concepts of the IKRSimLib. We present the basic architecture of the IKREmuLib in section 3 and detail the implementation in section 4. The accuracy and performance of the library is evaluated in section 5, and the application to the emulation of a Universal Mobile Telecommunications System (UMTS) Radio Access Network is described in section 6. Finally, section 7 concludes the paper.

2 IKRSimLib Architecture

The IKR Simulation Library (IKRSimLib) [8] is a tool which is mainly used for event-driven simulation of complex systems in the area of communications engineering. It is deployed as a C++ class library and was first presented in [7]. Since the initial design the library has been continuously enhanced and improved. It has been successfully used for performance evaluation in various projects and publications from different areas of communication networks research, e.g. IP, ATM, photonic, mobile, automotive and signaling networks.

IKRSimLib concepts and components can be structured in three main areas: basic simulation support, modeling, and standard system entities.

- Basic simulation support includes concepts and components for event handling, simulation control, distribution-oriented random number generation, as well as for online statistical evaluation. For flexibility, this also comprises a parameter file parser and an output concept.
- Modeling is facilitated by the construction of hierarchical models from individual components and—for this paper essential—the standardized exchange of messages among those components. This message exchange employs so-called ports, which are used to define a generic external interface of a model component, based on a simple message transfer protocol.
- Finally, standard system entities like traffic generators, queues, servers, multiplexers, traffic sinks etc. are provided to ease model implementation.

Illustrating those concepts, Fig. 1 depicts a simple queueing theory node model comprising generator, single server queue, and traffic sink standard system entities. Simulation messages are passed from component to component using the ports and the message transfer protocol.

The simulation library is based on the principle of event-driven, discrete time simulation. Commonly, *physical time*, *simulation time*, and *real-time* are distinguished when characterizing simulation/emulation systems. Physical time is the time experienced by the system under study while simulation time is its representation in the simulation/emulation obtained from a bijective transformation. Real-time refers to time experienced by the simulation/emulation host system during execution. In general it depends on the performance of the host system.

In discrete time models, state transitions are the instances in time at which all system functionality is executed. Thus, at state transitions both physical as well as simulation time can advance according to the system under study while

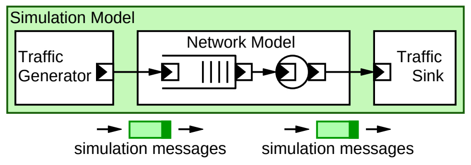

Fig. 1: Message based simulation

real-time advances continuously on the host system. In contrast, "idle time" between state transitions is bridged and no further processing, i.e., real-time, is needed for this.

For the remainder of this paper, we will not distinguish between physical and simulation time due to their equivalence via the bijective transformation. Also, we will use the term simulation time for the emulation as well as for the simulation case.

All currently pending events in the system are managed by a central calendar. An event loop in the simulation control takes care of the in-order processing of events in the calendar, i.e., every time an event finished processing it advances time and executes the next event as depicted in Fig. 2.

The state transitions are described by event objects. These events are characterized by their event time and the entity that will handle the event. An example of an event could be the arrival of a request at the generator or the end of service for a request in a service unit. In the former case, the processing of the event *request arrival* triggers construction of a simulation message in the generator as well as its transfer to the adjacent component via the generator's output port.

3 Architecture of the IKREmuLib extension

3.1 Functional Requirements

When making the transition from simulation to emulation or vice versa, we desire to be able to reuse existing system models without the need for extensive

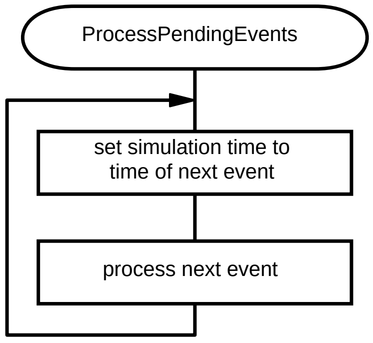

Fig. 2: Event processing loop

modifications. If, additionally, we can use the same program code of a system for both simulation and emulation without any modifications, we have obtained an integrated simulation and emulation environment.

For ease of use, we decided to implement the emulation's interface at layer 3, i.e., it intercepts network packets at the IP layer and delays or drops them as determined by the system model. This implies that the emulation host is able to act as a router for IP packets. It is therefore necessary to filter incoming IP packets according to a set of rules in order to determine whether they are meant to be treated by the emulation system, and which path they are supposed to take through it. Alike, it is necessary to route outgoing packets to the appropriate network interface and destination host.

For many applications, an emulation system has to satisfy real-time requirements in order to model the real system behavior correctly. This is for example the case if timer-controlled mechanisms or protocols, such as the Transmission Control Protocol (TCP), are involved. Strictly, this means that every module of the system model has to run as fast or faster than real-time. A relaxed formulation would only require the total transfer time of the emulation system to be real-time. In detail, the following inaccuracies can be distinguished:

– model imposed inaccuracies due to finite event execution times
– operating system imposed inaccuracies
– inaccuracies due to asynchronous packet arrivals

Model imposed inaccuracies are hard to alleviate. In fact, the only possibilities are to optimize the model implementation, use a faster emulation machine or to parallelize the model execution. Another approach is to pre-execute certain events. However, this is hard to realize as dependencies between different events and packet arrivals are often hard to determine.

Inaccuracies imposed by the operating system usually result from a multitasking environment. The only safe way to solve this issue is to use a real-time operating system, such as RTLinux [9]. As an initial step, we implemented our emulation library on top of a non-real-time operating system. The transition towards a real-time operation system could be done in the future without the

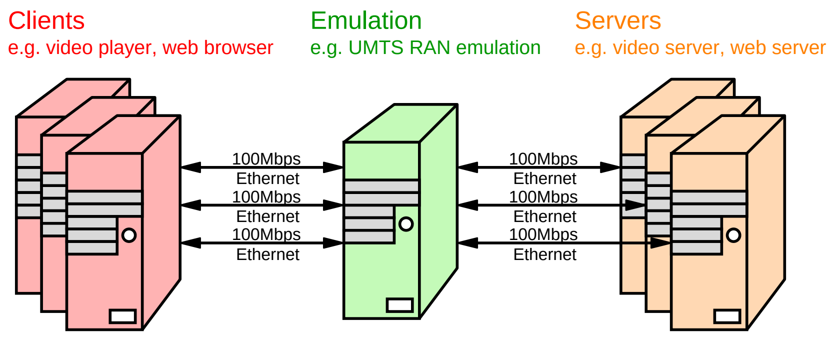

Fig. 3: Emulation setup

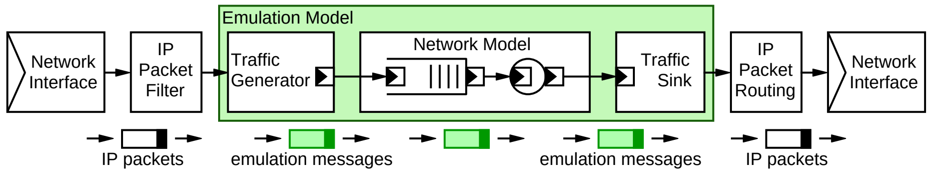

Fig. 4: Replacing the IKRSimLib generators with IKREmuLib generators

need for basic changes in the design of the emulation library. In section 5 we will show by means of measurements that we can achieve sufficient accuracy even with a non-real-time operating system.

Finally, inaccuracies due to asynchronous packet arrivals require an exact time-stamping to minimize timing uncertainty regardless of the current emulation state.

Fig. 3 shows an example of the basic emulation setup which we target at. The actual emulation system runs on the emulation host, which is a standard PC. One or several PCs running client and server applications, respectively, connect to the emulation PC via Ethernet. In general, the topology of the setup can be arbitrary, while the only constant is the central emulation PC.

3.2 Transition from simulation to emulation

As mentioned in section 3.1 we aim at a seamless and effortless integration of a system model into both simulation and emulation. From the system model point of view, a network interface primarily is a traffic source and a traffic sink. Starting from the simulation model introduced in Fig. 1, we will therefore replace the existing traffic generators and traffic sinks with new types of generators and sinks connecting to network interfaces.

This transition is shown in Fig. 4. The new traffic generator can logically be separated into two parts. The first part connects to the network interface and filters all relevant packets on the IP layer. The second part encapsulates the IP packets into emulation messages similar to those generated by conventional IKRSimLib traffic generators, which can directly be sent through the system model. Likewise, the new traffic sink can be separated into one part decapsulating the IP packet from a received model message and a second part routing the IP packet to the appropriate network interface. As each emulation message carries a reference to the corresponding IP packet, we can even access and modify the IP packet content at any point in the model. This also allows the creation of new IP packets and use the emulation system as a protocol endpoint.

3.3 Real-time synchronization

Figure 5 shows an example of a simulation timeline. The bottom time axis of the figure represents the simulation time, while the upper time axis represents the real-time which actually passes during the execution of events. While the execution of an event happens in zero simulation time, it may require a significant real-time period. For instance, event 2 cannot start processing in time, as event 1 has not been finished yet. Also, in simulation, whenever an event has been processed, the calendar immediately progresses to the next event. In contrast, in Fig. 5, the emulation is halted after the processing of event 2 has completed until the real time matches the simulation time t_3 for which event 3 is scheduled.

Both issues have to be considered when synchronizing simulation time with real time. They also highlight two critical problems that the synchronization of real-time and simulation time brings about when dealing with the arrival of new packets at network interfaces.

First, while the simulation is halted, we need to be able to detect the arrival of a new packet. In [4], an active wait loop was used, which continuously checks whether the real-time matches the time of the next scheduled event or a new packet has arrived at a network interface. This is a practical and very accurate approach, which is well suitable on a machine dedicated to emulation. However, it permanently consumes all available processor resources, which imposes problems to applications running in parallel, such as for example a visualization tool.

The second problem arises upon the arrival of a packet as is depicted in Fig. 5. The packet arrives while event 3 is being processed. As already mentioned, the creation of a simulation message within a generator is associated with an event, within which the message is instantiated and forwarded to the generator's output port. As the emulation is busy processing event 3, the packet arrival can only result in the generation of the indicated packet event as soon as event 3 is completed, i.e., at time $t_{p,1}$. It is obvious that this implies a significant timing error. Therefore, we need to capture the exact time instant of the packet arrival in order to be able to post the event as close to the time of packet arrival as possible, such as at time $t_{p,2}$ in Fig. 5.

In order to achieve this, we completely separate the packet reception and the model execution using a multi-threaded design. In this design, a *listener thread* is solely responsible for capturing and time stamping arriving packets, while a second *model thread* is mainly responsible for the model execution and transmission of packets. This concept is illustrated in Fig. 6. The listener thread connects to the network interfaces. After capturing an IP packet, it is filtered according to a set of rules. If not dropped by this filter, a time stamp is assigned

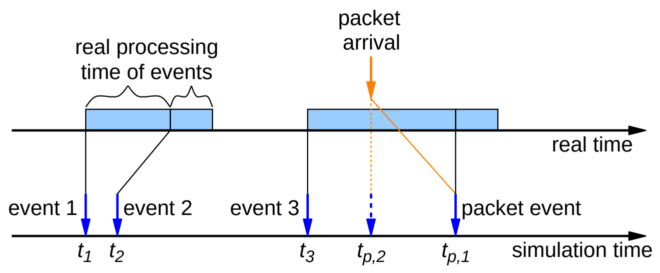

Fig. 5: Relation of real-time and simulation time

to the packet and it is placed in a queue. The packet arrival is signaled to the model thread, which creates a new event and dispatches the arriving packet to the corresponding traffic generator.

The transmission of an IP packet at the sink is less critical and can directly be done by the model thread. The kernel routing functionality is used in order to route the packet to the appropriate network interface.

4 IKREmuLib Implementation

4.1 Overview

In this section, we will detail the implementation of the IKREmuLib. We will first describe the framework used for our implementation in section 4.2. Next, we will describe the main loops of the listener and model threads in section 4.3 and 4.4, respectively. Afterwards, we illustrate the filter concept and dispatching procedure of packets to generators in section 4.5. Finally, we will elaborate on the synchronization between the two program threads in section 4.6.

4.2 Packet capturing and multithreading

The multithreaded design of the IKREmuLib is based on the Portable Operating System Interface (POSIX) threads [10, 11]. POSIX provides a powerful and easy-to-use interface for multithreaded programming. Besides the actual thread management, this includes support mechanisms such as mutual exclusion functionality (mutex) and conditional wait operations [11]. Virtually all UNIX derivates implement threads that conform to the POSIX specification, which makes it an attractive environment to use for our purposes.

As outlined in section 3.1, we require the emulation system to work as a router at the IP layer. Thus, in order to capture all IP packets arriving at the network interface and not only those destined for the emulation host itself, we need to access the network interfaces at the data link layer (i.e., at the Ethernet level) before any routing decision is performed. Packets are usually being captured in

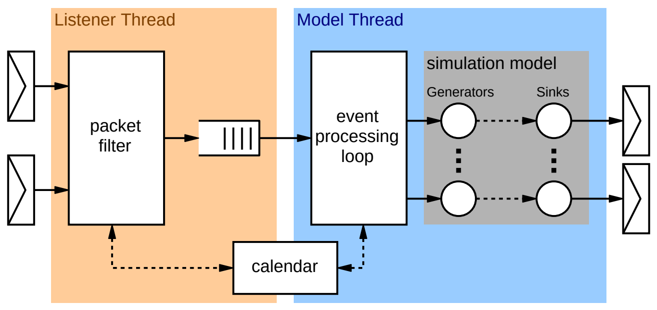

Fig. 6: Separation of emulation library into threads

the kernel-space of the operating system, from which they have to be copied into the user-space, where the emulation is executed. In order to minimize the overhead, it is desirable to filter the packets directly in the kernel-space and copy only relevant packets to user-space.

The regular UNIX socket interface is not an appropriate way to achieve this goal. Instead, various other possibilities exist, such as the BSD packet filter (BPF) [12] available for BSD UNIX, the Data Link Provider Interface (DLPI) for System V [13], or the Linux PF_PACKET interface. All of these approaches are dependent on the specific UNIX derivate they are used with. In contrast, the packet capture library pcap [14] is a publicly available open-source alternative, which not only provides kernel-space packet filtering but which also is available for practically all UNIX derivates. Thus, we selected the pcap library for our implementation.

4.3 Listener thread

In this section, we will detail the packet reception procedure within the listener thread. Figure 7 shows a simplified SDL diagram of the listener thread's main loop. A standard *select()* UNIX system call [13] is used to perform a blocking wait operation for the arrival of new data on one of several network interfaces. Upon the arrival of a new packet, it is fetched using the pcap library's *pcap_next()* function call. A timestamp is assigned to the packet and it is placed in the listener thread's output queue. The arrival of the new packet is signaled to the model thread using a condition variable and the *pthread_cond_signal()* function call [11]. Afterwards, the program flow returns into the waiting state with the *select()* call.

For the synchronization of the two threads, we additionally need a mutex to prevent certain conditions, which would lead to an illegal emulation state. The elements of this mechanism are shaded in Fig. 7 and will be detailed in section 4.6.

4.4 Model thread

The model thread is responsible for the actual model execution. Hence, it also performs the event processing loop described in section 2. This loop has to be extended in order to process events in real-time and be able to interact with the listener thread. Figure 8 shows the simplified SDL diagram of the model thread's real-time event processing loop. Again, the shaded mutex elements will be discussed in section 4.6.

We will describe the loop by assuming that there is an initial event in the calendar. First, the relative time to this event is determined and converted to real-time. If this difference is smaller or equal to zero, the event is immediately processed after all waiting packets have been fetched and dispatched from the listener thread's output buffer. The cleaning of the listener thread's buffer before the event processing is necessary to prevent the emulation from only processing events which follow up too closely on each other without dispatching new packets.

In case the relative time to the next event is larger than zero, the simulation

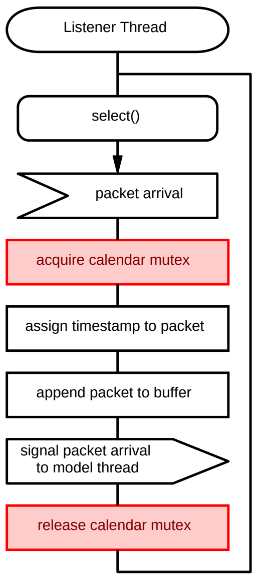

Fig. 7: Listener thread main loop

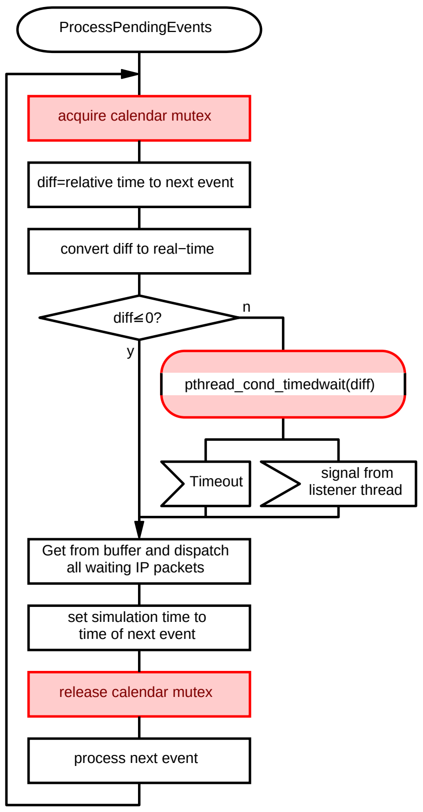
ProcessPendingEvents
acquire calendar mutex
diff=relative time to next event
convert diff to real−time
diff≤0?
n
y
pthread_cond_timedwait(diff)
Timeout
signal from listener thread
Get from buffer and dispatch all waiting IP packets
set simulation time to time of next event
release calendar mutex
process next event

Fig. 8: Real-time event processing loop

has to be halted until the real-time matches the simulation time of the next event. This is done in the right branch of Fig. 8 using the *pthread_cond_timedwait* function call. This call returns as soon as a given time period has elapsed or a signal from the listener thread has been received. In either case, an event needs to be processed. Hence, we will dispatch any waiting packets, set the simulation time to the time of the next event and process it.

4.5 Filtering and packet dispatching

In the kernel space, the pcap library filters those packets which are to be processed by the emulation library and ignores all other packets. Relevant packets are identified by an Ethernet destination MAC address equal to the emulation host's MAC address, but an IP address different from any of the emulation host's IP addresses.

Dispatching of packets to different traffic generators is done according to destination and source IP addresses. Additionally, the packets can be dispatched based on their TCP or UDP port numbers, if available. Each generator needs to be assigned at least a destination or source IP address, or both. Alike, a generator may also be assigned a destination and/or source port number. When dispatching a packet, the generator with the most precise match for the packet is identified, and the packet is forwarded to it.

4.6 Process synchronization

The multithreaded design as it was presented up to now, has the risk of resulting in an illegal emulation state. Consider the scenario shown in Fig. 9. The figure shows two time lines representing the real-time. The left time line shows those periods where the listener thread is being executed by the CPU, and the right time line those periods assigned to the model thread. A new packet arrives and is assigned a time stamp with value t_0 by the listener thread. Before it can be appended to the listener thread's buffer, the control is transferred to the model thread. The model thread processes the next event at time $t_1 > t_0$, since the listener thread's buffer is still empty. After another control transfer, the packet is appended to the listener thread's buffer. As soon as the model thread gets to the point where it fetches the packet from the buffer and posts the corresponding packet event for time t_0, this results in an illegal emulation state, as t_0 lies in the past compared to the current simulation time.

This situation can be avoided by introducing a mutual exclusive lock mechanism in the calendar. This calendar mutex is acquired and released as indicated in the SDL diagrams of Fig. 7 and Fig. 8. It is important to note that the *pthread_cond_timedwait* function offers the possibility to atomically unlock the mutex and wait for a possible signal from the listener thread. Alike, the mutex is re-acquired upon function exit. This interconnection between the signaling of the condition variable and the mutex is necessary to avoid race conditions [11].

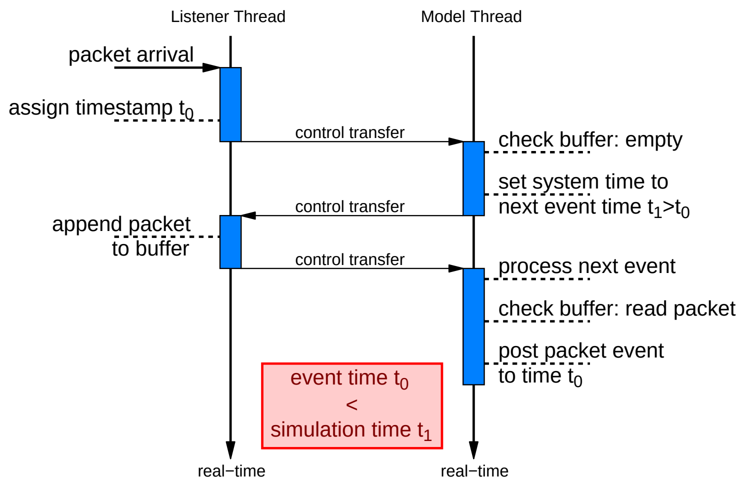

Fig. 9: Mutex scenario

5 Performance Evaluation

In order to evaluate the accuracy of the IKREmuLib, we performed measurements using a very fundamental system model. The measurement setup is shown in Fig. 10. The emulation PC hosts the emulation environment with the system model, consisting of an infinite server with constant service time T_D. Bulk UDP traffic was generated with a separate PC, with length L UDP-packets being transmitted at a rate of r packets/s. The transit time of the UDP packets through the emulation system was measured at the Ethernet-level using an Agilent Internet Advisor. In particular, we evaluate the absolute timing error of the emulation based on its complementary cumulative distribution function (ccdf) and its mean value. All ccdfs were obtained with a total of 500,000 transmitted packets per measurement.

For all measurements, we use standard Linux installations with kernel version 2.6. This kernel version provides improved process scheduling mechanisms compared to older Linux kernels. This is of advantage for minimizing the operating system imposed inaccuracies described in section 3.1.

Ideally, the measured delay should be exactly T_D. However, due to delays introduced by the network interfaces and the inaccuracies discussed in section 3.1, the actual packet delay will be larger. In the following, we set T_D to 10 ms and compare the actual packet delay to the desired delay T_D. Additionally, we compare the measured error with the error t_e tracked by the emulation itself. This self-determined error t_e corresponds to the time offset between the simulation time and the real-time in the sink when an IP packet is forwarded to the network interface.

Figure 11 shows the ccdf of the measured error and the self-determined error for a packet rate of $r = 200$ packets/s and a packet length of $L = 64$ Bytes. This corresponds to a data rate of 100 kbps at the IP level. In this case, both

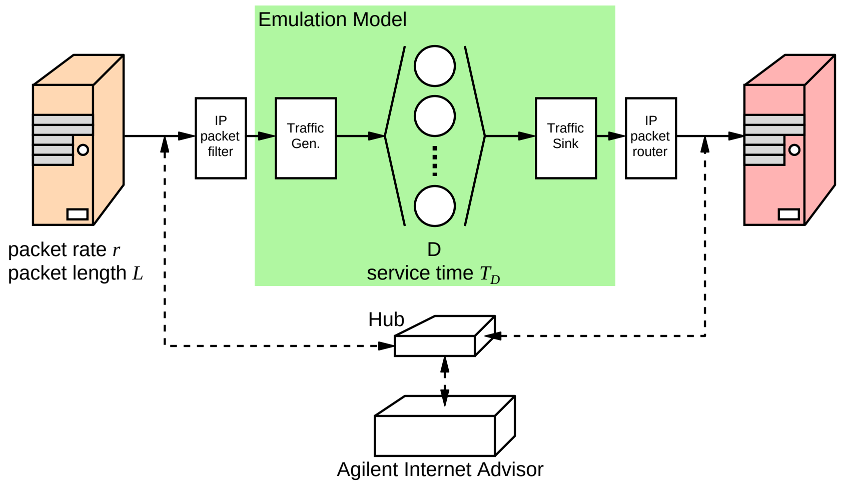

Fig. 10: Measurement setup

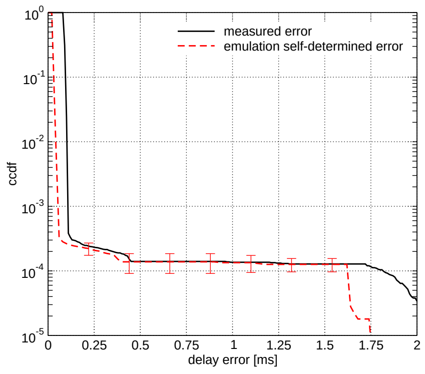

Fig. 11: Delay error for
$r = 200$ packets/s and $L = 64$ Bytes

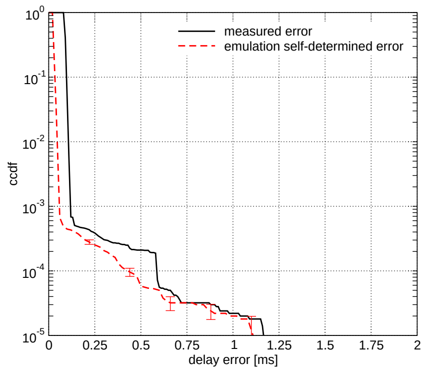

Fig. 12: Delay error for
$r = 2000$ packets/s and $L = 64$ Bytes

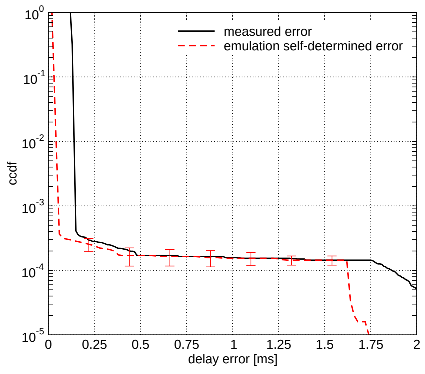

Fig. 13: Delay error for
$r = 200$ packets/s and $L = 530$ Bytes

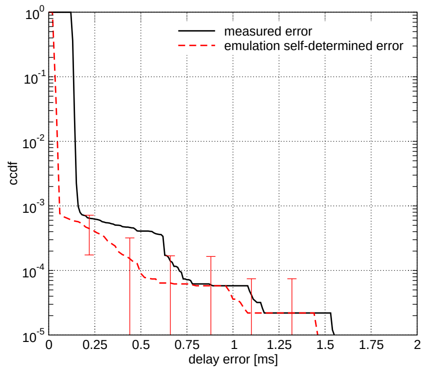

Fig. 14: Delay error for
$r = 2000$ packets/s and $L = 530$ Bytes

the mean measured error and the mean self-determined error stay below 0.1 ms. With 4.31 ms and 1.88 ms, the maximum measured delay error and the maximum self-determined error are significant compared to the target delay. However, we should note from the ccdf that such a large delay applies to very few isolated packets only, as about 10^{-4} of all packets have an error larger than 0.5 ms.

From Fig. 11 we also note the close match between the measured error and the emulation self-determined error. Both error metrics mainly differ by a constant offset due to the systematic error of additional delays in networking interfaces and the operating system, which cannot be seen by the emulation system. This fact is of particular interest, since it allows us to obtain a correct estimate of the timing error at the end of each emulation run without the need of additional external measurement equipment. By doing so, we can easily verify whether the timing errors within a particular emulation run stayed within acceptable ranges.

Figure 12 shows the ccdf of the measured and self-determined error for a ten times higher offered packet rate, i.e., $r = 2000$ packets/s. This corresponds to a data rate of 1 Mbps. When comparing the ccdf to the case of $r = 200$ packets/s, we can observe a significant decrease of the error distribution's tail, while the mean value remains almost unchanged. The majority of the packets still experience small errors, and the maximum error is within acceptable ranges. It is interesting to note that the maximum packet error actually decreases compared to the case of 200 packets/s, which is an effect of the Linux process scheduler. When data packets arrive at a small rate, it is more likely that the operating system switches from the emulation process to another process, and an arriving packet finds the processor busy with some other process. Higher packet arrival rates keep the emulation process busy for longer time periods, making process switches less frequent, thus increasing the time accuracy of the emulation. We should however note that a high packet rate may also result in a high system load, leading to an excessive emulation error and eventually an instable behavior.

In order to further increase the system load, we leave the packet rate r unchanged at 200 and 2000 packets/s, and increase the packet length from 64 Bytes to 530 Bytes. This corresponds to a data rate of 848 kbps and 8.48 Mbps at the packet level, respectively. The error statistics for these cases are shown in Fig. 13 and 14. Compared to the previous case of $L = 64$ Bytes, the ccdfs of the self-determined delay errors are virtually the same, while the ccdf of the measured error is slightly shifted towards larger errors. This shift results from the longer time it takes in the egress module to transmit the additional 466 Bytes of the larger packets (approximately 0.04 ms on a 100 Mbps Ethernet link).

In order to further study the influence of the packet arrival rate, Fig. 15 plots the mean self-determined error in dependence of the packet rate, and Figure 16 the corresponding maximum self-determined error. Both results were obtained by averaging 15 independent measurements with 500.000 transmitted packets each.

The graph shows systematic variations in the timing error depending on the packet rate. While the mean error varies only slightly, the maximum error shows significant variations. This goes well along with our previous observations, indicating that the packet rate does not affect the delay error of the bulk of all

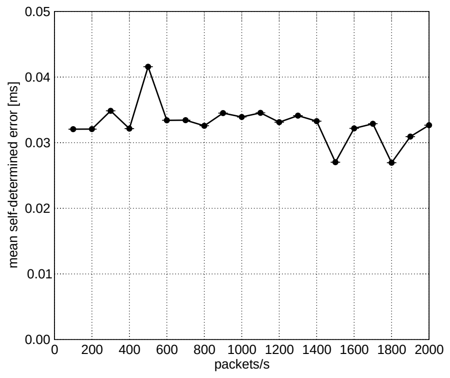

Fig. 15: Mean self-determined error depending on the packet rate for $L = 64$ Bytes

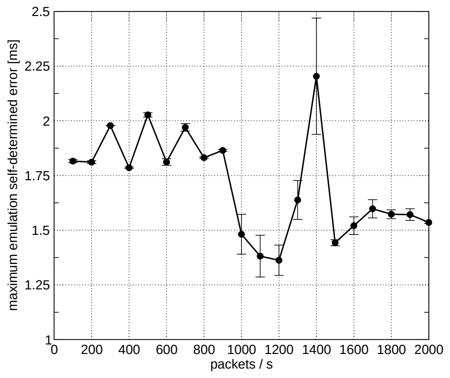

Fig. 16: Maximum self-determined error depending on the packet rate for $L = 64$ Bytes

packets, but rather the tail of the delay error distribution.

It is interesting to observe that the maximum delay error drops as the packet rate increases beyond 1000 packets/s. This correlates with the Linux kernel's scheduling frequency of 1000Hz and supports the observations and explanations of Fig. 11 through 14. The increase in the maximum delay for a packet rate of 1400 packets/s is likely to be explained by other effects and superpositions with the operating system's process scheduler.

To conclude this section, we note that the timing error does not depend on the link load, but rather on the packet arrival rate. A high rate of minimum sized packets is more likely to overload the emulation than a small rate of large packets, even if the actual traffic load on the link remains the same.

This imposes restrictions on the system we can emulate with a certain relative timing error. While the emulation of a system with a system delay of 10 ms may impose high relative errors on the packet delay, it is obvious that when emulating a mobile communication system with one-way delays of 50 ms and more, the mean relative timing error becomes excellent, while the maximum relative timing error is still acceptable. We will detail this issue in the following section, where we investigate the performance of the emulation library in combination with a complex model of the UMTS Radio Access Network. In the simple infinite server scenario, we can very well assume that the timing errors were introduced by external factors, such as the operating system's scheduler rather than by the model itself (cmp. section 3.1). Hence, it is likely that the maximum error found here can be reduced by the use of a real-time operating system.

6 Application Example: An integrated UMTS simulation and emulation testbed

In [15], the IKREmuLib environment was used in combination with a detailed UMTS radio access network model in order to evaluate the performance of web applications in a UMTS environment. While [15] considered regular UMTS Dedicated Channels (DCH) for data transmission, we now extend the model to the more complex High Speed Downlink Packet Access (HSDPA) in order to assess the accuracy of the emulation with a realistic state-of-the-art network model. HSDPA has been introduced within the evolution of 3GPP and provides a fast packet-switched data channel of up to 14MBit/s. The details of the HSDPA model are out of the scope of this paper, but can be found in [16]. Here, we will restrict ourself to the presentation of the considered scenario.

Fig. 17 shows the block diagram of our scenario. We consider a single-cell environment, where several User Equipments (UE) connect to the Node B via a High Speed Downlink Shared Channel (HS-DSCH) in the downlink and a dedicated channel (DCH) in the uplink direction. The Node B is connected to the RNC, which itself is connected to the Internet via the 3G-SGSN and 3G-GGSN of the cellular system's core network. The UEs establish a data connection with a host in the Internet. The Internet and core network were assumed to introduce a constant delay $T_{\mathrm{INet}} = 20\,\mathrm{ms}$ in each direction and not lose any IP packets.

We consider one UE where the server transmits packets of length $L = 64\,\text{Bytes}$ in the downlink at a constant rate of 100, 200 or 400 packets/s. This corresponds to data rates at the IP level of 51.2 kbps, 102.4 kbps or 204.8 kbps, respectively. The traffic for this UE is generated by the server in the testbed and then fed into the emulation system, that is we emulate the transmission of real traffic over HSDPA with this mobile.

We further consider one to five additional UEs, which generate cross-traffic on the same packet-switched HS-DSCH. For this cross-traffic, it is not so much important that it is generated by a real traffic source. Instead, it is well sufficient to generate the cross-traffic within the emulation model itself by the same mechanisms as traffic would be generated for a simulation run. We therefore call this traffic *simulated cross-traffic*, in contrast to the fully emulated traffic described above. In particular, we will consider bulk data transfer via TCP in the downlink direction for each cross-traffic UE. This exhibit one additional advantage of the close interrelation of simulation and emulation.

Fig. 18 plots the absolute error of the emulated traffic in combination with one cross-traffic user for the three considered packet arrival rates. In contrast to the simple infinite server scenario, an increased packet rate significantly increases the absolute delay error. The reason is the much more complex processing which has to be done for each packet within the model of the radio access network. That means that the major contribution to the timing error results from model imposed inaccuracies rather than operating system imposed inaccuracies.

Fig. 19 plots the same metric for five cross-traffic users. Compared to the case of one cross-traffic user, the accuracy gets worse, leading to very large error delay distribution tails for a packet rate of 400 packets/s. The reason are again model imposed inaccuracies, as the emulation model is under higher load when it has to deal with several data streams.

In order to assess the practical impact of the delay error, we have to set the absolute delay error in relation to the absolute delay of the packet within the emulation and study the relative delay error. Fig. 20 and Fig. 21 plot this relative delay error for the same scenarios as before. The general characteristics of the curves are identical to those of the absolute delay error. We can also observe that in all considered cases the relative error for 95% of all packets is below 4%. Depending on the load situation, the relative error for single packets grows up to 10-15%, but it is important to note that it stays within acceptable ranges for the majority of all packets and scenarios.

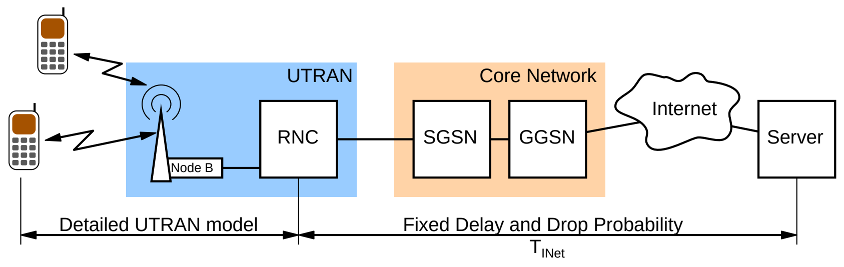

Fig. 17: Architecture of the considered UMTS system

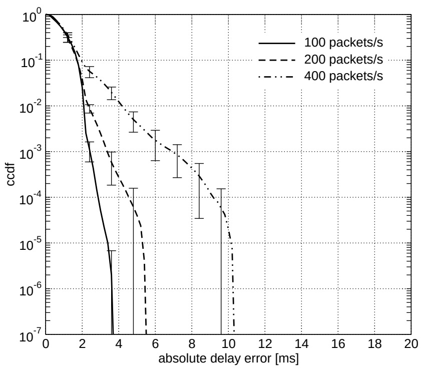

Fig. 18: Absolute error, $L = 64\,$Bytes, one simulated cross traffic user

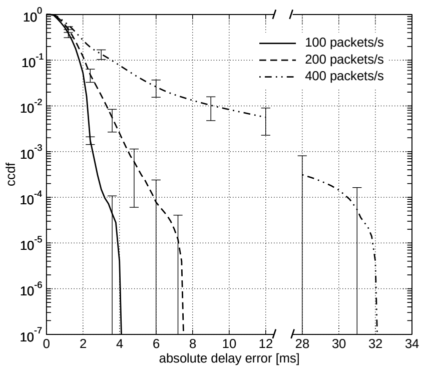

Fig. 19: Absolute error, $L = 64\,$Bytes, five simulated cross traffic users

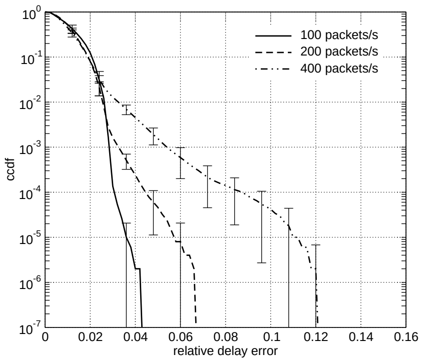

Fig. 20: Relative error, $L = 64\,$Bytes, one simulated cross traffic user

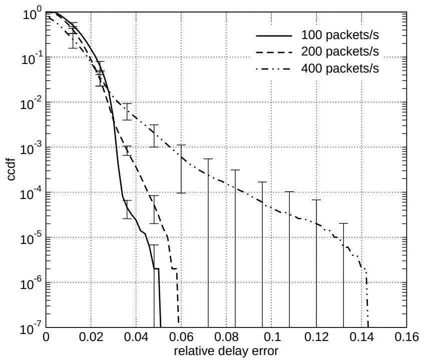

Fig. 21: Relative error, $L = 64\,$Bytes, five simulated cross traffic users

We conclude that in a realistic scenario the model imposed inaccuracies dominate the other inaccuracies introduced by the operating system and asynchronous packet arrivals. Consequently, when setting up an emulation testbed, one has to carefully keep in mind the complexity of the model and study the self-measured error in order to ensure correctness. Alike, the packet generation rate of the involved traffic sources is an important issue. When investigating real data traffic, such as HTTP or FTP traffic, it comes in handy that those traffic sources usually generate large data packets, keeping the packet rate at a relatively low value, giving a good chance to accurate emulation results.

7 Conclusion

We presented the new emulation library IKREmuLib, which settles on top of the well proven IKRSimLib. It allows for an easy and seamless combination of both the simulation and emulation method. Existing simulation models can easily be used for emulation purposes provided that they meet the model-imposed real-time requirements. In addition, we can add simulated cross traffic to the actually measured real data streams. We showed by means of measurement that the timing accuracy of the IKREmuLib for simple models is less than 0.5 ms for 99% of all packets and less than $2-4$ ms for all remaining packets in all considered scenarios. We also demonstrated that in combination with a complex radio access network model we can achieve relative timing errors of only a few percent for realistic scenarios. Together with the possibility to combine simulated and emulated components, this makes it an ideal complement to the performance evaluation of many network applications, in particular of the evaluation of radio access networks.

References

1. A. M. Law and W. D. Kelton, *Simulation Modeling & Analysis*, 2nd ed. McGraw-Hill, 1991.
2. S. Guruprasad, R. Ricci, and J. Lepreau, “Integrated network experimentation using simulation and emulation,” in *Proc. Tridentcom*, Feb. 2005, pp. 204–212.
3. L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu, “Advances in network simulation,” *IEEE Computer*, vol. 33, no. 5, pp. 59–67, May 2000.
4. K. Fall, “Network emulation in the VINT/NS simulator,” in *Proc. IEEE International Symposium on Computers and Communications*, Red Sea, Egypt, July 1999, pp. 244–250.
5. K. Pawlikowski, H.-D. Jeong, and J.-S. Lee, “On credibility of simulation studies of telecommunication networks,” *IEEE Communications Magazine*, vol. 40, no. 1, pp. 132–139, January 2002.
6. D. Mahrenholz and S. Ivanov, “Real-time network emulation with ns-2,” in *Proc. IEEE International Symposium on Distributed Simulation and Real-Time Applications*, October 2004, pp. 29–36.
7. H. Kocher and M. Lang, “An object-oriented library for simulation of complex hierarchical systems,” in *Proc. Object-Oriented Simulation Conference (OOS ’94)*, Tempe, AZ, 1994, pp. 145–152.
8. *IKR Simulation Library.* [Online]. Available: http://www.ikr.uni-stuttgart.de/Content/IKRSimLib/
9. *RTLinux.* [Online]. Available: http://www.fsmlabs.com/
10. “IEEE guide to the POSIX open system environment (OSE),” *IEEE Std 1003.0-1995*, 1995.
11. B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming.* O’ Reilly and Associates, 1998.
12. S. McCanne and V. Jacobson, “The BSD packet filter: A new architecture for user-level packet capture,” in *Proc. USENIX Winter*, January 1993, pp. 259–269.
13. W. R. Stevens, *UNIX Network Programming*, 2nd ed. Prentice Hall, 1998, vol. 1.
14. *pcap packet capture library*, 2005. [Online]. Available: http://www.tcpdump.org/
15. M. C. Necker, M. Scharf, and A. Weber, “Performance of TCP and HTTP proxies in UMTS networks,” in *Proc. European Wireless*, vol. 2, Nicosia, Cyprus, April 2005, pp. 504–510.
16. M. C. Necker and A. Weber, “Impact of Iub flow control on HSDPA system performance,” in *Proc. 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC 2005)*, Berlin, Germany, Sept. 2005.