Distributed Emulation of Shared Media Networks

Daniel Herrscher, Steffen Maier, Kurt Rothermel
University of Stuttgart, Institute of Parallel and Distributed Systems (IPVS)
Universitätsstr. 38, D-70569 Stuttgart, Germany
herrscher@informatik.uni-stuttgart.de

Keywords: Network emulation, distributed emulation, performance analysis, shared media, CSMA/CD.

Abstract

Comparative performance measurements of distributed applications and network protocols require the availability of appropriate network environments. Network emulation approaches offer a flexible way to mimic the properties of a variety of networks. While there are several centralized and distributed emulation tools available that can mimic the behavior of point-to-point links, shared media communication can only be emulated by totally centralized tools so far. To permit the emulation of large-scale mixed network scenarios, the emulation of shared media networks has to be performed in a distributed way also.

In this paper, we introduce several strategies to emulate shared media networks in a distributed fashion. For one promising approach, the distributed emulation of carrier sensing, we present a prototype implementation. Measurements show the applicability and limitations of our solution.

I. INTRODUCTION

During the design and implementation of distributed applications and protocols, it is essential to analyze the impact of various network environments on their performance. While mathematical analysis and simulations are commonly used in early design stages, measurements have to endorse the theoretical results as soon as implementations become available. These measurements should be conducted in various network environments to facilitate comparative performance estimations. Since it is very unlikely to have all these environments available in real hardware, there is a strong need for synthetic network environments that can be parametrized in order to reproduce an original or fictitious network. The process of introducing network properties that differ from the actual properties of the hardware in use is called *network emulation.* A *network emulation tool* is software capable of altering network traffic in a specified way. A facility consisting of a combination of flexible networking hardware and suitable emulation tools is called *network emulation testbed.*

There are two general approaches to network emulation: centralized and distributed. In centralized emulation solutions, all network traffic in a scenario is sent through a central emulation tool, which can constitute a bottleneck. Therefore, for larger scenarios, the emulation efforts have to be distributed. Various tools for distributed network emulation exist. However, these tools focus on the emulation of point-to-point links. Distributed emulation of networks with shared media has not been addressed yet. While in wired networks, shared media access protocols are becoming less important, the interest in shared media wireless networking is growing. To facilitate the distributed emulation of complete scenarios consisting of both point-to-point and shared media networks, we propose several strategies for shared media network emulation to complement existing emulation tools. We choose one promising approach for a prototype implementation and provide measurements that show the applicability of our solution.

The remainder of this paper is organized as follows: Section II gives a systematic overview of the related work in network emulation. In Section III, we discuss several possible approaches to distributed emulation of shared media networks. Section IV explains in detail the architecture of our favorite approach, namely the distributed emulation of carrier sensing, and provides measurements. Section V concludes the paper.

II. RELATED WORK

All emulation approaches have in common that they connect communicating software modules running on real machines to certain, different positions within an emulated network scenario. Since this scenario often consists of several network links, there are two basic architectures to build the emulated network: Either the whole scenario is emulated by one single, central emulation entity, or several instances of an emulator are connected together to form a comprehensive scenario, each of them responsible for emulating its own part of the network (usually, a single link).

A. Centralized Emulation

Since the simulation of complete network scenarios has been addressed in great detail already [1], it stands to reason that there are several efforts to reuse existing simulators for emulation purposes. Simulators can work with complex network models, including both exclusive links and shared media.

There are two basic problems to be solved to facilitate the usage of an existing simulator core for emulation purposes. First, common network simulators work with discrete event schedulers. Events are processed as soon as the effects of all prior events are evaluated. To interact properly with a real scenario, however, the scheduler has to be modified to be synchronized with real-time. Second, an interface for packet capture and generation has to be provided. These issues have already been addressed for the most common network simulators [2, 3]. The real-time requirement appears to become a problem when moving to realistic scenario sizes and bandwidths. To some extent, this can be addressed by parallelizing the discrete event simulator [4]. Other similar approaches explicitly state that they aim at the emulation of low bandwidth links in small scenarios only [5, 6].

B. Distributed Emulation

Since centralized emulation approaches are very limited in terms of scenario size, it is common to distribute the emulation efforts among several instances. A coherent idea is to partition the emulation scenario according to the emulated topology, leading to one emulation tool per emulated link in the extreme case.

Most emulation tools change network properties by intercepting, delaying or altering data in the protocol stack. Early emulation attempts aim at the emulation of a single network link only [7, 8, 9]. They differ in the parameters they can affect, but have in common that the emulation parameters stay constant during the experiments.

Recent approaches include dynamic parameter changes [10], triggered by the replay of previously gathered measurement data [11], or more generic models [12].

It is obvious that tools aiming at the emulation of a single point-to-point link cannot be used for the realistic emulation of shared media networks. Just connecting several instances of separate link emulators is not sufficient, because the emulating tools would need to cooperate in order to maintain a common media model. Only with a common media model, effects like bandwidth sharing and the dependence of throughput on overall load can be emulated. To our knowledge, there are no existing network emulation tools that provide the distributed emulation of a shared medium.

C. Testbeds

Two connected machines equipped with network emulation tools can help analyzing some aspects of e.g. a transport protocol. However, to perform measurements within more complex network topologies, e.g. to analyze routing protocols, more machines are necessary. With the growing number of participating nodes, both the setup of the machines and the coordination of the emulation tools becomes a problem, especially if the emulated scenario includes dynamic changes during the experiment.

Emulation testbeds face these problems, as they ease the setup and operation of emulation scenarios consisting of large numbers of nodes and connections. “Netbed” [13] at the University of Utah consists of 168 PC nodes connected by a number of switches, working together as large “programmable patchpanel” to create almost any virtual connection topology. Special link properties can be introduced between the nodes. Main focus of the netbed project is the efficiency and ease of the setup process to facilitate the parallel, intensive use of the resource by many different research groups at the same time.

The Network Emulation Testbed (NET) [10] at the University of Stuttgart consists of 64 PC nodes connected by both a monolithic, programmable gigabit switch, and a separate administration network for setup and control (see Fig. 1). Using VLAN technology, the gigabit switch can create an arbitrary connection topology between the nodes. Custom link emulation software running on all nodes introduces the respective link parameters. Currently, the main focus of the NET project is the realistic emulation of high bandwidths, and real-time link property changes.

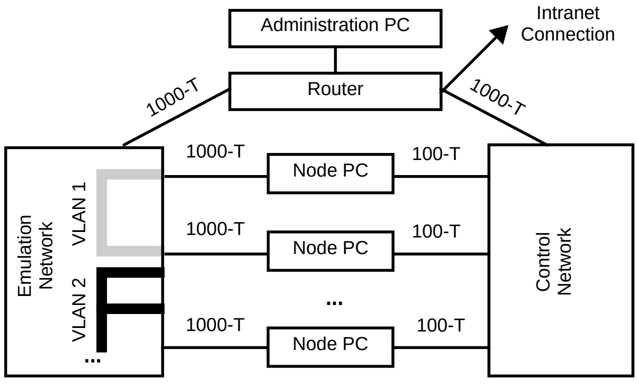

Figure 1. Hardware Architecture of the Network Emulation Testbed at the University of Stuttgart.

The operation of any network testbed is based on suitable emulation tools. Any improvement in the scope of these tools will directly broaden the application area of network testbeds.

III. POSSIBLE ARCHITECTURES

In this chapter, we will investigate the possible architectures for an emulation tool providing a shared media model.

A. Centralized Media Model

The straightforward way to manage a consistent media model with several participants is to hold the model in one central instance. One possible solution would be to send all transmission requests to an emulation tool instance dedicated to the respective media. Since the central instance has the global view of the medium, it can easily determine which transmission requests can be served at which time, and whether some transmissions get lost. The central emulation tool could handle the actual transmission to the target machine on its own (see Fig. 2). However, because of two reasons, the originator would need some feedback from the media model. First, some kind of flow control is needed to facilitate back pressure to the upper layers of the sender. Second, transmission errors that can be detected by the sender have to be reported to the sending process, e.g. if excessive collisions occur in the emulated model.

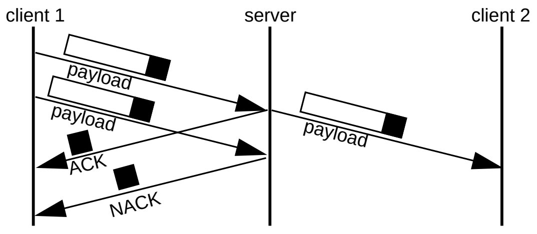

Figure 2. Centralized media model.

For an exact frame-level emulation with this architecture, it would be necessary to have perfectly synchronized clocks on all clients. A sender that marks each outgoing frame with the exact system time would enable the central emulation instance to determine whether two frames from different clients have been sent within the short time frame that would lead to a collision. Clock synchronization with microsecond accuracy would be needed to detect these effects. Clearly, this would be an unrealistic requirement for common testbed hardware.

Existing testbeds use an emulation network featuring high bandwidth and low latency [10, 13]. Therefore, it seems acceptable to neglect the actual transmission delays within the emulation network, and mark the incoming frames at the central instance with a timestamp using a reasonable accuracy. Using this simplification, the approach appears still promising to provide realistic results.

Apart from the inevitable additional transmission and processing delay that is introduced by the central instance, an intrinsic limitation is constituted by the available bandwidth and the processing power of this instance. However, in contrast to the totally centralized emulation approaches mentioned in Section II, this bottleneck would appear per shared media, not for the whole emulation scenario.

B. Fine-Grained Centralized Media Model / Distributed Emulation

The problem of having a bandwidth bottleneck at a central instance can be faced by an approach using a central media model, but performing the actual payload handling in a distributed fashion (cp. Fig. 3). Still, the central emulation instance needs to be contacted for each medium access request. However, the actual delivery of the frames is left to the respective sending instance.

With a combination of centralized model and distributed emulation, the abovementioned problem of additional transmission and processing delay remains. Worse yet, because the central instance would need an additional message to signal the media status back to a station that issued a media request, the additional transmission delay would be doubled. The benefit compared to the first approach would be the avoidance of an upper bandwidth limit imposed by the central instance.

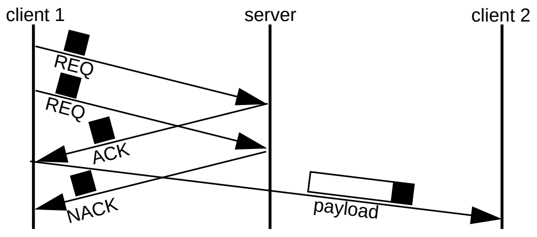

Figure 3. Centralized media model, distributed payload handling.

C. Coarse-Grained Centralized Media Model / Distributed Emulation

The only way to take the advantages of a central media model without introducing an excessive communication overhead is to abandon the concept of a media model with frame level accuracy. A more coarse-grained media model can also emulate most of the characteristic effects of shared media networks, like the correlation between load and throughput. For some experiments (like throughput analysis), a realistic emulation on frame level is not necessary to evaluate the typical performance in a specific environment, but a realistic "mean" network behavior will do.

Instead of keeping track of all media access requests, each station is assigned a fraction of the available bandwidth, along with the current values for typical medium access delay and transmission failure probability. The local emulation tool instances on each station are responsible to shape the outgoing traffic according to these parameters. At all times, they also keep track of the actual bandwidth usage, and whether the station wants to send more than its currently assigned share. On a regular basis Δt (e.g., 100 ms), these data are sent to the central instance. According to the gathered data and the respective media type to be emulated, the central instance can compute a new fragmentation of the available bandwidth, and communicate the updated emulation parameters to the respective local tools.

With this approach, the message overhead does not depend on load and bandwidth of the emulated network, but only on the parameter Δt. This parameter can be selected according to the required accuracy and the capability of the testbed.

While the average performance of a shared media can be emulated with this approach, one must keep in mind that, if observed at a fine time scale, the behavior of the emulated network will differ significantly from the original. For example, for a typical small LAN, it is a quite common situation that the medium is idle for most of the time, with excessive bursts if one of the participants issues a transfer. The real behavior the sender on a medium with CSMA/CD access would experience would be the instant allocation of the full media bandwidth. With a coarse-grained media model like discussed above, each potential sender will be assigned an equal share of the total bandwidth initially while the medium is idle. It will take up to Δt until the full bandwidth can be assigned to an exclusive sender.

The collection of media access requests at the potential senders appears to become another practical problem with this approach. It would be ideal to know the media usage of a sender for the next Δt period in order to get the appropriate bandwidth share assigned. Of course, for an emulation tool working at lower levels in the protocol stack, it is not straightforward to predict the bandwidth requirements of a sender. A possible solution to this problem has been proposed in [14].

With this approach, it is guaranteed that the overall bandwidth limitation of the emulated media will not be exceeded at any time. On large time scales, the emulated bandwidth assignment will be realistic. On frame level, however, the traffic patterns will differ significantly from the patterns observed in reality.

D. Distributed Media Model

To overcome the limitations of a central emulation instance, we will now propose an approach to hold the media model at each station separately, and to emulate the media access algorithms used in real shared media networks. Since the most common approach in today's shared media access technology is CSMA, it stands to reason to propose an approach that emulates CSMA behavior. While it would of course be possible to emulate the behavior of distributed mutual exclusion algorithms in a distributed fashion, we refrained from including them because the technologies using these algorithms (e.g. Token Ring and DQDB) are of decreasing importance in practice.

In the following, we describe an approach for CSMA/CD emulation. However, we want to point out that the concept would also be applicable to other CSMA versions.

The basic idea is to hold an up-to-date model of the emulated media at every participating station, and to keep the respective media models consistent. This can only be achieved if the emulation tools on all stations listen to all transmissions on the media, whether they are destined to the respective station or not. In addition to that, again we have to assume that the transmission delay in the emulation network can be neglected.

Given the above assumptions, it is possible for each local emulation tool to maintain its own model of the emulated media. On the reception of a frame, the emulation tool can calculate the time this frame would occupy the emulated media (according to the bandwidth, which is given as parameter), and update its local model accordingly. Frames that are destined to the local machine are forwarded to upper layers, all other frames can be discarded by the emulation tool. Frame transmission requests by the local station are handled according to the CSMA/CD algorithms, taking the emulated media status into account. If an emulation tool receives an additional transmission while its emulated media is in the "busy" status, it can react accordingly and emulate the effects of a collision.

E. Summary

Entirely centralized emulation tools can cover the emulation of shared media networks, but come with the well-known performance problems.

The combination of a centralized model and distributed payload management can solve these problems to some extent, but introduces additional artifacts for typical traffic patterns.

The distributed emulation of CSMA/CD is most promising to us, because it can emulate the typical characteristics of the protocol family that is most relevant today.

IV. DISTRIBUTED CSMA/CD EMULATION

To show that shared media emulation with a distributed media model is a feasible approach, we implemented a prototype tool emulating the behavior of CSMA/CD.

In the remainder of this section, we will show in detail the functionality and implementation of our tool "ethemu," provide some measurements, and point out some practical limitations.

A. Architecture

Ethemu is a software module that can emulate the behavior of a CSMA/CD network with certain parameters, while running on PCs that are actually connected to a high-speed networking infrastructure.

Similar to the related approaches Dummynet [9] and Netshaper [10], we inserted an additional "emulation layer" into the protocol stack of the operating system. The additional layer can be logically viewed as an additional, virtual medium access sublayer inside the actual link layer. Because it is completely transparent to upper layers, it facilitates the performance testing of protocols on layer 3 and above without having to modify the subject on investigation.

We implemented the additional emulation layer by adding a module to the Linux operating system kernel (version 2.4.18). Using a kernel module instead of directly manipulating the networking code in the kernel makes it possible to dynamically load and unload the emulation tool, without having to reboot the system. Furthermore, users of the emulation tool do not have to compile and install a special kernel. A standard kernel (e.g. like shipped with Red Hat Linux) will do.

B. Basic Functionality

Since the existing emulation testbeds use high-speed switched ethernet as underlying physical network, first of all we must ensure that every station that is logically attached to an emulated shared medium gets all frames that are sent. For that reason, ethemu encapsulates every frame into a broadcast frame, including the original frame header, before it is passed down to the actual device. On the receiving side of each station that gets the broadcast, ethemu strips off the encapsulating broadcast header again.

According to the specified emulation parameters (first of all: the bandwidth) and the current traffic observations, ethemu maintains a private model of the emulated shared media at each station. As long as all stations get the transmissions at the same time, the media model is consistent among the stations. Using the current media model, ethemu can emulate the behavior of the CSMA/CD media access scheme, resulting in a delayed frame delivery, and occasionally frame loss.

C. Sending Frames

On the sending side, the module appears as a virtual Ethernet device driver, and is treated like a real device by the network layer. The emulation device is named “ethemu0” in Fig. 4. It receives out-bound frames from the network layers, according to routing table entries associated with its device ID. If a frame is not being dropped due to emulated excessive collisions, it is encapsulated in a broadcast frame and gets sent out as soon as the emulated media access scheme allows. An attached real network device like e.g. “eth0” in Fig. 4 is used for the physical communication.

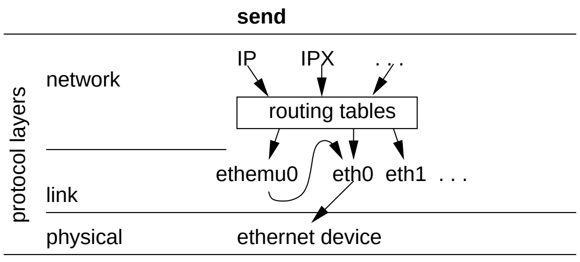

Figure 4. Hooking into the send data path.

D. Receiving Frames

On the receiving side, the module intercepts incoming emulation broadcast frames. It does so by registering as a virtual network layer implementation using an otherwise unused Ethernet protocol type (cp. Fig. 5). All emulation broadcast frames are marked with this special protocol type. This approach assures that the module is able to exclusively handle all incoming emulation frames, and thus hook into the reception data path without further interference with other network protocol implementations. Note that, although logically ethemu works between layer 2 and 3, it is implemented at layer 2 on sending side, and at layer 3 on receiving side.

If a frame was not already dropped because of an emulated collision, it finally gets delivered as soon as the emulated media access scheme allows. Delivery is realized by again acting like a network device driver and reinserting the frame as if it was just received from a device.

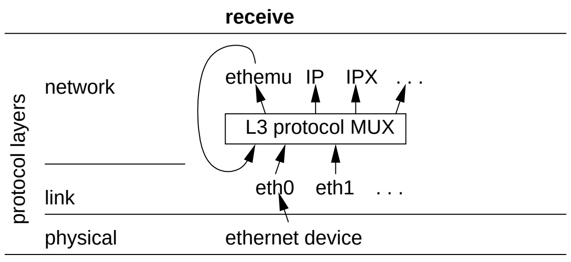

Figure 5. Hooking into the receive data path.

E. Media Model

Core of the emulation tool is a model of the emulated shared media. It was developed with the objective to emulate the media access scheme of IEEE 802.3 as precisely as possible. The respective media access algorithm that is usually carried out by network device hardware is re-implemented within our software module.

Fig. 6 shows both the control and data flow, as well as important side effects in the state machine emulating the media access algorithm. The status is held in a set of local variables: Two timestamp variables keep track of the carrier on the emulated shared media. The beginning of the carrier is denoted by *ts*, whereas *tsr* denotes the time when the carrier disappears. The local system time is denoted by *t*. A boolean flag named *sending* indicates whether a station is currently transmitting a frame. A frame that is currently being sent or received by a station is called an *active* frame. A pointer to its respective data buffer is held in a local variable. The term *t_s* in the figure denotes the propagation delay between the sender of a certain frame and the receiver.

The data flow basically consists of immediate enqueueing, and delayed dequeuing of frames both on the sending and the receiving side. Note that all frames keep their FIFO ordering.

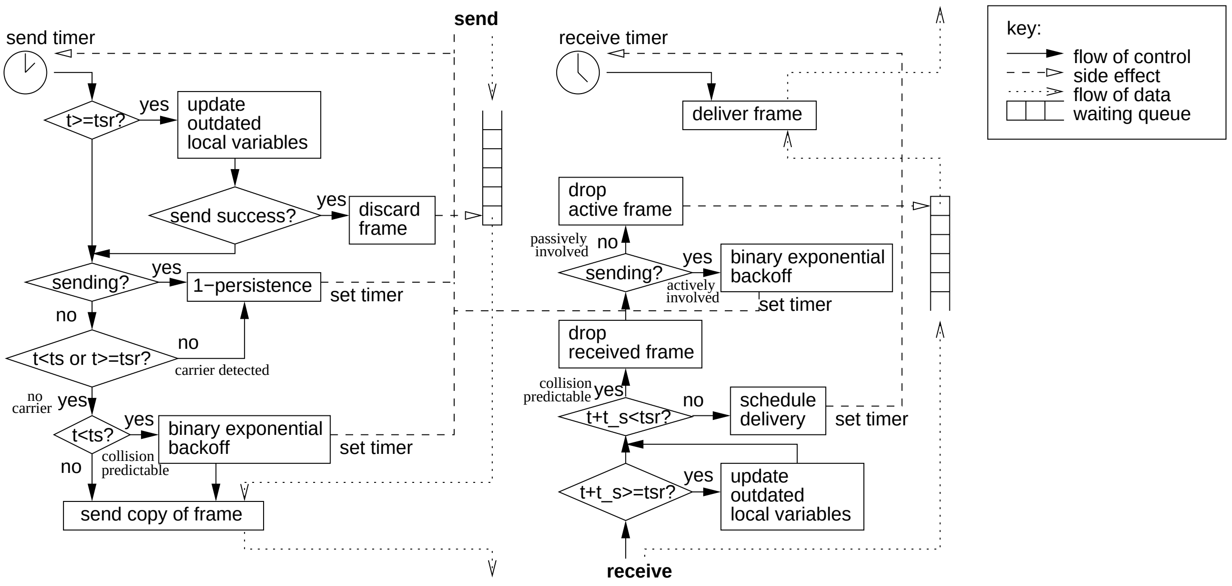

Figure 6. Schematic functionality of ethemu.

Time-based triggering is controlled by a send and a receive timer, respectively.

When a send request occurs, the station enqueues the frame into the send waiting queue and sets the send timer, if it was not already set for another, earlier frame. The trigger time is either immediately or *tsr*, if a carrier is sensed (i.e., $ts \leq t < tsr$ holds true). On send timer activation, local variables are updated at first. If the station is already sending, it adjusts the send timer to *tsr* to serialize transmissions according to 1-persistence. The same procedure applies to a sensed carrier belonging to a frame in transmission by another station. If a collision is already predictable because a frame is on its way to the station, but the transmission delay has not yet passed, the outbound frame is sent out to inform all other stations of the upcoming collision. Additionally, the send timer is set for a retransmission according to the binary exponential backoff algorithm. Finally, if there is no carrier sensed and no collision predictable, a copy of the frame is sent through the actual network device. Determination of send success is not time critical and thus left to the next update of local variables.

On reception of a frame, again the local variables are updated at first. If the carrier of the received frame does not overlap with an already present carrier, delivery is scheduled by enqueueing the frame into the receive waiting queue, and adjusting the receive timer to the time when both transmission and serialization delay have passed. Otherwise, an emulated collision will occur, and the received frame is discarded. If the station was actively involved in the collision because of an outgoing transmission, a retransmission is scheduled through the send timer according to the binary exponential backoff algorithm. If another station is causing the collision, the frame which was in progress of being received is discarded from the delivery waiting queue. If a frame was not involved in a collision until receive timer activation, it finally gets dequeued and delivered.

F. Limitations

The exactness of our emulation results is directly influenced by the timer granularity provided by the operating system. The accuracy of the send timer is most important: Its task is to trigger serialized frame transmissions according to 1-persistence, as well as scheduled retransmissions after collisions. In order to conduct these tasks, very short periods of time need to be permanently measured to determine possible collisions by overlapping carriers. For most realistic emulation results, a time granularity corresponding to the duration of one transmitted bit would be necessary. The bit duration t_B is reciprocal proportional to the transfer rate *b*: $t_B = 1/b$. Given the timer granularity, it is possible to determine the transfer rate that can be emulated perfectly, ensuring e.g. that every collision can be recognized, even if carriers overlap by just one single bit. In the following, we call this transfer rate the “safe rate”.

By default, timer interrupts are handled periodically every 10 milliseconds in the Linux kernel. It is possible to reasonably lower that value by an order of magnitude by recompiling the kernel with different timer settings [9]. However, even with millisecond accuracy, the resulting granularity is far from being precise enough for 10 MBit/s. Nevertheless, it is always possible to run the emulation with transfer rates higher than the “safe rate.” As one consequence, overlapping frames may not be recognized as collisions any more.

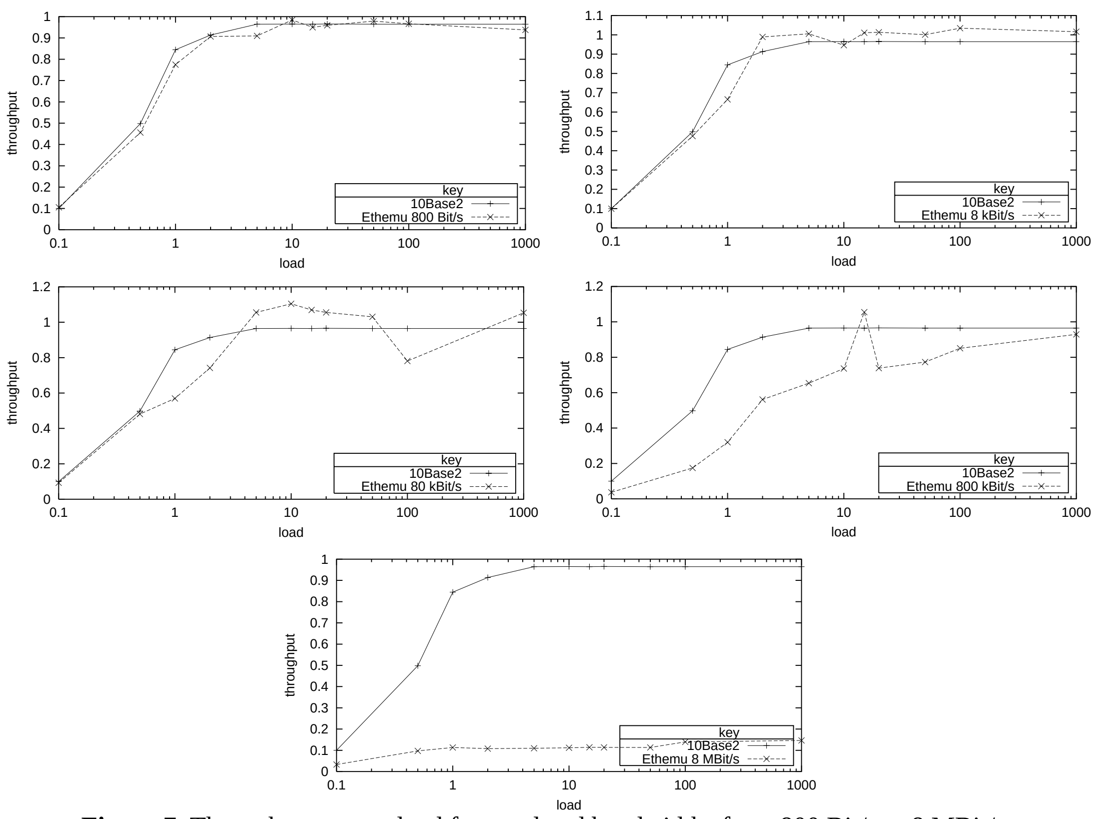

Figure 7. Throughput versus load for emulated bandwidths from 800 Bit/s to 8 MBit/s.

G. Measurements

By means of extensive measurements, we will show to what extent the abovementioned problems are actually relevant for operation, and how precisely we are able to emulate Ethernet.

The performance comparison between real Ethernet and our emulation approach is done on the basis of the characteristic curve which plots network throughput versus network load. The experimentally derived characteristic curve of a real 3-node 10Base2-Ethernet with shared media serves as reference plot. The ethemu measurements were conducted using 3 Pentium III PCs, connected by a Fast-Ethernet switch. We used a load generator injecting varying frame bursts of different sizes on link layer. The resulting characteristic curves for different emulated transfer rates compared to the reference curve are shown in Fig. 7. In the figures, “load” refers to the ratio of overall bandwidth requests to the available bandwidth, i.e. a load of 3 on a 10 MBit/s media would mean three stations try to send 10 MBit/s each. The throughput is normalized to the available bandwidth.

At a timer granularity of 10 milliseconds, the “safe rate” according to the above formula would be only 100 Bit/s. Because of practical limitations, we started our measurements at 800 Bit/s, however. Conducting measurements at lower speed with a reasonable amount of transferred data would simply take too long (several days).

The measurements at 800 Bit/s and 8 kBit/s show a convincing similarity to the reference curve. At 80 kBit/s, the results start to show a deviation; beginning at 800 kBit/s, the throughput stays clearly below the reference. This can be explained as follows: If a station wants to start sending exactly when a busy medium is free again (1-persistence), the respective emulation tool has to rely on an exact timer. If the timer triggers too late, the medium remains unused, although there was data to send, which results in a lower throughput. The higher the bandwidths, the smaller the timeframes, and the higher the relative errors due to coarse timers. There is also the converse effect that with higher timer errors, less collisions can be discovered, which would lead to a higher throughput. On a quantitative basis, however, the first effect prevails by far.

From these measurements, we derive that we are able to reasonably emulate Ethernet with transfer rates of at least 2 orders of magnitude higher than the “safe rate.”

H. Improvements

The gap to emulation of practically relevant transfer rates can be closed with the help of a finer time granularity. This aim is achieved by both using a more granular local clock for timestamps and time comparison, as well as more precise timers.

The first can be easily accomplished by using time stamp counter registers which are common in current processor types. Implementations for more precise timers in the Linux kernel are freely available, e.g. from the University of Karlsruhe [15]. This implementation uses the local advanced programmable interrupt controller in recent Intel-compatible processors, which is triggered by the processor bus clock frequency. Assuming a bus clock frequency of 100 MHz, we would gain 5–6 orders of magnitude over the standard timer frequency of 100–1000 Hz. Transferred to measurement results, this would push the "safe rate" for bit-precise emulation up to 10 MBit/s. Given that our measurements have also shown that emulation works for rates 2 orders of magnitude higher than the "safe rate," the emulation of existing practically relevant CSMA/CD technologies is assured. We are currently in progress of adapting our prototype to use fine-grained timers, and we do not expect fundamental problems with that. However, additional measurements have to ensure that the compute power of the emulation nodes is still sufficient to run the emulation tool at high transfer rates.

The current implementation aims at the reproduction of CSMA/CD, and is thus able to emulate those parts of the IEEE 802.3 family dealing with shared media. In this regard, it is certainly interesting to look into the distributed emulation of similar media access schemes like CSMA/CA, especially because the emulation of wireless LANs is becoming more and more relevant to the community. Since the basic idea of having a private model of the shared media at every participating station can be also used for CSMA/CA emulation, we are confident that it is possible to adapt our existing tool with reasonable effort.

V. CONCLUSION

Network emulation is essential for comparative performance measurements of distributed applications and protocols. For larger scenarios and higher bandwidths, emulation efforts have to be distributed. Existing approaches to distributed network emulation focus on the emulation of single network links. The distributed emulation of shared media networks has not been addressed before.

In this paper, we identified several possible approaches to realize the emulation of shared media networks in a distributed fashion. For a promising approach, the distributed emulation of CSMA/CD, we provided a prototype implementation.

Currently, our prototype tool has limitations concerning the maximum emulated transfer rate, but shows qualitatively realistic results. We identified the operating system timer granularity as reason for this limitation, and proposed possible workarounds.

The resulting emulation tool "ethemu" clearly broadens the scope of emulation testbeds, and is already in use in the "Network Emulation Testbed" at the University of Stuttgart. With the growing possibilities of emulation tools and testbeds, we believe that emulation is likely to gain more importance for performance analysis as complement to simulation and live testing.

REFERENCES

[1] Breslau, L., D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. 2000. "Advances in Network Simulation." *IEEE Computer,* 33(5):59–67.

[2] Fall, K. 1999. "Network Emulation in the Vint/NS Simulator." In *Proceedings of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems,* (Red Sea, Egypt), 244–250.

[3] Ke, Q., D.A. Maltz, and D.B. Johnson. 2000. "Emulation of Multi-Hop Wireless Ad Hoc Networks." In *Proceedings of the 7th International Workshop on Mobile Multimedia Communications (MoMuC 2000),* (Tokyo, Japan).

[4] Simmonds, R. and B. Unger. 2001. "Towards Scalable Network Emulation." In *Proceedings of SPIE Vol. 4526 (2001): Scalability and Traffic Control in IP Networks,* (Denver, August), 252–262.

[5] Davies, N., G.S. Blair, K. Cheverst, and A. Friday. 1995. "A Network Emulator to Support the Development of Adaptive Applications." In *Proceedings of the 2nd USENIX Symposium on Mobile and Location Independent Computing,* (Ann Arbor, April), 47–55.

[6] Kojo, M., A. Gurtov, J. Manner, P. Sarolahti, T. Alanko, and K. Raatikainen. 2001. "Seawind: A Wireless Network Emulator." In *Proceedings of 11th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems,* (Aachen, Germany, September).

[7] Carson, M. "NISTNet Network Emulator." available from *http://www-x.antd.nist.gov/nistnet/.*

[8] Ingham, D.B. and G.D. Parrington. 1994. "Delayline: A Wide-Area Network Emulation Tool." *Computing Systems,* USENIX, 7(3):313–332.

[9] Rizzo, L. 1997. "Dummynet: A simple approach to the evaluation of network protocols." *ACM Computer Communication Review,* 27(1):31–41

[10] Herrscher, D. and K. Rothermel. 2002. "A Dynamic Network Scenario Emulation Tool." In *Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN '02),* (Miami), 262–267

[11] Noble, B.D., M. Satyanarayanan, G.T. Nguyen, and R.H. Katz. 1997. "Trace-Based Mobile Network Emulation." In *Proceedings of the ACM SIGCOMM '97,* (Cannes, France, September), 51–61.

[12] Herrscher, D., A. Leonhardi, and K. Rothermel. 2002. "Modeling Computer Networks for Emulation." In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02),* (Las Vegas), 1725–1731.

[13] White, B., J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. 2002. "An Integrated Experimental Environment for Distributed Systems and Networks." In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02),* (Boston), 255–270.

[14] Dudkowski, D. 2002. "Emulation Concepts for Mobile and Ad Hoc Networks." Diploma Thesis No. 2004, IPVS, University of Stuttgart.

[15] Oberle, V. "APIC timer module for Linux." University of Karlsruhe, Germany, available from *http://www.telematik.informatik.uni-karlsruhe.de/forschung/apic/apic_timer-index.html.*